Interference-Free Regions and Their Application to Compiler Optimization and Data-Race Detection

Laura Effinger-Dean

A dissertation

submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

University of Washington

2012

Reading Committee:

Dan Grossman, Chair

Luis Ceze

David Notkin

Program Authorized to Offer Degree:
Computer Science & Engineering

University of Washington

Abstract

Interference-Free Regions and Their Application to Compiler Optimization and Data-Race Detection

Laura Effinger-Dean

Chair of the Supervisory Committee:
Associate Professor Dan Grossman
Computer Science & Engineering

Programming languages must be defined precisely so that programmers can reason carefully about the behavior of their code and language implementers can provide correct and efficient compilers and interpreters. However, until quite recently, mainstream languages such as Java and C++ did not specify exactly how programs that use shared-memory multithreading should behave (e.g., when do writes by one thread become visible to another thread?). The *memory model* of a programming language addresses such questions. The recently-approved memory model for C++ effectively requires programs to be "data-race-free": all executions of the program must have the property that any conflicting memory accesses in different threads are ordered by synchronization. To meet this requirement, programmers must ensure that threads properly coordinate accesses to shared memory using synchronization mechanisms such as mutual-exclusion locks.

We introduce a new abstraction for reasoning about data-race-free programs: *interference-free regions*. An interference-free region, or IFR, is a region surrounding a memory access during which no other thread can modify the accessed memory location without causing a data race. Specifically, the interference-free region for a memory access extends from the last acquire call (e.g., mutex lock) before the access to the first release call (e.g., mutex unlock) after the access. Using IFRs, we can reason sequentially about code that contains synchronization operations. IFRs enable entirely thread-local reasoning, meaning we do not

need to have the whole program available in order to make useful inferences. We develop IFRs as a abstract concept, and also present two practical applications of IFRs.

First, IFR-based reasoning can be used to extend the scope of compiler optimizations. Compilers typically optimize within synchronization-free regions, since the data-race-freedom assumption permits sequential reasoning in the absence of synchronization. We make the observation that this rule of thumb is overly conservative: it is safe to optimize across synchronization calls as long as the calls are interference-free for the variable in question. (We say that a variable is interference-free at a call if the call falls in the interference-free region for an access to that variable.) We have developed two symmetric compiler analyses for determining which variables are interference-free at each synchronization call, thereby allowing later optimization passes to optimize in larger regions that may include synchronization.

Second, we have developed an algorithm for dynamic data-race detection based on the concept of IFRs. Data-race detection is an important problem to the programming languages community: programmers need to eliminate data races during software development in order to avoid costly bugs in production systems. Our algorithm monitors active IFRs for each thread at runtime, reporting a data race if conflicting IFRs in different threads overlap in real time. Conservative approximations of IFRs are inferred using a static instrumentation pass. We compare our algorithm to two precise data-race detectors, and determine that our algorithm catches many data races and provides better performance on most benchmarks.

As a final step, we extend the compiler analyses used in both projects to be interprocedural (i.e., analyzing more than one function at a time). Specifically, we classify functions according to their synchronization behavior, making it easier to infer when IFRs propagate through function calls. On the compiler optimization side, this change means that we can optimize across calls that contain internal synchronization. On the data-race detection side, we are able to statically infer longer IFRs, meaning that we are more likely to detect data races.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

First, I must thank my incredible advisor, Dan Grossman. Dan was always patient and insightful, pushing me to be productive without dictating exactly how I should progress. Thanks, Dan—I am a computer scientist because of you!

Thanks to the other members of my reading committee, Luis Ceze and David Notkin, for their helpful input and advice. Thanks also to Lindsay Michimoto, without whom I probably would not have passed quals, and to Maryam Fazel and Emily Bender, who served as GSR for my general and final exams, respectively.

Thanks to Hans-J. Boehm, my mentor at HP Labs, who convinced me that this "interference-free region" idea was worth investigating.

Thanks to my wonderful family, who asked me relatively rarely when I was going to graduate, and always had confidence that I could succeed. Mom, Dad, and Sean—I love you!

Finally, thanks to my friends in the UW CSE Band, with whom I've had a ridiculous amount of fun over the last six years. At last, I've climbed the "Stairway to Graduation"!

DEDICATION

To the band.

Chapter 1

INTRODUCTION

This thesis introduces the concept of *interference-free regions*, and presents two distinct and useful applications of this concept. An interference-free region is a subsection of a single thread's execution surrounding a memory read or write. While a thread is executing an interference-free region, the memory location read or written by the region's defining access cannot be written to by another thread without inducing a *data race*. That is, the two accesses, one in the interference-free region and one in another thread, will not be ordered by synchronization.

The interference-free region for an access is defined in terms of the synchronization operations surrounding that access in the thread's execution trace. In particular, the interference-free region extends from the first *acquire* synchronization before the access to the first *release* synchronization after the access. Acquire synchronizations are operations, like mutex locks, that induce orderings with operations that happened earlier in the execution. Release operations, like mutex unlocks, induce orderings with later operations in an execution.

Interference-free regions are a useful tool for reasoning about data-race-free programs. The latest C and C++ language specifications—C11 [22] and C++11 [23], respectively—require that programs be data-race-free. A program is defined to be data-race-free if all executions of the program have no data races. The reason for this requirement is that various hardware and software optimizations (basically any optimizations that may reorder memory operations) can break a program if the program has data races. Even so-called "benign" data races can lead to very subtle bugs when the program is compiled and executed. Chapter 2 will go into much more detail on the new C and C++ specifications and data-race-freedom.

Chapter 3 formally defines interference-free regions, proves their correctness, and gives illustrative examples. This work began during the author's internship at Hewlett-Packard Laboratories in 2010 and was published at the MSPC workshop in 2011 [14]. In the simplest

case, an interference-free region is simply the innermost critical section enclosing an access. However, if an access does not take place within a critical section, then the interference-free region may include synchronization operations besides the enclosing acquire and release. For example, if a memory location is accessed between two critical sections, then the interference-free region for the access includes both critical sections:

```
mtx_lock(mtx1);
...                    )
mtx_unlock(mtx1);      |
...                    |
A: r1 = *x;            } interference-free region for A
...                    |
mtx_lock(mtx2);        |
...                    )
mtx_unlock(mtx2);
```

In other words, the location being accessed can be assumed not to be modified by any other threads for the duration of both critical sections. This kind of reasoning, although useful for program optimization, is not used in any standard C and C++ compiler. This leads to the first of two applications for interference-free regions.

Our first application for interference-free regions is *compiler optimization* (Chapter 4): if we may assume data-race-freedom (as we can in C and C++), we can make existing compiler optimizations more aggressive by identifying interference-free regions within the program. (This application was included in the MSPC paper.) For example, suppose two reads of the same memory location `x` are separated by a mutex lock operation:

```
r1 = *x;
mtx_lock(mtx);
r2 = *x;
```

Normally, the compiler will assume that the mutex lock is a memory barrier, and therefore that another thread could modify the location's value in between the two reads. By reasoning about possible executions of the program, and the interference-free regions for the two reads in those executions, we can determine that any modification of `x` in another thread would be a data race. Therefore the compiler may optimize out the second read. We present two intraprocedural dataflow analyses that refine the modified/reference summaries

for acquire and release synchronization calls, and also discuss how to extend the analyses to handle barrier synchronization.

The second application is *dynamic data-race detection* (Chapter 5). Programmers use data-race detection tools to find and eliminate bugs in multithreaded programs. However, dynamic data-race detection is very expensive, usually increasing execution time by one or two orders of magnitude. This chapter presents a dynamic data-race detection algorithm based on interference-free regions. The high-level description of the algorithm is that we instrument the code with the beginnings and ends of interference-free regions (building on one of the two algorithms described in Chapter 4), then monitor executions for overlapping, conflicting IFRs. This algorithm permits no false positives and has desirable performance characteristics. In particular, most instrumentation calls require no synchronization. We compare the performance and race detection efficacy of this algorithm to two other race detectors for C and C++. This work will be published at OOPSLA 2012 [15] and is a collaboration with Brandon Lucia, Luis Ceze, Dan Grossman, and Hans-J. Boehm.

In Chapter 6, we extend the compiler analyses used in Chapters 4 and 5 to be *interprocedural*: analyzing more than one function at a time. The purpose of this extended analysis is to handle functions with internal synchronization correctly. For example, the intraprocedural analyses cannot handle even simple synchronization wrappers, so interprocedural reasoning is necessary to handle more complex programs. The interprocedural analysis does not reason about interference-free regions directly. Instead, it classifies functions according to their synchronization behavior (e.g., no synchronization, only acquire operations, etc.). This information is then used by the previous analyses to produce better results—more optimization opportunities for the analyses in Chapter 4, and more precise interference-free region boundaries for the analysis in Chapter 5.

To summarize, this thesis motivates, defines, and applies the notion of interference-free regions in shared-memory multithreaded programs. We argue that interference-free regions are a useful tool for reasoning about the behavior of data-race-free programs, and support this argument by building and evaluating two applications of interference-free regions. Our contributions improve the state of C and C++ programming in the wake of the new language standards and the increasingly important role of multithreading.

Chapter 2

BACKGROUND

Concurrent programs consist of multiple "threads" working together to accomplish a task, either time-sharing one processor or (for *parallel* programs) using more than one processor at a time. Concurrency is a powerful tool for improving the performance and/or responsiveness of a program by making better use of system resources.

Concurrency is a pressing issue in the programming languages research community. There is a massive body of research on developing concurrency mechanisms, catching concurrent bugs, and so on. Complicating this research is the fact that compiler and hardware optimizations can change the behavior of concurrent programs in surprisingly subtle ways [2, 35, 8]. To fix this problem, we need clear and precise documentation of how a language's concurrency mechanisms affect the language's semantics.

This chapter introduces shared-memory multithreading and gives some relevant background on how concurrency is defined in the language specifications of C, C++, and Java. As we shall see, there are numerous subtle issues that make this problem more difficult than one might expect.

2.1 Concurrency and Parallelism in Software

This dissertation is set in the context of *shared-memory multithreading*, a paradigm for writing parallel programs. In shared-memory multithreading, the program has multiple threads of control, each of which has its own stack and registers. Threads communicate by reading and writing memory locations in the shared heap. Figure 2.1 illustrates this idea.

When multiple threads are sharing a resource, it becomes necessary to coordinate the actions of the threads so that they do not conflict with each other. In other words, we need some way of making sure threads are not reading and writing the same memory location at the same time. Programming languages therefore provide various *synchronization*

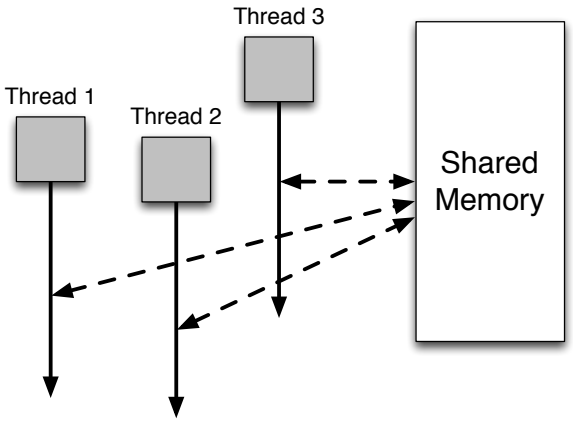

Figure 2.1: In shared-memory multithreading, multiple threads of control communicate with each other by reading from and writing to the shared memory. A gray box represents a thread's local memory. The solid lines represent the threads' executions. The dashed lines represent reads of and writes to the shared memory.

mechanisms for coordinating threads.

2.1.1 Mutual-Exclusion Locks

The most common mechanism, and the one we will use for examples throughout this dissertation, is the *mutual-exclusion lock*. Mutual-exclusion locks (called "locks" or "mutexes" interchangeably) are special resources that can be held by at most one thread at a time. A thread can request to hold a lock by calling a *lock acquire* function. Once the thread is granted the lock, it can relinquish it by calling a *lock release* function. We will use the C11 interface for mutual-exclusion locks [22]:

```
#include <threads.h> // type mtx_t defined
int mtx_lock(mtx_t *mtx);
int mtx_unlock(mtx_t *mtx);
```

`mtx_lock` acquires the lock argument, blocking until the lock is available; `mtx_unlock` releases the lock. Both functions return a code indicating whether the operation succeeded.[1]

[1]These functions fail only in unusual cases—e.g., if the mutex is configured to be held only by threads of a certain priority and the current thread's priority is too low, or if a thread tries to release a mutex it has not acquired [31].

```
struct BankAccount {
  int balance;
  mtx_t lock;
  ...
}

void deposit(struct BankAccount *b, int amt) {
  mtx_lock(&b->lock);
  b->balance += amt;
  mtx_unlock(&b->lock);
}

...
```

Figure 2.2: A simple bank account application in C.

Java, C++, and other languages that support shared-memory multithreading generally provide similar functionality (e.g., Java's `synchronized` construct).

2.1.2 Example: Bank Account

This section gives a simple example of a multithreaded program, and shows how inter-thread interference can produce unexpected results.

Consider the bank account program shown in Figure 2.2. Here, a bank account is represented by a `struct` that includes the current balance of the account. The balance is *protected* by a mutual-exclusion lock; in other words, when a thread reads or writes the balance field, it must first acquire the lock.

Figure 2.3 shows a multithreaded client of the bank account application. Two threads concurrently deposit $100 each into a bank account with initial balance 0. What are the possible final values of the balance field?

Because the `deposit()` function accesses the balance only when holding the lock, there are only two possible executions of this program, both listed in Figure 2.4. If Thread 1 gets the lock first, we get execution 2.4a; if Thread 2 gets the lock first, then we get execution 2.4b. Both possible executions result in a final balance of 200. Therefore this is an example of a *properly-synchronized* program—the threads safely coordinate access to the

Initially, `b->balance == 0`.

Thread 1	Thread 2
`deposit(b, 100);`	`deposit(b, 100);`

(a) Client code.

Thread 1	Thread 2
`mtx_lock(&b->lock);`	`mtx_lock(&b->lock);`
`b->balance += 100;`	`b->balance += 100;`
`mtx_unlock(&b->lock);`	`mtx_unlock(&b->lock);`

(b) Client code, with `deposit()` inlined.

Figure 2.3: Multithreaded client of the bank account program listed in Figure 2.2.

Thread 1	Thread 2
`mtx_lock(&b->lock);`	
`b->balance += 100;`	
`mtx_unlock(&b->lock);`	
	`mtx_lock(&b->lock);`
	`b->balance += 100;`
	`mtx_unlock(&b->lock);`

(a)

Thread 1	Thread 2
	`mtx_lock(&b->lock);`
	`b->balance += 100;`
	`mtx_unlock(&b->lock);`
`mtx_lock(&b->lock);`	
`b->balance += 100;`	
`mtx_unlock(&b->lock);`	

(b)

Figure 2.4: Possible executions of the multithreaded program listed in Figure 2.3. Note that at most one thread holds `b->lock` at a time.

```
struct BankAccount {
  int balance;
  ...
}

void deposit(struct BankAccount *b, int amt) {
  b->balance += amt;
}

...
```

Figure 2.5: Improperly-synchronized version of the bank account program listed in Figure 2.2.

shared memory.

Now consider a version of the bank account program that does not properly synchronize accesses to `balance`. Figure 2.5 lists a version in which the mutex calls have been removed. If we run the client program (Figure 2.6) with this version of the bank account instead, what are the possible values of `balance` after the execution finishes?

The answer depends on the order in which the two threads access field `balance`. In Figure 2.6b, we have broken the statement `b->balance += 100` into three component steps: (1) reading the value of the memory location corresponding to `balance` into a temporary variable; (2) adding 100 to the temporary variable's value; and (3) writing the temporary variable's value back to the `balance` field.

Figure 2.7 shows two of many possible executions. In Figure 2.7a, Thread 1 completes its execution before Thread 2 reads the value of `balance`, resulting in the expected result of 200. Figure 2.7b shows another possible execution, in which both threads read the value of `balance` before either thread writes the new value back to `balance`. In this case, the final value of `balance` is 100 because both threads see an initial balance of 0.

Therefore this program is erroneous, because (although not formally specified) we expect that depositing 100 twice should result in a net increase of 200. Instead, we found at least one execution where a deposit is "lost." This disparity is an example of how improper synchronization of threads that access the same shared memory location(s) may interfere

Initially, `b->balance == 0`.

Thread 1	Thread 2
`deposit(b, 100);`	`deposit(b, 100);`

(a) Client code.

Thread 1	Thread 2
`int tmp1 = b->balance;`	`int tmp2 = b->balance;`
`tmp1 = tmp1 + 100;`	`tmp2 = tmp2 + 100;`
`b->balance = tmp1;`	`b->balance = tmp2;`

(b) Client code, with `deposit()` inlined and `b->balance += 100` broken into component steps.

Figure 2.6: Multithreaded client of the bank account program listed in Figure 2.5.

Thread 1	Thread 2
`int tmp1 = b->balance;`	
`tmp1 = tmp1 + 100;`	
`b->balance = tmp1;`	
	`int tmp2 = b->balance;`
	`tmp2 = tmp2 + 100;`
	`b->balance = tmp2;`

Result: `b->balance == 200`.

(a)

Thread 1	Thread 2
`int tmp1 = b->balance;`	
`tmp1 = tmp1 + 100;`	
	`int tmp2 = b->balance;`
	`tmp2 = tmp2 + 100;`
`b->balance = tmp1;`	
	`b->balance = tmp2;`

Result: `b->balance == 100`.

(b)

Figure 2.7: Two possible executions of the bank account client program listed in Figure 2.6.

with one another.

2.1.3 Data Races

We say that the executions listed in Figure 2.7 have *data races*: two threads access a shared memory location (here the `balance` field) in such a way that the accesses are not ordered by synchronization and at least one access is a write. For example, the statement `int tmp1 = b->balance` in Thread 1 forms a data race with `b->balance = tmp2` in Thread 2. Note that even the "correct" execution in Figure 2.7a has data races (e.g., `b->balance = tmp1` races with `b->balance = tmp2`. This indicates that the execution is, in a sense, broken, even though the memory accesses just so happened to occur in an order that produced the correct answer.

In general, data races should be considered errors, even if they may not always produce an incorrect result. As we have already seen, data races can result in unexpected behavior in the event of unfortunate thread interleavings. In addition, we shall see in Section 2.2 that data races can result in even more insidious errors as a result of unexpected interactions with hardware and compiler optimizations.

2.1.4 Synchronizes-With Orderings

When writing multithreaded code, it is important to use proper synchronization such that no possible executions have data races. The program in Figure 2.3 prevents data races using mutual-exclusion locks. Neither of the possible executions in Figure 2.4 have data races, because accesses to `balance` between the two threads are ordered by synchronization. Formally, we say that a lock release *synchronizes-with* a later acquire of the same lock. For example, in Figure 2.4a, Thread 1's lock release (`mtx_unlock(b->lock)`) synchronizes-with Thread 2's lock acquire (`mtx_lock(b->lock)`). This ordering between the unlock and lock actions induces a partial ordering over the memory accesses in the two threads: all the actions in Thread 1's execution "happen before" the actions in Thread 2.

This concept of synchronizes-with orderings extends to types of synchronization beyond mutual-exclusion locks. In general, we say that any synchronization operation that

Operation	Synchronization type
`mtx_lock`	Acquire
`mtx_unlock`	Release
`thrd_create`	Release
`thrd_join`	Acquire
`atomic_store`	Release
`atomic_load`	Acquire

Table 2.1: Synchronization operations.

synchronizes-with earlier operations is an *acquire operation* (e.g., `mtx_lock` is an acquire operation because it synchronizes-with earlier release operations). Acquire operations induce orderings on the operations in the same thread that come after the acquire. Similarly, any synchronization operation that synchronizes-with later operations is a *release operation* (e.g., `mtx_unlock`). Release operations induce orderings on the operations in the same thread that come before the release.

The next few sections will have more examples of acquire and release operations; Table 2.1 lists the operations we will discuss and their synchronization types.

2.1.5 Thread Fork and Thread Join

Java, C and C++ all use the same language mechanism to create threads: a special *fork* operation that spawns a new thread. In C11, the fork operation is a library function with the following signature [22]:

```
#include <threads.h> // defines thrd_t and thrd_start_t
int thrd_create(thrd_t *thr, thrd_start_t func, void *arg);
```

Here `thr` is a pointer to the thread ID (initialized by `thrd_create`), `func` is the function for the new thread to execute, and `arg` is the argument to pass to the function.

The thread fork operation synchronizes-with the first operation of the newly-created thread. Therefore thread fork is a *release* operation: it induces ordering on operations that came before it. Practically, this means that a thread can initialize shared state before

```
int worker(void *arg) {
  int me = (int) arg;
  ... // do work
}

int main() {
  thrd_t tids[NUM_THREADS];

  for (int i = 1; i < NUM_THREADS; i++) {
    thrd_create(&tids[i], worker, (void *) i);
  }

  worker(0);

  for (int i = 1; i < NUM_THREADS; i++) {
    int res;
    thrd_join(tids[i], &res);
  }

  ... // process result
}
```

Figure 2.8: An example of thread fork/join. This program spawns `(NUM_THREADS - 1)` worker threads, then waits for them to complete their tasks.

forking off threads, and those initialization writes will not conflict with reads or writes in spawned threads.

Joining a thread means waiting for the thread to complete. This is useful, for example, if the main thread forks several worker threads, and needs to aggregate their results once all threads have finished (a model typically called *fork/join parallelism*). C11 supports thread join as follows:

```
#include <threads.h> // defines thr_t
int thrd_join(thrd_t thr, int *res);
```

`thr` is the thread ID and `res` is where the return value of the thread is stored. `thrd_join` synchronizes-with the end of the joined thread, so it is an acquire operation.

Figure 2.8 gives a simple example of a multithreaded program that uses fork/join parallelism to farm out a large task to worker threads.

2.1.6 Volatile Variables

The last type of synchronization we will discuss, available in Java, C and C++, is *volatile variables*. By definition, concurrent accesses to volatile variables do not constitute a data race. In C and C++, volatile variables are called *atomics*. They have the following interface (here specialized for integers, but available for many types):

```
#include <stdatomic.h> // defines atomic_int, atomic_long, ...
void atomic_store(volatile atomic_int *object, int desired);
int atomic_load(volatile atomic_int *object);
```

Volatile loads synchronize-with previous volatile stores to the same location. Therefore volatile loads are acquire operations and volatile stores are release operations.

2.2 Motivating Memory-Consistency Models

In order to make these concurrent features usable by programmers, they must be integrated into the specification of the language such that the semantics of concurrent programs are well-defined and straightforward to reason about. A *memory-consistency model* (or *memory model*) is the component of the language specification that defines how threads interact through shared memory. In particular, the memory model defines which values may be returned by read operations. Memory models were originally defined at the hardware level; we do not discuss hardware models in detail and instead refer the reader to Adve and Gharachorloo's tutorial [2].

The strongest, most intuitive semantics for multithreaded programs is *sequential consistency*, originally described by Lamport [25]. Sequential consistency requires that the memory actions of each thread be interleaved in a total order consistent with the program order in each thread, such that a read always sees the most recent write to the same location in this total order. From the programmer's perspective, sequential consistency is the easiest semantics to reason about and provides the optimal level of programmability. However, modern high-level programming languages have sophisticated compilers that rely on complex code transformations to achieve good performance. These code transformations may reorder memory operations, potentially violating sequential consistency. Ubiquitous optimizations like common subexpression elimination and hoisting operations out of loops are

Initially, `x == 0` and `y == 0`.

Thread 1	Thread 2
`r1 = x;`	`r2 = y;`
`if (r1 != 0)`	`if (r2 != 0)`
`  y = 42;`	`  x = 42;`

`r1 == 42` and `r2 == 42` is illegal.

Figure 2.9: Data-race-free programs must have sequentially-consistent semantics.

essentially forms of memory reordering. Therefore, modern languages have adopted *relaxed memory-consistency models*, which provide weaker guarantees than sequential consistency so that key compiler optimizations are legal.

The memory-consistency model of a language acts as a contract between the programmer and the compiler. The approach taken by the vast majority of models is for *correctly synchronized* code to behave as if the language supported sequential consistency, while *incorrectly synchronized* programs may have weaker semantics. A program is correctly synchronized if no sequentially consistent executions of the program have data races (i.e., two threads accessing the same location at the same time where at least one access is a write). As long as a program is data-race-free, the compiler guarantees that any optimizations will not change the semantics of the program. This compromise, called the *data-race-free guarantee*, exploits the fact that most compiler optimizations are safe for data-race-free programs. Specifically, compilers must avoid optimizing across synchronization calls or introducing new data races.

For example, consider the program in Figure 2.9 (reprinted from the Java Memory Model paper [35]). Although the program contains writes to `x` and `y`, they will never occur in a sequentially consistent execution of the program. Therefore the program is data-race-free, and the only legal outcome is `r1 == 0` and `r2 == 0`. The compiler must be careful not to move either write earlier, because such a transformation could violate the semantics of the program.

In the last decade, three popular mainstream languages have defined precise memory-

Initially, `x == 0` and `y == 0`.

Thread 1	Thread 2
`r1 = x;`	`r2 = y;`
`y = 1;`	`x = r2;`

`r1 == 1` and `r2 == 1` is legal.

Figure 2.10: Legal compiler optimizations may introduce non-sequentially-consistent behavior.

Initially, `x == 0` and `y == 0`.

Thread 1	Thread 2
`r1 = x;`	`r2 = y;`
`y = r1;`	`x = r2;`

Incorrectly synchronized, but we want to disallow `r1 == 42` and `r2 == 42`.

Figure 2.11: In safe languages like Java, compiler optimizations must not introduce values out of thin air.

consistency models: Java [35, 19], C, and C++ (both of which use the same underlying model) [11, 22, 23]. The existence of these two models, as well as the considerable community effort that went into developing both of them, stresses the importance of this problem. Both models take the data-race-free approach to provide programmability while also allowing reasonable compiler optimizations. Given that both models support the data-race-free guarantee, the differences between the two models define a clear design space. The Java memory model must support Java's strong language guarantees of security and safety. Therefore the model provides reasonably strong semantics even to programs that are not correctly synchronized. C++ is an unsafe language, so its model does not define any semantics for incorrectly synchronized programs, focusing instead on simplicity and flexibility. The C++ model has the advantage of having a concise specification, as well as allowing more compiler optimizations than Java. We discuss these and other models in Section 2.3.

2.2.1 Compiler Optimizations

We illustrate how compiler optimizations can affect the semantics of multithreaded programs via two simple examples. Consider the program shown in Figure 2.10, which is reprinted from the JMM paper [35]. Here `x` and `y` are shared variables, and `r1` and `r2` are local variables. In a sequentially consistent language, the outcome `r1 == 1` and `r2 == 1` is not legal, because at least one of the reads must execute before either of the writes. But an optimizing compiler may reorder independent operations in a single thread; if the compiler reorders the read and the write in Thread 1, the outcome `r1 == 1` and `r2 == 1` is possible. Because this optimization is entirely standard and the spurious result does not violate any safety guarantees, this example should be legal for any reasonable language-level memory model.

In the context of a safe language like Java, compiler optimizations can have profound implications for security. Consider Figure 2.11, also from [35]. This example is similar to Figure 2.10, except that the values written to `x` and `y` are now the values read from `y` and `x`, respectively. Now, consider a compiler optimization that transforms Thread 1 as follows:

```
y = 42;
r1 = x;
if (r1 != 42) {
  y = r1;
}
```

If Thread 2 is transformed symmetrically, the outcome `r1 == 1` and `r2 == 42` is allowed. Note that the transformed code is safe in a single-threaded setting, so a compiler could theoretically implement this transformation (although in practice it is very unlikely). The problem here is that the value 42 appeared "out of thin air": 42 is never assigned to `x` or `y` in the original program. Practically, the issue is that the compiler added a speculative write whose value was not justified by the original program.

Out of thin air (OoTA) reads are a potential security hole in a safe language like Java. Hypothetically, untrusted code could gain access to thread-local objects or confidential data (e.g., a user's password). Therefore, Manson et al. (the authors of the Java memory model) concluded that such optimizations, no matter how unlikely, must be explicitly disallowed

by the memory model [35]. In contrast, C and C++ are unsafe languages, so they make no guarantees about OoTA reads [11].

2.3 Programming-Language Memory Models

This section discusses the design of memory-consistency models for programming languages in the context of two important examples: the Java memory model [35] and the C/C++ memory model [11]. These models represent two extremes on the design spectrum for memory models. We will describe the Java and C/C++ models using a hybrid of both models' formalizations [35, 11].

2.3.1 Definitions

A running program emits a series of actions. The kinds of possible actions vary depending on the synchronization mechanisms provided by the language. For the moment, we will assume that the language supports shared mutable variables x and non-reentrant mutual-exclusion locks l.[2] For brevity, we omit synchronization mechanisms other than locks; we address volatile variables briefly in Section 2.3.2.

Let A be a set of actions, where each action is a triple of a thread ID t, a kind of action k, and a UID (unique ID) u: (t, k, u). Kinds of actions include reads and writes to shared variables and acquires and releases of non-reentrant locks:

$$\text{Kinds } k ::= \mathsf{read}(x) \mid \mathsf{write}(x) \mid \mathsf{mtx\_lock}(l) \mid \mathsf{mtx\_unlock}(l)$$

We specify the memory model for a language by giving a set of constraints, or axioms, over executions. Some models are concise and easy to state (e.g., sequential consistency) while models like the JMM are considerably more complex. This axiomatic approach is not the only way to specify a memory model, but it is the most prevalent in the literature, both because the initial formulations of memory models for hardware took this approach [2] and because it is versatile enough to describe a large variety of models.

---

[2]Non-reentrant locks cannot be reacquired by the thread holding the lock.

We describe several axioms shared by all the memory models we will discuss. These axioms rely on several auxiliary structures, which we shall summarize after defining the axioms.

1. Let the writes-seen function W be a map from UIDs of read actions in A to UIDs of write actions in A, and let the values-written function V be a map from UIDs for write actions in A to values. These two functions must be defined properly: W is defined on UID u if and only if there exists $(t, k, u) \in A$ and k is a read, and V is defined on UID u if and only if there exists $(t, k, u) \in A$ and k is a write. Moreover, for all $(t, k, u) \in A$ where k is a read and $(t', k', W(u)) \in A$, k and k' must access the same variable and k' must be a write. (This formulation is used by the Java Memory Model; in the C++ model, all read and write actions include the value read or written.)

2. The synchronization operations in A must obey mutual exclusion. Assume there is a strict total order $<_{\mathsf{so}}$ over UIDs for synchronization operations (that is, acquire and release operations) in A. Then mutual exclusion imposes the following constraints: acquires and releases of each lock must alternate in the total order $<_{\mathsf{so}}$; the first action for each lock in $<_{\mathsf{so}}$ must be an acquire; and the thread that releases a lock must also have been the most recent thread (according to $<_{\mathsf{so}}$) to have acquired that lock.

3. The execution must satisfy *intra-thread consistency*. Let the *program order* $\leq_{\mathsf{po}}$ be a partial order over UIDs such that $u_1 \leq_{\mathsf{po}} u_2$ only if u_1 and u_2 correspond to the same thread ID. Note that the actions of a single thread need not be totally ordered by $\leq_{\mathsf{po}}$; for example, C and C++ leave undefined the order of evaluation of function arguments [11]. Then the program (we assume there is some program P associated with A) must be able to produce the actions in A in the order given by $\leq_{\mathsf{po}}$, according to the sequential (i.e., "intra-thread") semantics of the language. The sequential semantics uses the functions W and V to get a value for each read operation: the value returned by a read u is $V(W(u))$. Similarly, for every write action u, the value written by the sequential semantics must match the value given by $V(u)$.

4. The program order and synchronization order must be consistent. That is, the reflexive transitive closure of the union of $\leq_{\mathsf{po}}$ and $<_{\mathsf{so}}$ is antisymmetric.

The Java and C/C++ models include other axioms (e.g., for variable initialization), but these four axioms are the most broadly relevant. Gathering the various definitions together, we say that an execution E consists of a program P, a set of actions A, a program order $\leq_{\mathsf{po}}$, a synchronization order $<_{\mathsf{so}}$, a writes-seen function W, and a values-written function V: $E = (P, A, \leq_{\mathsf{po}}, <_{\mathsf{so}}, W, V)$. Furthermore, executions satisfying the four constraints listed above are *well-formed*.

The intra-thread consistency requirement is interesting in that it is unique to language-level memory models. This requirement is the only link between the sequential and concurrent features of the language, so it is crucial in determining the possible legal executions of a program. Hardware memory models may include constructs for data or control dependencies, but they generally ignore details like values being read and written or the semantics of non-memory instructions (e.g., [3]). (An exception is the x86-CC model [41], which includes an instruction semantics as well as an axiomatic model.) We omit the syntax and sequential semantics of the program P; for our purposes, the program is effectively a *predicate over executions*. (This observation is due to Aspinall and Ševčík [5], who proved the DRF guarantee for the JMM without having to define a sequential semantics.) However, defining the program syntax and semantics is crucial when reasoning about the correctness of compiler optimizations [8, 46].

We will use the notion of a well-formed execution to describe sequential consistency, the Java memory model, and the C++ memory model. For the purposes of exposition, we discuss the C++ model before the Java model, which is considerably more complex. The Java model actually predates the C++ model. The Java section is not necessary for understanding the remainder of this dissertation.

2.3.2 Volatile Variables

We can add volatile variables to our semantics by augmenting the syntax of actions:

$$\text{Kinds } k ::= ::= \ldots \mid \mathsf{readv}(t, v) \mid \mathsf{writev}(t, v)$$

Volatile reads and writes are included in the synchronization order, and we extend the definition of well-formedness to require that volatile reads and writes exhibit sequentially-consistent semantics. The happens-before order (defined in Section 2.3.4) is extended to include edges from writes to reads of volatile variables.

Volatiles differ from normal variables in that they define ordering constraints. In particular, the compiler cannot optimize away volatile operations using, for instance, redundant read elimination, because removing a volatile operation also changes the happens-before relation of an execution [35].

2.3.3 Sequential Consistency

Sequential consistency is our most basic memory model, defining the most intuitive semantics possible for a multithreaded program. In short, the actions of each thread are interleaved in a global order. This global order must be consistent with the program and synchronization orders, and each read must see the most recent write in the global order. Formally, we say a well-formed execution $E = (P, A, \leq_{\mathsf{po}}, <_{\mathsf{so}}, W, V)$ is sequentially consistent if there exists a total order $\leq$ over the UIDs in A such that:

1. $u_1 \leq_{\mathsf{po}} u_2$ implies that $u_1 \leq u_2$;
2. $u_1 <_{\mathsf{so}} u_2$ implies that $u_1 < u_2$;
3. For all reads $u_r \in A$, there does not exist a write $u_w \in A$ such that u_w and u_r access the same variable and $W(u_r) < u_w < u_r$.

This axiomatic specification of sequential consistency is somewhat oblique, but the intuition is for the actions of all threads to interleave in the natural way.

2.3.4 Data Races

Most memory models provide the data-race-free guarantee: programs that are "correctly synchronized" will have simple semantics (i.e., sequential consistency). We have already defined sequential consistency; now we must define the notion of a *correctly synchronized* program. In short, a correctly synchronized program is a program that has no data races.

As discussed in Section 2.1.3, a data race occurs when two threads access the same

location concurrently, and at least one of those accesses is a write. It is difficult to reason about the semantics of a program in the presence of data races, even if the language supports a strong memory model such as sequential consistency. Therefore, most memory models consider a program with data races to be fundamentally erroneous. Following Boehm and Adve [11], we distinguish between two competing definitions of data race, one of which formalizes the intuitive definition given earlier and one of which departs from that intuition. For many languages, these two definitions are equivalent.

A *type 1 data race* is defined in terms of actions that could have happened at the same time in a sequentially consistent execution. Let $E = (P, A, \leq_{\mathsf{po}}, <_{\mathsf{so}}, W, V)$ be a sequentially consistent execution with total order $\leq$. Then E contains a type 1 data race if there exist two read/write actions (t_1, k_1, u_1) and $(t_2, k_2, u_2) \in A$ such that $t_1 \neq t_2$, k_1 and k_2 access the same location, at least one of k_1 and k_2 is a write, and u_1 and u_2 are consecutive in the sequential order.

A *type 2 data race* is defined in terms of a partial order on a program execution called the *happens-before order*. Following the Java memory model [35], we can define the happens-before order as follows (the C++ definition is similar). Two UIDs u_1 and u_2 in a well-formed execution $(P, A, \leq_{\mathsf{po}}, <_{\mathsf{so}}, W, V)$ are ordered by the *synchronizes-with order* $<_{\mathsf{sw}}$ if $u_1 <_{\mathsf{so}} u_2$ and there exists a lock l such that u_1's corresponding action is $\mathsf{mtx\_unlock}(l)$ and u_2's corresponding action is $\mathsf{mtx\_lock}(l)$. The happens-before order $\leq_{\mathsf{hb}}$ is then defined as the reflexive transitive closure of the union of $\leq_{\mathsf{po}}$ and $<_{\mathsf{sw}}$. Finally, two read/write actions u_1 and u_2 form a type 2 data race if u_1 and u_2 are performed by different threads, the actions corresponding to u_1 and u_2 access the same location, at least one of u_1 and u_2 is a write, and u_1 and u_2 are not ordered by happens-before; that is, neither $u_1 \leq_{\mathsf{hb}} u_2$ nor $u_2 \leq_{\mathsf{hb}} u_1$.

Note that type 2 data races are more general than type 1 data races. If an execution has a type 1 data race, the two actions in question are clearly not ordered by happens-before, and therefore also form a type 2 data race. However, an execution may have a type 2 data race but no type 1 data race (if the racy actions do not happen to occur consecutively in the total order).

For many languages, the distinction between type 1 and type 2 data races is moot, because they can be proved equivalent (a proof for C/C++ is given in Theorem 8.2 of

the PLDI paper introducing the C++11 model [11]). Proving this equivalence consists of finding an execution of the program P in which the actions forming the type 2 data race are consecutive. As we shall see in Section 2.3.5, some types of synchronization invalidate this equivalence. In such instances, the memory model is defined in terms of type 2 data races, making the language less usable to the programmer.

We highlight the difference between type 1 and type 2 data races because they illustrate a design choice where the programmer and the compiler have different demands. Programmers can more easily reason about type 1 data races, because the idea of two actions occurring simultaneously is simple and intuitive. Compiler designers require the precision of the happens-before relation to reason carefully about code transformations. Therefore it is ideal for languages to support type 1/type 2 data race equivalence if possible.

2.3.5 The C++ Memory Model

Given the definition of a type 1 data race, the C/C++ memory model is simple [11]:

1. If a program is type 1 data-race-free, then all executions of that program are sequentially consistent.
2. If a program is not type 1 data-race-free, then the program's semantics is undefined.

In effect, the model declares that "there are no benign data races" in C and C++ [11]. A program that includes a data race could (figuratively) set the computer on fire. Drawbacks of this approach include that debugging racy programs is difficult, and of course that we have none of the safety or security guarantees provided by Java's model. Boehm and Adve, in their overview of the model [11], argue that data races are too difficult to deal with in an unsafe language like C++, and give an example in which a data race effectively causes the program to start executing code at an arbitrary address. Leaving the semantics of incorrectly synchronized programs undefined significantly simplifies the work of the compiler. For example, eliminating a redundant read is safe as long as there are no synchronization operations between the two reads, because no other thread could possibly update the value of the location without causing a data race. (This insight will be crucial to our work on compiler optimization in Chapter 4.)

Complications

As observed by Boehm [8], adding a `mtx_trylock` primitive (which attempts to acquire a lock and succeeds if and only if the lock is available) to the set of synchronization primitives available to the language impacts the semantics of the memory model. To be precise, the use of `mtx_trylock` invalidates the equivalence of type 1 and type 2 data races, unless we strengthen the happens-before relation to include edges between acquire and trylock operations. However, this change would disallow a key compiler optimization, namely moving memory actions into critical sections by reordering them with lock acquires (i.e., roach-motel reordering [35]). The C/C++ model instead redefines the semantics of `mtx_trylock` such that it can spuriously fail to acquire the lock even if the lock is available, neatly sidestepping this semantic issue [11].

The C/C++ model is further complicated by the assumption that there is a total order over all synchronization operations in an execution ($<_{\mathsf{so}}$ in our formalism). Some architectures do not support this requirement, which makes it impossible to reason about the semantics of a program with respect to a sequentially consistent execution. Instead, the C/C++ memory model provides an "expert" version in which synchronization operations (or, in their terminology, "low-level atomics") do not have a total global order. This model is considerably more complicated than the model without low-level atomics, but allows for finely-tuned optimization of low-level concurrent code [11, 22, 23].

Despite these complications, the C/C++ model is, for the vast majority of programmers, very simple. Most programmers must simply avoid type 1 data races, and the compiler will guarantee sequentially consistent semantics. This model represents an interesting point in the design space of memory models, where we sacrifice *security* and *safety* in favor of *simplicity* and *flexibility*.

2.3.6 The Java Memory Model

This section discusses the Java memory model. Although the information presented provides useful background, it is not essential since the rest of the dissertation focuses on C and C++.

The Java memory model (JMM), developed by Manson, Pugh, and Adve with the input

of the Java community [35], differs from the C/C++ model in that the Java model defines the semantics of all programs, even those with races. This requirement comes from Java's status as a type-safe language with strong security guarantees. The JMM is the state of the art for defining a precise and secure memory model for a type-safe language.

The JMM is a data-race-free model, guaranteeing sequentially consistent semantics for correctly synchronized programs. (Unlike C and C++, the JMM defines data races using the type 2 definition; type 1 and type 2 data races are not equivalent in Java [9].) The main hurdle in formulating the JMM was to define safe but efficient semantics for incorrectly synchronized code.

Out of Thin Air Reads

In order to maintain Java's security guarantees, the memory model goes to great lengths to disallow so-called *out of thin air* (OoTA) reads. OoTA reads (as in Figure 2.11) return values that do not occur in any reasonable interpretation of the program semantics, even when allowing for standard compiler transformations. If values were permitted to appear out of thin air, it could theoretically result in threads getting access to confidential data by exploiting incorrectly synchronized programs. Although real compilers are unlikely to compile programs such that values appear out of thin air, it is critical that the memory model explicitly disallow such behavior.

An intuitive but imprecise definition of Java's OoTA restriction is as follows: *a thread must never see a value for a read that could never be written to that location in a sequentially consistent execution.* However, the JMM's definition of OoTA is somewhat more restrictive: it states that "early execution of an action does not result in an undesirable causal cycle if its occurrence is not dependent on a read returning a value from a data race" [35]. This definition is confusing, but the intuition is that that speculative executions (which are used to justify early writes) should not rely on one possible outcome of a data race if the final execution relies on a different outcome.

Overview of the JMM

The JMM disallows causal cycles by requiring that every execution be justified by a series of intermediate executions. Formally, in order to justify a well-formed execution E, we must build up a series of executions $E_0 \ldots E_n$ where $E = E_n$ and each execution has the same program P.[3] Execution E_i "commits" a set of actions $C_i \subseteq A_i$, such that $C_0 = \{\}$, $C_0 \subset C_1 \subset \ldots ... \subset C_n$ and $C_n = A$. In other words, each successive execution in the series justifies one or more of the actions in the final execution, cumulatively justifying the entire execution.

Given this notion of a set of *justifying executions*, we can characterize the JMM as follows: the JMM is a set of axioms which, when taken together, describe all legal series of justifying executions. A (very) simplified explanation of the axioms is that each justifying execution must be both well-formed (as defined in Section 2.3.1) and *well-behaved*, which means that all reads see writes that happen-before them. However, committed reads do not need to be well-behaved, which allows executions to have read-write data races. The JMM, then, defines a *commit procedure* wherein we iteratively commit non-well-behaved reads, fixing the write seen by the committed read in order to disallow causal cycles. (We must also commit all other actions in the execution, including writes, synchronization operations and well-behaved reads, but the trick is in committing non-well-behaved reads.)

An alternative formulation of the model, due to Aspinall and Ševčík [6], is to characterize the JMM's commit procedure as a process in which we commit data races one at a time. For each intermediate execution in the series of justifying executions, we choose a data race to commit, and commit all actions involved in the data race. For example, Figure 2.10 has two data races: Threads 1 and 2 have read-write data races on `x` and `y`. We can justify the behavior `r1 == 1` and `r2 == 1` as follows. First, we commit the data race on `y` (i.e., Thread 1's write and Thread 2's read) using an execution in which Thread 2 reads Thread 1's write of 1 to `y`. (Note that we could not use the commit procedure to commit a write of 42 to `y` in Figure 2.11, because we must justify the value for the write using a well-behaved execution.) Then we commit the data race on `x` (i.e., Thread 1's read and Thread 2's

[3] Assume for simplicity that the executions are finite; the formalism extends to infinite executions.

write), also using value 1. The commit procedure therefore allows some non-sequentially-consistent executions, but only those in which the values returned by reads can be justified in a non-cyclic way, giving us a notion of *causality*.

Given the definition of the JMM, it is possible to show that type 2 data-race-free programs exhibit sequentially-consistent semantics [35] (and mechanized in [21, 5]). Unlike C++, in which the memory model and the DRF guarantee are one and the same, the DRF guarantee is merely a *consequence* of the Java model. In other words, the DRF guarantee is not part of the language definition; instead, given the language definition, one can prove a theorem stating that the model satisfies the DRF guarantee.

Problems with the JMM

The JMM is difficult to understand. The commit procedure is considerably more complex than the simplified explanation given in the previous section. For example, there are several axioms that constrain the $\leq_{\mathsf{hb}}$ and $<_{\mathsf{so}}$ orders for the various intermediate executions. Moreover, the intermediate executions are not actually sequentially consistent—they instead use a model similar to weak ordering [3] called *happens-before consistency*. It appears that only a few people fully understand the JMM, which lessens the likelihood that compiler writers will implement it correctly. One difficulty is that the JMM defines which executions are legal, but it is unclear how to reverse this definition in order to show that a given execution is *not* legal.

A more concrete problem is that a number of bugs in the JMM have been identified in the seven years since its release as part of Java 5.0. The JMM was designed to support a variety of common compiler optimizations. The JMM paper [35] includes a proof that independent reordering of operations is legal, but a counterexample was given by Cenciarelli et al. [13]. A fix for this bug was proposed by Aspinall and Ševčík [6], but the same authors also uncovered issues with roach-motel reorderings and other important optimizations [46] which had previously been assumed to be legal by Java compiler experts [28]. These problems with the JMM definition were one of the reasons the C++ model chose not to give semantics to racy programs [11]. The Java Language Specification has yet to address any of these

problems [19].

2.4 Conclusion

We have now reviewed the C/C++ and Java memory-consistency models. The two models are of great practical importance and also nicely illustrate the current spectrum of language memory models. The rest of this dissertation concentrates on the C/C++ model, which (as discussed) requires program to be data-race-free. We will address two research questions raised by the new model:

1. How can the compiler take advantage of the DRF assumption to optimize programs more effectively?
2. How can the programmer ensure that their program is data-race-free?

In the remaining chapters, we introduce our key insight (*interference-free regions*), and apply this insight to compiler optimization and data-race detection.

Chapter 3

INTERFERENCE-FREE REGIONS

This chapter introduces a simple but powerful abstraction called *interference-free regions*. The observation we make is that when a variable x is accessed, its value cannot be changed by another thread between the immediately preceding acquire synchronization call and the immediately following release synchronization call. A powerful consequence of this generalization is that the interference-free regions for different accesses to the same variable may overlap, revealing previously unrecognized optimization opportunities.

Our analysis is straightforward, but general and applicable to a number of interesting cases. We hope that the discussion sheds light on issues that have not been well understood prior to our work.

3.1 Interference-Free Regions

It is well-known that, in the absence of data races, when a variable x is accessed, its value cannot be changed by another thread between the immediately preceding and immediately following synchronization operations. If another thread could modify the variable between those two points (i.e., within a *synchronization-free region*), then the two accesses of x would not be ordered by synchronization, and therefore they would form a data race. Similarly, if x is written within a synchronization-free region, it cannot be read or written by another thread within that region.

These observations are fundamental to code optimization in current compilers. They allow compilers for languages like C and C++ to largely ignore the presence of threads when transforming synchronization-free code. No other thread can detect such transformations without introducing a data race. As long as synchronization operations are treated as "opaque" (i.e., as potentially modifying any memory location), and no speculative operations, and hence no new data races, are introduced, safe sequential transformations remain

correct in the presence of multiple threads.

We concern ourselves with regions of code that are *not* synchronization-free. Can we regain some sequential reasoning even in the presence of synchronization? To illustrate the problem we are trying to address, consider:

```
mtx_lock(*mtx_p);
...
mtx_unlock(*mtx_p);
```

where `mtx_p` is a shared global pointer of type `mtx_t **`.[1] We will also assume, for purposes of the examples, that ellipses represent synchronization-free code that does not modify any of the variables mentioned in the example. Is it the case that both instances of `mtx_p` always refer to the same lock? Or could another thread modify `mtx_p` in the interim?

The prohibition against data races often allows us to answer such questions, *without analyzing code that might be run by other threads.* Without such a prohibition, and without knowledge about other threads in the system, we clearly could not guarantee anything about concurrent updates of `mtx_p`. Moreover, reasoning about synchronization-free regions is insufficient in this case: the lock is acquired between the first and second load of `mtx_p`, and hence the two references are not in the same synchronization-free region. Nevertheless, we argue that the data-race-freedom assumption is strong enough to establish, for this example, that another thread cannot concurrently modify `mtx_p`.

We make an easily provable, but foundational, and apparently normally overlooked observation: the region of code during which other threads cannot modify a locally accessed variable may often be extended in both directions beyond the access's enclosing synchronization-free region. In particular, we can extend the boundary of the region backwards in the trace past any earlier release operations, such as `mtx_unlock()` calls, and forwards through any later acquire operations, such as `mtx_lock()` calls. To put it another way, the variable cannot be modified by other threads in the region between the most re-

[1] Typically, a global mutex variable will have type `mtx_t` or `mtx_t *`. However, LLVM (the compiler we will use in Chapters 4–6) transforms the program during compilation by adding an extra level of indirection to the type of all global variables, so the mutexes become constant variables of type `mtx_t *` or `mtx_t **`, respectively. We assume in our code examples that all globals have been transformed in this way. This also means that loads and stores are easily identifiable, since they use the `*` operator.

cent acquire operation and the next release operation. We call this extended region the *interference-free region* for an access.

Thus, in our example, the interference-free region corresponding to the initial load of `mtx_p` extends through the immediately following `mtx_lock()` call, and includes the second load of `mtx_p`, guaranteeing that both loads must yield the same pointer. Here we have separated the loads of `mtx_p` from the synchronization operations and labeled the first load, but the idea is the same:

```
A: mtx_t *tmp1 = *mtx_p;
mtx_lock(tmp1);
...
mtx_t *tmp2 = *mtx_p;
mtx_unlock(tmp2);
```

} interference-free region for A

We believe that the notion of an interference-free region is a fundamental observation about the analysis of multithreaded code, in that it gives us a much better characterization of the applicability of single-threaded techniques. This is strictly more general than synchronization-free regions, which do not discern between different types of synchronization operations. Note that we make these deductions with no specific information about other threads in the execution; we are relying only on the data-race-freedom requirement imposed by the language.

Given a memory access in an execution, we can infer the interference-free region for that access. In the execution trace for a single thread in Figure 3.1, the IFR for access `X` extends backwards through line `A` and forwards through line `B`. Any conflicting write must happen-before the lock of `mtx1` or happen-after the unlock of `mtx6`. The identity of the locks being acquired or released is irrelevant; we simply identify acquire and release operations.

3.1.1 Overlapping Regions

We extend our reasoning about interference-free regions by considering cases in which two or more regions for the same variable *overlap*. If x cannot be changed in either interval a or interval b, and a and b overlap, then clearly it cannot change in $a \cup b$.

For example, suppose there is a critical section nested between two accesses, as in Figure 3.2. In this case, the interference-free region for load A extends forwards into region B. The

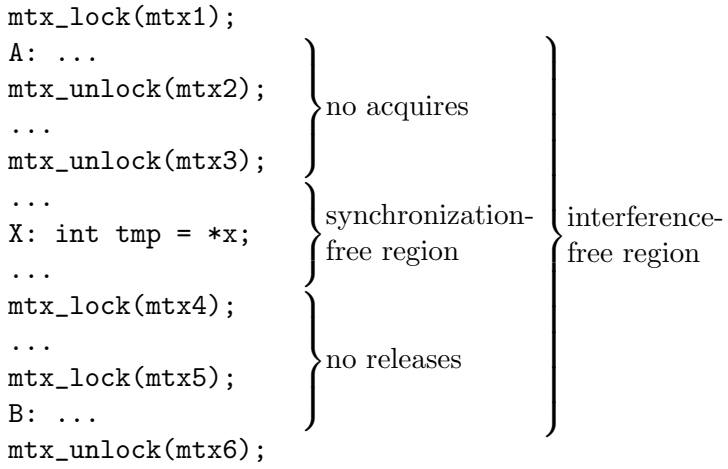

Figure 3.1: An interference-free region in a thread trace. Ellipses are synchronization-free code.

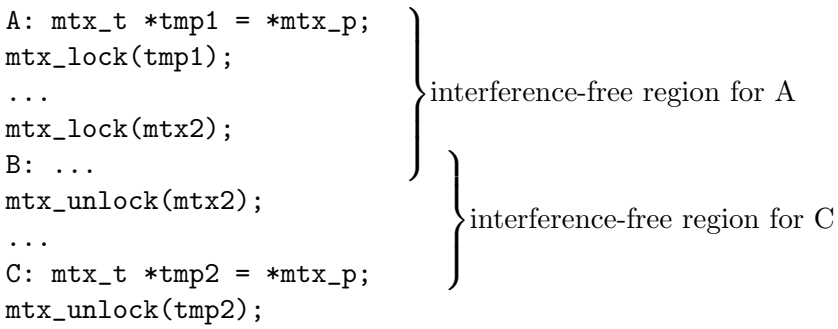

Figure 3.2: The interference-free regions for accesses A and C overlap, despite the intervening critical section.

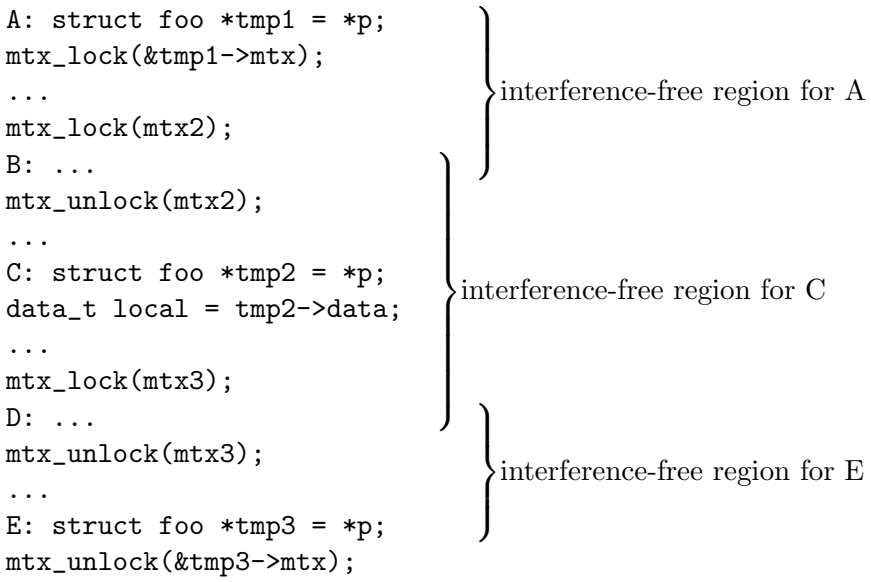

Figure 3.3: Here, the load of `p` at line C means that `p` is interference-free during both nested critical sections.

interference-free region corresponding to load C extends backwards, past the unlock into the region B. Thus `mtx_p` must be interference-free for the entire region, and we can conclude that all locks acquired are released.

The above reasoning does not generally apply if there is more than one nested critical section in a row. However, there are cases for which we can derive similar results even then. Consider the common case in which, rather than `mtx_p`, we have a pointer `p` to a structure that includes both a mutex and some data, with the program as shown in Figure 3.3. The program includes three loads of `p`, at lines A, C, and E. The interference-free region from the load of `p` at line A extends forward through region B. The interference-free region from the load of `p` at line C extends backwards through B and forwards through D. The interference-free region from the load of `p` at line E extends backwards through region D. Thus we conclude that `p` cannot be modified by another thread.

More generally, we may conclude a variable x is interference-free along a section of the execution trace if it is accessed between every pair of release and subsequent acquire

```
while (...) {
  A: r1 = *x;
  ...
  mtx_lock(mtx);
  ...
  mtx_unlock(mtx);
}
```

Figure 3.4: The access at line A is loop-invariant.

```
if (...) {
  r1 = *x;
  ...
  mtx_lock(mtx);
  ...
  mtx_unlock(mtx);
  while (...) {
    ...
    mtx_lock(mtx);
    ...
    mtx_unlock(mtx);
  }
}
```

Figure 3.5: Figure 3.4 with the load hoisted out of the loop.

operations.

3.1.2 Loop Invariance

We can also use interference-free regions to determine loop-invariant references for loops that contain synchronization. The loop in Figure 3.4 is again not a simple synchronization-free region, so it is not immediately clear whether the load of `x` can be moved out of the loop. However, `x` is guaranteed to be accessed between every lock release and the next lock acquire operation. Hence the interference-free region for each access of `x` must overlap with the previous and next one, if they exist. Therefore, all loaded values of `x` must be the same, so it is safe to move the load out of the loop, taking care to guard the load so as not to introduce a data race (Figure 3.5).

Similar observations apply to loops that access C11 [22] `atomic` objects (see Section 2.1.6). If we consider the loop below where `a` has type `volatile atomic_int *`, we can deduce that the loop contains only acquire operations, and therefore the interference-free region of any access in the loop includes all later iterations of the loop.

```
do {
  r1 = *x;
  ...
} while(atomic_load(a));
```

Thus the read of `x` can safely be hoisted out of the loop.

3.2 Formalism

Memory models are delicate and must be reasoned about in a formal setting. This section formally defines the interference-free regions described in Section 3.1 and proves their correctness: if one thread writes to a location while another thread is in an interference-free region for that location, there is a data race.

3.2.1 Model

Recall from Section 2.3 that an execution E consists of a program P, a set of actions A, a program order $\leq_{\mathsf{po}}$, a synchronization order $<_{\mathsf{so}}$, a writes-seen function W, and a values-written function V: $E = (P, A, \leq_{\mathsf{po}}, <_{\mathsf{so}}, W, V)$. We ignore P, W, and V because they are irrelevant to IFRs. We also assume a *synchronizes-with* relation $<_{\mathsf{sw}}$ instead of a synchronization order $<_{\mathsf{so}}$. Note that if we assume some concrete type of synchronization (e.g., mutexes), it is trivial to infer the synchronizes-with relation from the synchronization order. Starting with the synchronizes-with order instead means we do not have to specify what types of synchronization may be used. Instead, we assume that if $u_1 <_{\mathsf{sw}} u_2$, then u_1 corresponds to a release action and u_2 corresponds to an acquire action. We assume that actions include reads and writes of shared variables as well as unspecified synchronization actions:

$$\text{Kinds } k ::= \mathsf{read}(x) \mid \mathsf{write}(x) \mid \ldots$$

Therefore an execution E is the triple $(A, \leq_{\mathsf{po}}, <_{\mathsf{sw}})$.

As in Section 2.3, the *happens-before order* is the reflexive transitive closure of the union of the program order and the synchronizes-with order.[2] More formally, $u_1 \leq_{\mathsf{hb}} u_2$ if (1) $u_1 \leq_{\mathsf{po}} u_2$, (2) $u_1 <_{\mathsf{sw}} u_2$, or (3) there exists u_3 such that $u_1 \leq_{\mathsf{hb}} u_3$ and $u_3 \leq_{\mathsf{hb}} u_2$. We assume that the execution is well-formed and therefore that $\leq_{\mathsf{hb}}$ is antisymmetric. We also assume that $\leq_{\mathsf{po}}$ totally orders all actions in a given thread; that is, for all $(t, k_1, u_1), (t, k_2, u_2) \in A$, either $u_1 \leq_{\mathsf{po}} u_2$ or $u_2 \leq_{\mathsf{po}} u_1$.

3.2.2 Incoming and Outgoing Edges

We define the notion of incoming and outgoing edges: happens-before edges that start in one thread and end in another. An action has an *outgoing edge* if it synchronizes-with an action in another thread. An action has an *incoming* edge if an action in another thread synchronizes-with that action. The intuition is that by establishing that a subsection of a single thread's execution does *not* have any incoming or outgoing edges, we can convince ourselves that no other thread could interfere with that thread's execution.

Definition 3.1 (Outgoing Edge)**.** *Let $(t_1, k_1, u_1) \in A$. u_1 has an* outgoing happens-before edge *if there exists $(t_2, k_2, u_2) \in A$ such that $u_1 <_{\mathsf{sw}} u_2$ and $t_1 \neq t_2$.*

Definition 3.2 (Incoming Edge)**.** *Let $(t_2, k_2, u_2) \in A$. u_2 has an* incoming happens-before edge *if there exists $(t_1, k_1, u_1) \in A$ such that $u_1 <_{\mathsf{sw}} u_2$ and $t_1 \neq t_2$.*

If there is a happens-before edge between two actions in two different threads, then there must be an action in the first thread with an outgoing happens-before edge.

Lemma 3.1 (Existence of Outgoing Edge)**.** *Let $(t_1, k_1, u_1), (t_2, k_2, u_2) \in A$ such that $u_1 \leq_{\mathsf{hb}} u_2$ and $t_1 \neq t_2$. Then there exists u_3 such that $u_1 \leq_{\mathsf{po}} u_3$, $u_3 \leq_{\mathsf{hb}} u_2$, and u_3 has an outgoing edge.*

Proof. The proof is a straightforward inductive case analysis on $u_1 \leq_{\mathsf{hb}} u_2$. Intuitively, as happens-before is the closure of the program order and synchronizes-with relations, clearly

[2]The actual C/C++ happens-before relation is not transitively closed, so that `memory_order_depends` can be supported [23]. The effect is to prevent $\leq_{\mathsf{po}}$ from contributing to $\leq_{\mathsf{hb}}$ in certain contexts. This does not affect our arguments.

any chain of happens-before edges that crosses threads must include an outgoing synchronizes-with edge.

Formally, we can prove this by induction on $u_1 \leq_{\mathsf{hb}} u_2$:

- Clearly $\neg(u_1 \leq_{\mathsf{po}} u_2)$, as $t_1 \neq t_2$.
- If $u_1 <_{\mathsf{sw}} u_2$, then let $u_3 = u_1$. As $t_1 \neq t_2$, u_1 has an outgoing edge. By reflexivity, $u_1 \leq_{\mathsf{po}} u_1$. By assumption, $u_1 \leq_{\mathsf{hb}} u_2$.
- Now suppose $u_1 \leq_{\mathsf{hb}} u_4$ and $u_4 \leq_{\mathsf{hb}} u_2$, where $(t_4, k_4, u_4) \in A$. Either $t_1 = t_4$ or $t_1 \neq t_4$.
  - If $t_1 = t_4$, then either $u_1 \leq_{\mathsf{po}} u_4$ or $u_4 \leq_{\mathsf{po}} u_1$ by the definition of $\leq_{\mathsf{po}}$. If $u_4 \leq_{\mathsf{po}} u_1$, then $u_4 = u_1$ because $\leq_{\mathsf{hb}}$ is antisymmetric, so $u_1 \leq_{\mathsf{po}} u_4$ as well. Use the inductive hypothesis to find u_3 such that $u_4 \leq_{\mathsf{po}} u_3$, $u_3 \leq_{\mathsf{hb}} u_2$, and u_3 has an outgoing edge. Then u_3 is the witness because $u_1 \leq_{\mathsf{po}} u_3$ by transitivity.
  - If $t_1 \neq t_4$, then use the inductive hypothesis to find u_3 such that $u_1 \leq_{\mathsf{po}} u_3$, $u_3 \leq_{\mathsf{hb}} u_4$, and u_3 has an outgoing edge. By transitivity, $u_3 \leq_{\mathsf{hb}} u_2$. □

Lemma 3.2 (Existence of Incoming Edge)**.** *Let $(t_1, k_1, u_1), (t_2, k_2, u_2) \in A$ such that $u_1 \leq_{\mathsf{hb}} u_2$ and $t_1 \neq t_2$. Then there exists u_3 such that $u_1 \leq_{\mathsf{hb}} u_3$, $u_3 \leq_{\mathsf{po}} u_2$, and u_3 has an incoming edge.*

The proof is omitted as it is entirely symmetric to the proof of Lemma 3.1.

We now establish our key result: if a region of code after a normal memory access has no outgoing happens-before edges, then any writes to the same location in other threads must happen-after that region of code. This result depends crucially on the fact that in an execution of a data-race-free program, conflicting accesses to the same variable are ordered by happens-before.

Theorem 3.1 (Forwards Interference-Free)**.** *Assume $(t_1, \mathsf{read}(x), u_1), (t_1, k_2, u_2) \in A$ such that $u_1 <_{\mathsf{po}} u_2$. Assume further that for all u_3 such that $u_1 \leq_{\mathsf{po}} u_3 <_{\mathsf{po}} u_2$, u_3 does not have an outgoing edge. Finally, assume there is some write $(t_4, \mathsf{write}(x), u_4)$ such that $u_1 \leq_{\mathsf{hb}} u_4$ and $t_1 \neq t_4$. Then $u_2 \leq_{\mathsf{hb}} u_4$.*

Proof. By Lemma 3.1, there exists u_3 such that $u_1 \leq_{\mathsf{po}} u_3$, $u_3 \leq_{\mathsf{hb}} u_4$, and u_3 has an

outgoing edge. Clearly $u_2 \leq_{\mathsf{po}} u_3$, because otherwise u_3 would violate an assumption. Then $u_2 \leq_{\mathsf{hb}} u_4$ by transitivity of happens-before. □

Symmetrically, we can show that a variable is interference-free for a region in which there are no incoming happens-before edges.

Theorem 3.2 (Backwards Interference-Free)**.** *Assume* $(t_1, k_1, u_1), (t_1, \mathsf{read}(x), u_2) \in A$ *such that* $u_1 <_{\mathsf{po}} u_2$*. Assume further that for all* u_3 *such that* $u_1 <_{\mathsf{po}} u_3 \leq_{\mathsf{po}} u_2$*,* u_3 *does not have an incoming edge. Finally, assume there is some write* $(t_4, \mathsf{write}(x), u_4)$ *such that* $u_4 \leq_{\mathsf{hb}} u_2$ *and* $t_1 \neq t_4$*. Then* $u_4 \leq_{\mathsf{hb}} u_1$*.*

Proof. By Lemma 3.2, there exists u_3 such that $u_4 \leq_{\mathsf{hb}} u_3$, $u_3 \leq_{\mathsf{po}} u_2$, and u_3 has an incoming edge. Clearly $u_3 \leq_{\mathsf{po}} u_1$, because otherwise u_3 would violate an assumption. Then $u_4 \leq_{\mathsf{hb}} u_1$ by transitivity of happens-before. □

By applying Theorems 3.1 and 3.2, we can conclude that any write must happen-before or happen-after the entire interference-free region for an access. We can also state similar theorems for the interference-free regions around writes:

Theorem 3.3 (Forwards Interference-Free (Write Case))**.** *Assume* $(t_1, \mathsf{write}(x), u_1), (t_1, k_2, u_2) \in A$ *such that* $u_1 <_{\mathsf{po}} u_2$*. Assume further that for all* u_3 *such that* $u_1 \leq_{\mathsf{po}} u_3 <_{\mathsf{po}} u_2$*,* u_3 *does not have an outgoing edge. Finally, assume there is some memory access* (t_4, k_4, u_4) *such that* $k_4 = \mathsf{read}(x)$ *or* $\mathsf{write}(x)$*,* $u_1 \leq_{\mathsf{hb}} u_4$*, and* $t_1 \neq t_4$*. Then* $u_2 \leq_{\mathsf{hb}} u_4$*.*

Theorem 3.4 (Backwards Interference-Free (Write Case))**.** *Assume* $(t_1, k_1, u_1), (t_1, \mathsf{write}(x), u_2) \in A$ *such that* $u_1 <_{\mathsf{po}} u_2$*. Assume further that for all* u_3 *such that* $u_1 <_{\mathsf{po}} u_3 \leq_{\mathsf{po}} u_2$*,* u_3 *does not have an incoming edge. Finally, assume there is some memory access* (t_4, k_4, u_4) *such that* $k = \mathsf{read}(x)$ *or* $\mathsf{write}(x)$*,* $u_4 \leq_{\mathsf{hb}} u_2$*, and* $t_1 \neq t_4$*. Then* $u_4 \leq_{\mathsf{hb}} u_1$*.*

3.3 Barriers

This section extends our analysis to handle rendezvous-style barriers (e.g., the `pthread_barrier_t` type [30]). Barriers allow threads to stop and wait for other threads to reach a certain point in their execution before continuing. Treating barriers as release and acquire actions

```
struct global_info *global;

void worker(int pid, pthread_barrier_t *barrier) {

  (*global).threads[i] = pthread_self();

  if (pid == 0) { /* master thread */
    (*global).data = ... /* prepare data to be processed */
  }

  pthread_barrier_wait(barrier);

  data mydata = (*global).data[i];
  ... /* first stage of processing */

  pthread_barrier_wait(barrier);

  ... /* more pipeline stages, separated  */
  ... /* by calls to pthread_barrier_wait */
}
```

Figure 3.6: A typical use of barriers within a program.

is too imprecise, so we handle them specially. Our key observation is that even though barriers act like a release operation followed by an acquire action, effectively cutting off interference-free regions in both directions, we can still reason about interference-freedom if there are accesses to a variable both before and after a given invocation of a barrier.

For example, Figure 3.6 shows a typical use of barriers in a multi-stage input processing program. All threads execute the `worker()` function, including the master thread. The threads update a global struct with bookeeping information, then all non-master threads sit at a call to `pthread_barrier_wait`, waiting until the master thread has initialized the input data. All threads then process the input data in parallel, calling `pthread_barrier_wait()` when the first stage is complete. In this way, threads can coordinate so that no thread gets ahead of another.

Note that in the example, the variable `global` is read both before and after the first call to `pthread_barrier_wait()`. We are interested in whether `global` is interference-free between the two reads—i.e., if another thread could write to `global` without causing a data race. Intuitively, it seems like `global` should be interference-free. It is clearly interference-free before the barrier call and after the barrier call. Any writes to `global` would therefore have to (1) happen-after the code before the barrier call and (2) happen-before the code after the barrier call. In other words, the other thread would have to modify `global` *during* its own call to `pthread_barrier_wait`, which is not possible. This intuition proves to be correct: if a variable is accessed *both before and after* a call to the barrier wait function, then we can extend the interference-free regions for both accesses through the call.

Barriers are not included in the C/C++ memory model, so this section makes an educated guess on how they would be modeled if added to the standard. Our model is similar to that of Unified Parallel C [47]. We model each barrier wait call as two actions: a "notify" action, which tells other threads that the current thread has arrived at the barrier, and a "wait" action, which waits for other threads to reach the barrier. Notifies are releases and waits are acquires. Assume that the possible types of actions include notify and wait actions on barrier identifiers b:

$$\text{Kinds } k ::= \mathsf{notify}(b) \mid \mathsf{wait}(b) \mid \ldots$$

Notify and wait actions have the following behavior:

Notify synchronizes-with wait: If $(t_1, \mathsf{notify}(b), u_1), (t_2, \mathsf{wait}(b), u_2) \in A$, then $u_1 <_{\mathsf{sw}} u_2$.

Notify and wait only synchronize-with each other: If $(t_1, k_1, u_1), (t_2, k_2, u_2) \in A$ and $u_1 <_{\mathsf{sw}} u_2$, then $k_1 = \mathsf{notify}(b)$ if and only if $k_2 = \mathsf{wait}(b)$.

Notify and wait are called in the proper order with no intervening synchronization: If a thread t calls notify and wait on a barrier b ($(t, \mathsf{notify}(b), u_1^n), (t, \mathsf{wait}(b), u_1^w) \in A$), then notify is called before wait ($u_1^n \leq_{\mathsf{po}} u_1^w$), and t does not perform any synchronization actions between the calls to notify and wait.[3]

Note that each barrier is invoked only once per thread; else it would not always be the case that a notify always synchronizes-with a wait on the same barrier, or that notify is always called before wait with no intervening synchronization. Although real programs may wait on the same barrier multiple times, this is simply a convenience; it is possible to allocate a new barrier for every invocation. Moreover, we could make our formalism more realistic by tagging notify and wait actions with a generation that indicates how many times a thread has invoked this particular barrier, but such a change would not increase the expressiveness of the model.

Our key insight is that *nothing can happen-between a notify and a wait.* First, we prove two lemmas. The first lemma establishes that, if an action in another thread happens-after a notify, then either the action also happens-after the subsequent wait, or the happens-before chain between the notify and the remote action includes a synchronizes-with edge originating from the notify.

Lemma 3.3 (Inversion of Happens-After-Notify)**.** *Suppose* $(t_1, \mathsf{notify}(b), u_1^n), (t_1, k_1, u_1), (t_1, \mathsf{wait}(b), u_1^w), (t_2, k_2, u_2) \in A$ *such that* $t_1 \neq t_2$, $u_1^n \leq_{\mathsf{po}} u_1 \leq_{\mathsf{po}} u_1^w$, *and* $u_1 \leq_{\mathsf{hb}} u_2$. *Then either (1)* $u_1^w \leq_{\mathsf{hb}} u_2$ *or (2)* $u_1^n = u_1$ *and there exists* u_3 *such that* $u_1^n <_{\mathsf{sw}} u_3 \leq_{\mathsf{hb}} u_2$.

[3]Formally, if $\exists u_2$, u_3 such that $u_1^n \leq_{\mathsf{po}} u_2 \leq_{\mathsf{po}} u_1^w$ and $u_2 <_{\mathsf{sw}} u_3$, then $u_2 = u_1^n$; if $u_1^n \leq_{\mathsf{po}} u_2 \leq_{\mathsf{po}} u_1^w$ and $u_3 <_{\mathsf{sw}} u_2$, then $u_2 = u_1^w$.

```
r1 = x;
...
pthread_mutex_lock(...);
...
pthread_mutex_lock(...);
...
pthread_barrier_wait(b);
...
pthread_mutex_unlock(...);
...
pthread_mutex_unlock(...);
...
r2 = x;
```

Figure 3.7: Interference-free region around a barrier wait call.

Proof. Proof by induction on $u_1 \leq_{\mathsf{hb}} u_2$.

- Clearly $\neg(u_1 \leq_{\mathsf{po}} u_2)$, as $t_1 \neq t_2$.
- If $u_1 <_{\mathsf{sw}} u_2$, then by the assumption that there is no synchronization between a notify and a wait, $u_1^n = u_1$. Let $u_3 = u_2$. Then $u_1^n <_{\mathsf{sw}} u_3$, and also $u_3 \leq_{\mathsf{hb}} u_2$ by reflexivity of happens-before.
- Now suppose $u_1 \leq_{\mathsf{hb}} u_4 \leq_{\mathsf{hb}} u_2$ where $(t_4, k_4, u_4) \in A$. Either $t_1 = t_4$ or $t_1 \neq t_4$.
  - Suppose $t_1 = t_4$. Then by the definition of $\leq_{\mathsf{po}}$, either $u_1 \leq_{\mathsf{po}} u_4$ or $u_4 \leq_{\mathsf{po}} u_1$. If $u_4 \leq_{\mathsf{po}} u_1$, then by antisymmetry of happens-before $u_4 = u_1$, so $u_1 \leq_{\mathsf{po}} u_4$ by reflexivity of $\leq_{\mathsf{po}}$. Also by assumption, either $u_1^w \leq_{\mathsf{po}} u_4$ or $u_4 \leq_{\mathsf{po}} u_1^w$. If $u_1^w \leq_{\mathsf{po}} u_4$, then $u_1^w \leq_{\mathsf{hb}} u_2$ by transitivity of happens-before. If $u_4 \leq_{\mathsf{po}} u_1^w$, then use the inductive hypothesis on $u_4 \leq_{\mathsf{hb}} u_2$. There are two possible cases:
    - $u_1^w \leq_{\mathsf{hb}} u_2$. Then we are done.
    - $u_1^n = u_4$ and there exists u_3 such that $u_1^n <_{\mathsf{sw}} u_3 \leq_{\mathsf{hb}} u_2$. Since $u_1^n = u_4$ and $u_1^n \leq_{\mathsf{po}} u_1 \leq_{\mathsf{po}} u_4$, $u_1^n = u_1$ also.
  - Suppose $t_1 \neq t_4$. Apply the inductive hypothesis to $u_1 \leq_{\mathsf{hb}} u_4$. There are two possible cases:
    - $u_1^w \leq_{\mathsf{hb}} u_4$. Then $u_1^w \leq_{\mathsf{hb}} u_2$ by transitivity.
    - $u_1^n = u_1$ and there exists u_3 such that $u_1^n <_{\mathsf{sw}} u_3 \leq_{\mathsf{hb}} u_4$. Again, $u_3 \leq_{\mathsf{hb}} u_2$

by transitivity. □

A symmetric lemma establishes that if a remote action happens-before a wait, then either the action also happens-before the notify, or the happens-before chain includes a synchronizes-with edge terminating at the wait.

Lemma 3.4 (Inversion of Happens-Before-Wait)**.** *Suppose* $(t_1, \mathsf{notify}(b), u_1^n), (t_1, k_1, u_1), (t_1, \mathsf{wait}(b), u_1^w), (t_2, k_2, u_2) \in A$ *such that* $t_1 \neq t_2$, $u_1^n \leq_{\mathsf{po}} u_1 \leq_{\mathsf{po}} u_1^w$, *and* $u_2 \leq_{\mathsf{hb}} u_1$. *Then either (1)* $u_2 \leq_{\mathsf{hb}} u_1^n$ *or (2)* $u_1^w = u_1$ *and there exists* u_3 *such that* $u_2 \leq_{\mathsf{hb}} u_3 <_{\mathsf{sw}} u_1^w$.

The proof is omitted as it is entirely symmetric to the proof of Lemma 3.3.

Given these two lemmas, the key theorem follows easily:

Theorem 3.5 (Nothing Happens-Between Notify and Wait)**.** *Suppose* $(t_1, \mathsf{notify}(b), u_1^n), (t_1, \mathsf{wait}(b), u_1^w), (t_2, k_2, u_2) \in A$ *such that* $t_1 \neq t_2$. *Then* $\neg(u_1^n \leq_{\mathsf{hb}} u_2 \leq_{\mathsf{hb}} u_1^w)$.

Proof. Assume $u_1^n \leq_{\mathsf{hb}} u_2 \leq_{\mathsf{hb}} u_1^w$. We will establish a contradiction by proving that $\leq_{\mathsf{hb}}$ is no longer antisymmetric. By Lemma 3.3, either $u_1^w \leq_{\mathsf{hb}} u_2$ or there exists u_3 such that $u_1^n <_{\mathsf{sw}} u_3 \leq_{\mathsf{hb}} u_2$. The first case is a violation of the antisymmetry of $\leq_{\mathsf{hb}}$, as we have that $u_2 \leq_{\mathsf{hb}} u_1^w$, and we know $u_1^w \neq u_2$ because they are from different threads. Similarly, we can rule out the first case for Lemma 3.4. Therefore (using u_4 as the witness for the second case of Lemma 3.4), we have that $u_1^n <_{\mathsf{sw}} u_3 \leq_{\mathsf{hb}} u_2 \leq_{\mathsf{hb}} u_4 <_{\mathsf{sw}} u_1^w$. By assumption, u_3 is $\mathsf{wait}(b)$ and u_4 is $\mathsf{notify}(b)$. Therefore $u_4 <_{\mathsf{sw}} u_3$, which creates a cycle in $\leq_{\mathsf{hb}}$: $u_3 \leq_{\mathsf{hb}} u_4$ and $u_4 \leq_{\mathsf{hb}} u_3$. We know that $u_3 \neq u_4$ as they are notify and wait actions, respectively, so the theorem is proved. □

We can combine Theorem 3.5 with Theorems 3.1 and 3.2 to infer larger interference-free regions. For instance, consider the code in Figure 3.7. The call to `pthread_barrier_wait` performs both the notify and wait actions, satisfying the requirement that these actions occur in the proper order with no intervening synchronization. Suppose a remote write to `x` were to happen-after the first read and happen-before the second. By Theorem 3.1, the write must happen-after the notify; by Theorem 3.2, the write must happen-before the wait. Therefore, by Theorem 3.5, no such write exists, so the two reads see the same value. In

effect, the interference-free region for `x` extends across the call to `pthread_barrier_wait`—but only because there are reads of `x` before and after the barrier. Else the remote write could happen-before or happen-after the barrier.

3.4 Other Applications

Interference-free regions are useful for understanding the behavior of functions that contain internal synchronization. For example, the C++ draft specification defines `malloc()` and `free()` as acquire and release synchronization operations, respectively [23]. We can use interference-free regions in the code below to establish that global variables `p` and `q` are not modified by other threads, and therefore that the two `free()` operations properly deallocate the memory allocated by the two `malloc` operations.

```
p = malloc(...);
q = malloc(...);
free(p);
free(q);
```

This kind of reasoning is applicable not just to compilers, but also to static analysis tools, where reasoning about properties such as deadlock freedom or memory allocation often requires knowledge about variables that might conceivably be changed by other threads.

3.5 Conclusion

We have developed interference-free regions as a new way of reasoning about data-race-free programs. Since our analysis is based heavily on data-race-freedom, it requires only information about a single thread. Our approach enables new kinds of inferences about program behavior that had not previously been considered. The next two chapters explore two interesting applications of interference-free regions.

Chapter 4

INTRAPROCEDURAL COMPILER OPTIMIZATION USING INTERFERENCE-FREE REGIONS

This chapter covers the first of two practical applications of interference-free regions: using IFRs to extend the scope of compiler optimizations. Specifically, we can remove redundant reads across synchronization calls, as long as the calls are interference-free for the variable in question. This application was the original motivation of our work on interference-free regions and is a natural consequence of IFR-based reasoning.

We have implemented a pass in the LLVM compiler framework [26] that uses interference-free regions to refine side effect information for synchronization calls. Because we do not know which path through a program a given execution will take, we must be conservative: we identify synchronization calls that, no matter which path is taken, fall into some interference-free region for a given variable. We then remove the variable from the set of variables modified by each identified synchronization call. This change allows later optimization passes to ignore the side effects of the call with respect to the interference-free variable.

4.1 IFR-Based Optimization

Consider our original motivating example for interference-free regions, from Chapter 3:

```
mtx_lock(*mtx_p);
...
mtx_unlock(*mtx_p);
```

Again, ellipses represent code that neither performs synchronization nor modifies mentioned variables. We observed that the call to `mtx_lock()` is *interference-free* for `mtx_p`, because it falls in the interference-free region for the first read of `mtx_p`. Below we have separated the reads into their own statements and bracketed the IFR for the first read:

```
A: mtx_t *tmp1 = *mtx_p;
mtx_lock(tmp1);
...
mtx_t *tmp2 = *mtx_p;
mtx_unlock(tmp2);
```

} interference-free region for A

Therefore the second read of `mtx_p` is redundant and can be removed. This chapter tackles how to implement an analysis that is capable of recognizing this redundancy and eliminating the second read.

Currently, no mainstream compiler will eliminate the second read, because the two reads are separated by a synchronization call, and compilers generally assume that external calls such as synchronization calls are *opaque*: they might modify any variable. More precisely, the compiler assumes that synchronization calls might modify any variables that are thread-shared.

As a clarification, the synchronization calls themselves do not actually modify anything.[1] Rather, memory accesses may be taking place concurrently in other threads. Since we assume data-race-freedom, we can assume that if another thread is concurrently modifying a variable that the current thread also accesses, the threads must use some form of synchronization to coordinate their memory accesses. Therefore the compiler can (for the most part) ignore memory accesses in other threads by assuming that all such accesses take place during synchronization calls.

This assumption—that synchronization calls are opaque—means that optimizations are performed only within *synchronization-free regions*. We make the simple observation that this assumption is unnecessarily conservative: it is safe to optimize within *interference-free regions*, rather than within synchronization-free regions.

As a concrete example, the LLVM compiler framework [26], which we use in this chapter to implement our analysis, uses the internal `AliasAnalysis` interface to record the side effects of calls. Figure 4.1 shows the relevant part of the `AliasAnalysis` interface: the function `getModRefInfo()`, which, given a call site and a "location" (i.e., a variable), returns a code indicating if and how that call accesses that location. This interface can be used by

[1] The bodies of the synchronization functions may modify some variables—e.g., `thrd_create`, a synchronization operation with release behavior, modifies the new thread ID—so we must take care to distinguish these (non-racy) writes from writes in other threads.

```
enum ModRefResult { NoModRef = 0,
                    Ref = 1,
                    Mod = 2,
                    ModRef = 3 };

virtual ModRefResult getModRefInfo(ImmutableCallSite CS,
                                   const Location &Loc);
```

Figure 4.1: Subset of LLVM's `AliasAnalysis` interface related to call side effects.

other analyses to discover the side effects of a call. For synchronization calls, this function always returns `ModRef`, indicating that the call may read or write the variable. Our approach will be to reimplement `getModRefInfo()` so that it is more precise for synchronization calls.

4.2 Algorithm

We identify synchronization calls whose side effect information can be refined by exploiting two symmetric insights:

1. If, on a path through a function that passes through an acquire call C, there is an access A to a variable x such that A precedes C and there are no release calls between A and C, then C is in the interference-free region for A for that path. Therefore, if such an access A exists for every path through C, C does not modify x.
2. If, on a path through a function that passes through a release call C, there is an access A to a variable x such that A follows C and there are no acquire calls between C and A, then C is in the interference-free region for A for that path. Therefore, if such an access A exists for every path through C, C does not modify x.

Our analysis determines two pieces of information. First, for each acquire call, we need the set of variables that must have been *accessed since the last release call* (ASLR). Second, for each release call, we need the set of variables that must be *accessed before the next acquire call* (ABNA). The former is a simple forward dataflow analysis; the latter is a simple backward dataflow analysis.

Statement type	Statement form	ASLR$[p']$
Load	p : `r` = $*$`x`;	ASLR$[p] \cup \{x\}$
Store	p : $*$`x` = `r`;	ASLR$[p] \cup \{x\}$
Acquire	p : `mtx_lock`(`m`);	ASLR$[p]$
Release	p : `mtx_unlock`(`m`);	$\{\}$
Call	p : `f`(...);	$\{\}$
Other	p : ...;	ASLR$[p]$

Figure 4.2: Summary of the forwards ASLR analysis used to identify variables that are interference-free at acquire calls. p' is the program point after the statement at point p.

4.2.1 Forwards ASLR Analysis

Figure 4.2 gives the forwards dataflow analysis that infers the ASLR sets for acquire calls. The set is initialized to be empty at the start of each function. The set propagates through acquire calls and is killed at other calls. (See Section 6.4 for an improved version of the analysis in which the set may propagate through other calls.) At memory accesses, we add the accessed variable to the ASLR set.

The ASLR and ABNA analyses assume that the code is in *single static assignment form* (SSA) [4]—i.e., that any named variables are constant. Else we could infer that a location is interference-free when it is not; e.g., in the code below, we would infer that the location pointed-to by `x` is interference-free at the acquire call, when in fact the first load accessed a different location:

```
int tmp = *x;
x = &y;
mtx_lock(&mtx);
```

Using SSA form means we can conflate the terms "variable" and "location," as every named variable refers to a single location at runtime. LLVM bytecode is in SSA form, so this assumption holds.

Figure 4.3 lists the code for an example program on which we will run the ASLR and ABNA analyses. This program includes a for loop that loads variable `x` on every iteration

```
for (int i = 0; i < *x; i++) {
  mtx_lock(m);
  ...
  mtx_unlock(m);
}
```

Figure 4.3: Code for the example in Figures 4.4 and 4.6.

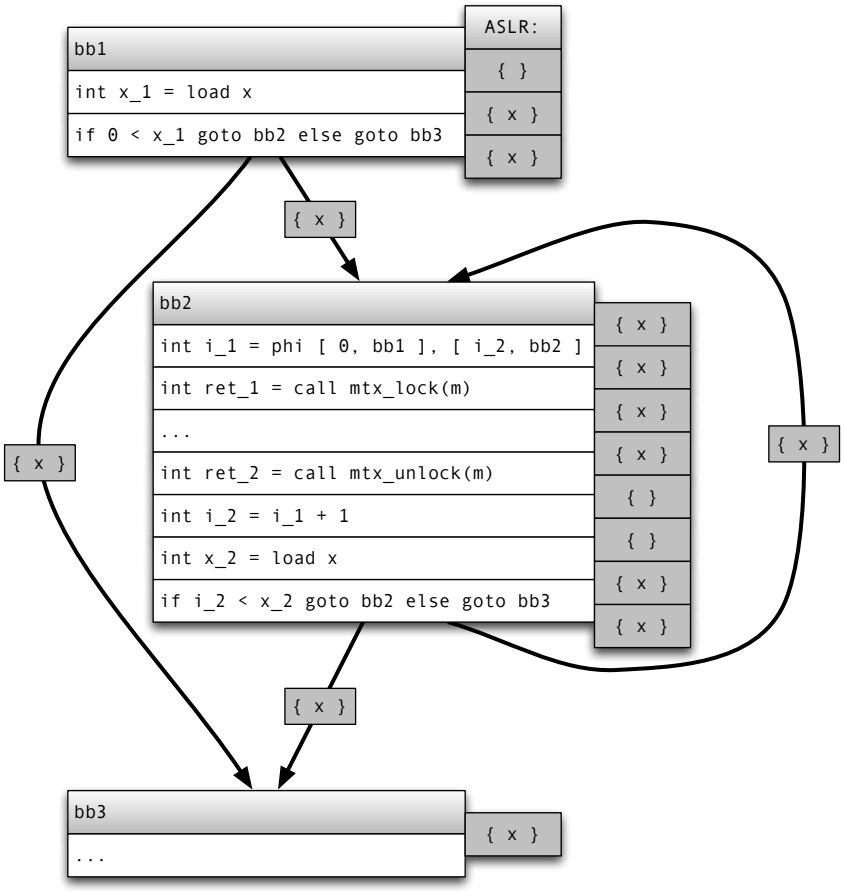

Figure 4.4: Running the ASLR analysis on a program with a loop.

Statement type	Statement form	ABNA$[p]$
Load	p : `r` = ∗`x`;	ABNA$[p'] \cup \{x\}$
Store	p : ∗`x` = `r`;	ABNA$[p'] \cup \{x\}$
Acquire	p : `mtx_lock(m);`	$\{\}$
Release	p : `mtx_unlock(m);`	ABNA$[p']$
Call	p : `f(...);`	$\{\}$
Other	p : . . . ;	ABNA$[p']$

Figure 4.5: Summary of the backwards ABNA analysis used to identify variables that are interference-free at release calls. p' is the program point after the statement at point p.

for the termination check. We should be able to hoist this load out of the loop, even though the loop contains synchronization.

Figure 4.4 shows the control-flow graph for the program and the final ASLR sets once the algorithm is run to a fixpoint. The key inference is that the ASLR set at the call to `mtx_lock()` is {`x`}, meaning that `x` is interference-free at the call. This is true only because there is a load of `x` on *all* paths to the call: one load in `bb1`, and another at the end of `bb2`. Therefore the call will always be in an interference-free region for `x` when it executes.

4.2.2 Backwards ABNA Analysis

Figure 4.5 gives the backwards dataflow analysis that infers the ABNA sets for release calls. The set is initialized to be empty at the end of each function. The ABNA sets propagate up through release calls and are killed at all other calls. At memory accesses, we add the variable being accessed to the ABNA set.

Figure 4.6 shows the results of the ABNA analysis after it reaches a fixpoint on the program in figure 4.3. Because ABNA is a backwards analysis, we have reversed the control-flow edges to indicate the direction of data flow. The key inference is that the call to `mtx_unlock()` has ABNA set {`x`}. This inference is trivial, since `x` is accessed immediately after the release call.

Combining Figures 4.4 and 4.6, we can see that `x` is interference-free for the entire loop.

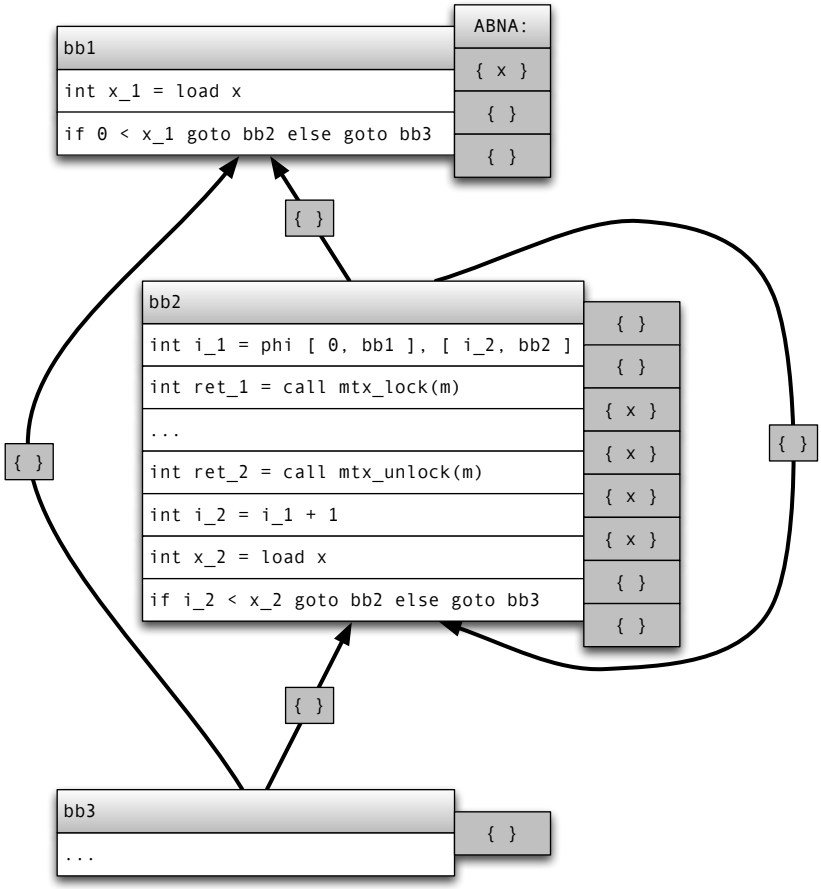

Figure 4.6: Running the ABNA analysis on a program with a loop. Since this is a backwards analysis, the control flow edges have been reversed to indicate the direction of data flow.

Algorithm 4.1 getModRefInfo(ImmutableCallSite CS, const Location &Loc)

```
if CS is an acquire call AND ASLR(CS) contains Loc then
  return Ref
end if
if CS is a release call AND ABNA(CS) contains Loc then
  return Ref
end if
if CS is a barrier call AND ASLR(CS) contains Loc AND ABNA(CS) contains Loc then
  return Ref
end if
... /* Fall back on existing alias analysis */
```

Removing `x` from the set of variables modified by both synchronization calls will allow a later optimization to hoist `x` out of the loop.

4.2.3 LLVM Implementation

The analysis extends the LLVM `AliasAnalysis` interface, specifically by overriding the method `getModRefInfo()`. A high-level outline of the implementation appears in Algorithm 4.1. LLVM allows alias analyses to compose, so we chain our analysis into the existing alias analysis pipeline. For acquire calls, any variables in the ASLR set are listed as not being modified; for release calls, any variables in the ABNA set are listed as not being modified. A call to `pthread_barrier_wait` is interference-free for a given variable if that variable is in both the ASLR and ABNA sets for the call.[2]

As an example, in Figure 3.2, `mtx_p` is found not to be modified for the first three synchronization calls, although not the last because it is a release action and its ABNA set is empty. A redundant load analysis will therefore find that the second load can be eliminated. Figure 3.4 is an example of conservatism in our analysis: `x` is not in the ABNA set for the `mtx_unlock` call (because there is a path that does not access `x` after the unlock), so access A will not be hoisted out of the loop.

In order to improve the accuracy of the analysis, we distinguish between read and write accesses in the implementation. For example, if a variable `x` must be *modified* (not just

[2]To be more precise, the variable must be in the ASLR set for the program point just before the call, and in the ABNA set for the program point just after the call.

accessed) before an acquire call, then we may assume that the call neither reads nor writes `x`. We will discuss a similar analysis in Chapter 5, where distinguishing between reads and writes is crucial for detecting data races.

4.2.4 Imprecision of the ASLR/ABNA Approach

As mentioned above, Figure 3.4 is an example of our analysis being overly conservative: we cannot hoist the load of `x` out of the loop, because `x` is not interference-free at the call to `mtx_unlock()`. Specifically, on the last iteration of the loop, the release call is not interference-free for `x` because `x` is not accessed after the call.

Another example of imprecision is the example below:

```
A: int tmp = *x;
mtx_lock(&m);
...
mtx_unlock(&m);
if (b) {
  B: int tmp2 = *x;
  ...
}
```

Here it is clear that the second load of `x` is redundant, since in executions where the if statement is taken, the interference-free regions of the two loads overlap:

```
A: int tmp1 = *x;      ⎫
mtx_lock(&m);          ⎭ interference-free region for A
...                    ⎫
mtx_unlock(&m);        ⎬ interference-free region for C
B: int tmp2 = *x;      ⎭
...
```

However, the ABNA analysis infers that the ABNA set for the call to `mtx_unlock` is {}, because in executions where the if statement is not taken, there is no load of `x` after the call:

```
A: int tmp1 = *x;      ⎫
mtx_lock(&m);          ⎭ interference-free region for A
...
mtx_unlock(&m);
```

In Chapter 6, we discuss an alternate approach that handles these cases. Specifically, instead of refining the side effect information for synchronization calls, we analyze the paths between memory accesses. A path is interference-free for the variable being accessed if it does not have a release call and an acquire call, in that order. In the example above, the path from `A` to `B` has an acquire call followed by a release call, so it is interference-free for `x`.

4.3 Data-Race-Freedom

Since we assume data-race-freedom, programs with data races may be transformed in hard-to-predict ways, complicating debugging. This is already true for current compilers; otherwise current sequential techniques could not even be applied within synchronization-free regions [11]. We are not qualitatively changing the situation, and it does not appear to be a major problem in practice. Even if a compiler does not use our analysis, programmers should debug using a data-race detector (see Chapter 5) in order to avoid unpredicatable behavior due to incorrect optimizations and to catch data races immediately rather than after data structure corruption.

We expect this analysis might be useful on an opt-in basis. Programmers could, for example, specify on the command line that "aggressive DRF optimizations" should be enabled. We envision an "-ODRF" flag in future C/C++ compilers, the documentation for which would make explicit that the compiler would not restrict its optimizations to synchronization-free regions.

4.4 Results

We inserted our analysis into LLVM 2.8's link-time optimization pipeline just before the loop-invariant code motion (LICM) and global value numbering (GVN) transformations. The machine used for compilation and running the tests was a 4-core 2.8GHz Intel Xeon with 16GB of RAM, running Linux.

Microbenchmark Figure 4.7 shows a small C program that contains a redundant load: every iteration of the loop in function `f()` checks the value of `max`, which is constant. But because the loop contains synchronization, the standard LLVM alias analysis pass assumes

```
int max, shared_counter;
pthread_mutex_t m;

void *f(void *my_num) {
  int n = *((int *) my_num);
  while (n <= max) {
    pthread_mutex_lock(&m);
    shared_counter++;
    pthread_mutex_unlock(&m);
    n += 2;
  }
}

int main() {
  pthread_t t1, t2;
  int n1 = 1;
  int n2 = 2;
  max = 10000000;
  shared_counter = 0;
  pthread_mutex_init(&m, NULL);
  pthread_create(&t1, NULL, f, (void *) &n1);
  pthread_create(&t2, NULL, f, (void *) &n2);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
}
```

Figure 4.7: A microbenchmark demonstrating the effectiveness of IFR-based optimizations. The microbenchmark uses the `pthreads` API because C11 threads had not yet been implemented at the time of writing. When combined with our analysis, GVN moves the load of `max` out of the loop in `f()`.

SPLASH-2 Benchmark	LOC	Syncs in IFRs	Loads deleted	Speedup
lu-n	678	14	16	1.0080
radix	833	27	22	0.9962
fft	899	12	13	1.0081
lu-c	911	17	19	0.9988
water-n	2,063	27	24	0.9941
water-s	2,670	27	30	0.9701
barnes	2,864	20	11	0.9977
ocean-n	3,046	34	53	0.9840
volrend	4,204	31	25	0.9412
fmm	4,325	37	36	1.0050
ocean-c	4,774	95	79	0.9915
cholesky	5,139	83	19	0.9869
raytrace	10,649	19	14	1.0260
radiosity	11,760	93	45	1.0000

Table 4.1: SPLASH-2 results.

the synchronization calls may modify `max`. Our analysis removes `max` from the modified sets for the synchronization calls, allowing GVN to hoist the load out of the loop. Running the program under valgrind shows that the optimization is indeed effective: the optimized program performs 10 million fewer loads. However, the program performs over 400 million memory operations in total, so there is no noticable performance improvement.

Realistic applications We compiled the SPLASH-2 benchmarks [49] using our analysis. The analysis did not affect the LICM pass (we suspect because LICM handles only function calls that do not modify any memory locations), but the GVN pass found numerous opportunities to remove redundant loads (Table 4.1).[3] The third column in Table 4.1 lists the number of synchronization calls which were found to be in the interference-free region of at least one variable. The fourth column lists the difference in the number of loads deleted by GVN when run with and without our analysis. The fifth column gives the speedup for each

[3]Theoretically, our analysis should also be useful for dead store elimination, but we did not observe any improvement in LLVM's dead store elimination pass, so we concentrate on loads here.

benchmark, which we compute as the runtime for the version of the code compiled without our analysis divided by the runtime for the version compiled with out analysis.

Although the analysis exposed a number of redundant loads, we have had little success in terms of actually extracting performance from these optimizations. The benchmarks either have similar performance on both versions of the code, or our "optimized" version is slightly worse. One problem is that the loads may not be located on hot paths. Another possibility is that the optimizations increased the live ranges of variables, resulting in more loads as register variables are spilled to the stack (perhaps due to the low number of callee-saves registers on the x86 architecture).

4.5 Related Work

Our analysis is related to a compiler transformation known as *roach-motel reordering*. This transformation increases the size of critical sections by moving actions either past a lock acquire or before a lock release. In some cases, it is possible to use this line of reasoning to infer interference-free regions by repeatedly swapping an access until it reaches the end of a region. Sevcik established that this transformation is legal for data-race-free programs [44]. He also observed that it is legal to eliminate reads across synchronization calls as long as there is no intervening release/acquire pair [45]. We believe our characterization is independently useful, particularly since we make very minimal assumptions about the language and synchronization primitives and we avoid complex reasoning about syntactic code transformations. In particular, we know of no prior presentation of a similar compiler analysis, nor any discussion of the consequences of overlapping regions.

Work by several of the authors of the MSPC paper [14] describes a framework for enabling sequential optimizations in multithreaded programs [24]. They identify paths on which variables are "siloed" by iteratively refining a graph of the program. (This "siloed" property is essentially the same as our notion of interference-freedom.) Like us, their implementation refines the modified/referenced sets for synchronization calls. Our work is complementary to this work, as our analysis could likely be incorporated neatly into their framework as an "interference-type refinement."

Chapter 5

DYNAMIC DATA-RACE DETECTION USING INTERFERENCE-FREE REGIONS

Programmers must write programs carefully to ensure data races are prohibited in all executions. Unfortunately, code with potential data races is easily overlooked and even thorough testing may fail to uncover data races that only rarely affect program behavior. The difficulty of dealing with data races necessitates tools to detect them. Over the last fifteen years, many data-race detectors have been developed, exploring the design space along familiar axes of static vs. dynamic detection and performance vs. precision.

This chapter describes *IFRit*, a new dynamic data-race detector based on the fundamental notion of interference-free regions.[1] Our work is the first to use IFRs for data-race detection. We have implemented our technique in the LLVM compiler framework [26] and used it to detect data races in mature real-world software. The implementation requires a novel static analysis for soundly identifying IFRs and a dynamic analysis for finding races using the static instrumentation. We directly compare our system with two state-of-the-art systems and show our performance is considerably better—orders of magnitude better in several cases. We show that the races our system detects include nearly all the races reported by the precise detectors. We show that by combining our approach with sampling we can reduce our overheads enough that our technique could be used in deployed systems or integrated into a build environment. We have developed a formal model of IFRs for data-race detection and used it prove correctness: Any data race reported by our approach is a true data race.

---

[1] Ifrits are spirit-beings from Arabian mythology that, like data races, are known for being mischievous and elusive.

5.1 Overview

A typical dynamic data-race detector observes an execution and reports if data races occur on (just) that execution. Ideally such a detector would run fast, report all data races, and not give any false reports, i.e., reported data races that did not occur. In practice, fully precise data-race detectors run programs orders of magnitude slower [39, 18] than uninstrumented execution, so it is typically useful to sacrifice precision in principled ways while still detecting many data races.

IFRit is an imprecise data-race detector: it never reports false data races but may miss data races. Prior work with this strategy has relied on sampling: removing instrumentation from some memory accesses. Our work can also leverage sampling, but more fundamentally, it separates the instrumentation from the memory access and can coalesce the instrumentation for many accesses to the same variable. For example, consider:

```
mtx_lock(m);
for(int i = 0; i < 1000; i++) {
  ...
  *x = ... ;
  ...
}
mtx_unlock(m);
```

Assuming the loop contains no synchronization, there is no reason to instrument each access to `x`. Instead, to detect data races on `x`, it suffices to instrument the region between the `mtx_lock` and `mtx_unlock` as "writing to `x`," which dynamically requires instrumentation only before and after the loop.

Moreover, we go beyond simple synchronization-free regions by incorporating interference-free regions, as explained in more detail in Section 5.1. Consider this example, where again we assume any code not shown is known to be synchronization-free:

```
*x = ...;
while(...) {
  *x = ...;
  mtx_lock(m);
  ...
  mtx_unlock(m);
```

```
}
*x = ...;
```

Here again it suffices to instrument that `x` is written to somewhere in this code region, which can be done once before and once after the loop. Instrumenting the code thusly cannot lead to reporting data races that are not true data races, even though the code has synchronization: any concurrent access to `x` would have to race with one of the accesses to `x` in the code above.

To place instrumentation in sound places while improving performance, we use static analysis. For a given variable, we can conservatively identify interference-free regions, henceforth IFRs, which for the purposes of data-race detection are regions in which any concurrent access to the variable is indeed a data race. For the second example above, the key insight is that the IFRs induced by the accesses to `x` overlap such that every code point falls in at least one IFR for `x`.

5.1.1 IFRs for Data-Race Detection

Chapter 4 used IFRs for compiler optimization. This chapter takes the fundamental idea of interference-free regions and applies it to another purpose: dynamic data-race detection.

We distinguish two types of IFRs: regions surrounding a read of a shared variable, and regions surrounding a write. We will call these *read IFRs* and *write IFRs*, respectively.

We say that two IFRs *overlap* if parts of their executions happen simultaneously: that is, the first IFR to start must end before the second IFR begins.[2] The novel insight of this work, then, is this: *If the IFRs for two accesses to the same variable in different threads overlap, and at least one is a write IFR, then the accesses form a data race.*

For example, consider the execution shown in Figure 5.1. Two threads access variable `x`, and one of the accesses is a write. We have highlighted the corresponding (overlapping) IFRs for these accesses: a write IFR for the write to `x` in Thread 1, and a read IFR for the read of `x` in Thread 2. The figure shows a number of possible happens-before edges in the execution. Note that there is no way to trace a path between the two accesses to `x` using

[2]Our implementation inserts instrumentation that serializes the beginnings and ends of IFRs for the same variable. See Section 5.3.3 for more details.

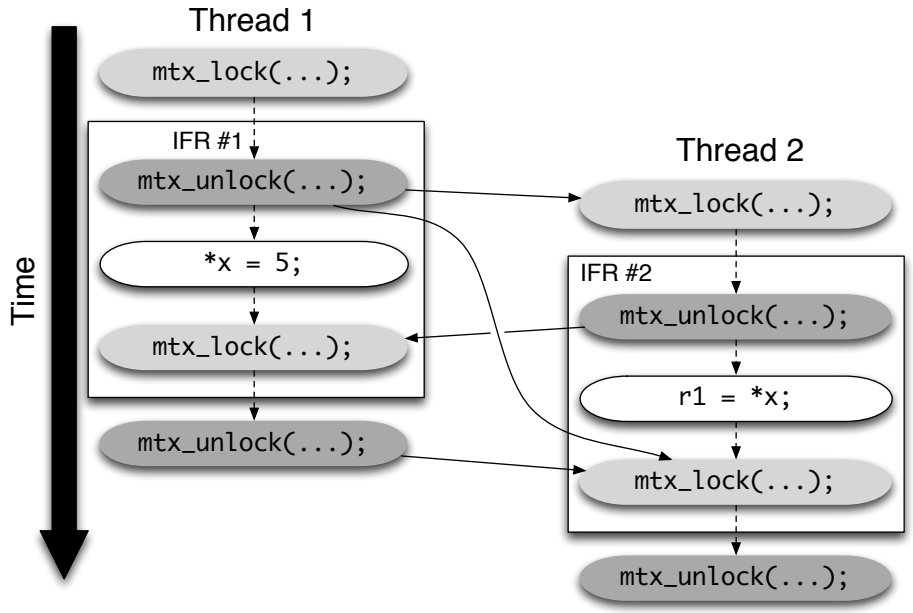

Figure 5.1: Overlapping IFRs for racy accesses in an execution. The solid blocks indicate interference-free regions. Dashed lines indicate program order; solid lines indicate possible happens-before edges between synchronization actions. The two accesses must form a data race.

the happens-before edges. Therefore, the two accesses are not ordered by happens-before, and form a data race.

We propose a dynamic data-race detection scheme based upon this insight about overlapping IFRs:

1. First, a compiler analysis identifies code points that fall within IFRs for accesses.
2. Based on this analysis, we statically insert calls to our run-time to start and stop dynamic monitors for different memory locations.
3. During execution, we report overlapping monitored regions for conflicting, concurrent accesses.

If two IFRs for the same variable do not overlap, the two accesses may or may not form a data race. Figure 5.2 shows a case in which the accesses are ordered by synchronization on a mutex `m1`. However, it is possible that the IFRs for two racy accesses will not overlap; in Figure 5.3, the two threads use different mutexes to protect variable `x`, but the critical

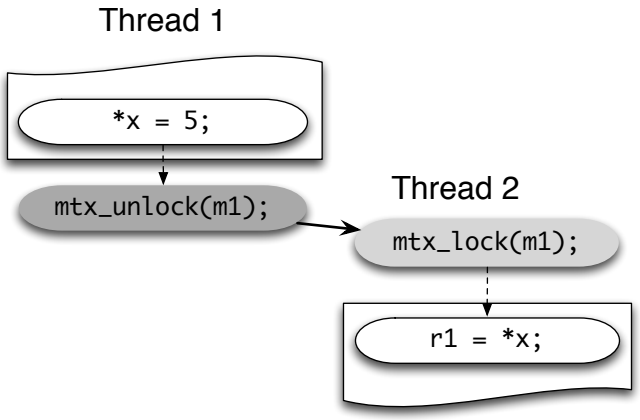

Figure 5.2: These two accesses do not form a data race, so their IFRs do not overlap.

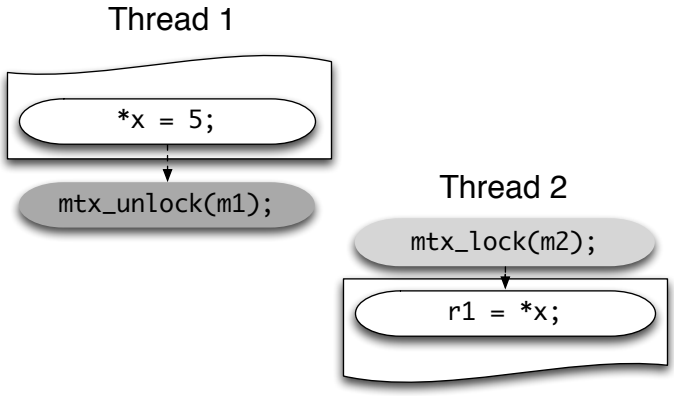

Figure 5.3: These two accesses form a data race, even though their IFRs do not overlap.

sections do not happen to overlap, so we will not catch the race. Such cases represent false negatives in our detector. Our detector is therefore sound and incomplete: no false positives, but some false negatives.

Although our algorithm has false negatives, it does have the nice quality that we might informally call "pseudo-completeness": if we run a program with our detector on enough different executions, we will *eventually* catch any data race in the program. This property follows from a standard theorem about happens-before memory models (e.g., Theorem 8.2 in [11]): if an execution of a program has a data race (i.e., two conflicting accesses unordered by happens-before), then there exists a *sequentially consistent* execution in which the two accesses execute consecutively.[3] If the racing accesses occur consecutively, their IFRs will overlap, so there exists an execution of the program for which our algorithm would catch the race. This distinguishes us from heuristic-based algorithms [42], which only look for certain classes of data races (e.g., races caused by inconsistent locking).

Because there are typically many variable accesses during an execution, and therefore many interference-free regions, we use several techniques to reduce the overhead of our dynamic detector. First, our static analysis merges the IFRs for accesses to the same variable whose IFRs overlap, allowing us to insert a single instrumentation call for many actual IFRs in the execution. Second, if IFRs for two or more variables start and stop at the same point, we handle all of the variables with a single instrumentation call. Third, we can sample IFRs to reduce the burden on the run-time. Since any two overlapping IFRs for conflicting, concurrent accesses represent a data race, sampling does not compromise our soundness guarantee. The detector is usable without sampling, but even limited use of sampling (say, monitoring 50% of the time) yields appealingly low performance overheads (see Section 5.5).

5.1.2 Synchronization-Free Regions

A possible variant on this data-race detection scheme would be to use the same kind of instrumentation, but monitor for overlapping synchronization-free regions instead of over-

[3]Recall that an execution is sequentially consistent if all threads see the same global order of actions, and the actions of each thread are in program order.

lapping interference-free regions. We believe that a scheme using interference-free regions is superior for two reasons. First, as shown in Figure 3.1, the interference-free region for an access always subsumes the synchronization-free region for an access, so monitoring interference-free regions can find more bugs. Second, the larger size of interference-free regions directly implies a smaller number of instrumentation calls (for example, the detector does not need to stop monitoring variables at acquire calls), so the performance overhead of a synchronization-free region detector would likely be higher.

Prior work on *conflict exceptions* by several of the authors of the OOPSLA paper [15] uses synchronization-free regions to implement a lightweight hardware concurrency exception model [33]. The model ignores data races in non-overlapping SFRs, much as IFRit's algorithm ignores data races in non-overlapping IFRs. The paper proves formally that exception-free executions (i.e., executions with no data races in overlapping SFRs) are guaranteed to have sequentially consistent behavior. Moreover, SFRs execute atomically in the absence of exceptions. If IFRit's static inference of IFRs were ideal, IFRit too would guarantee sequential consistency for exception-free executions and atomicity of SFRs, since IFRs are always strictly larger than SFRs and therefore IFRit would report strictly more races than the conflict-exceptions work. Of course, IFRit's static analysis is necessarily conservative, so the SFR surrounding an access may not be fully covered by the dynamic monitors.[4] Therefore IFRit has weaker guarantees than the conflict-exceptions work. IFRit's advantage is that it does not need to instrument every memory access and it does not require specialized hardware.

5.2 Static Analysis

This section presents our static analysis to insert instrumentation for the run-time. We start by explaining the types of instrumentation calls implemented by our analysis (Section 5.2.1) and giving a simple correctness criterion for the analysis (Section 5.2.2). Then we present the algorithm in two steps. First, we describe a simplified algorithm for inserting instrumentation calls (Section 5.2.3). Second, we present the refined algorithm actually used

[4]For an example of why this might happen, see Section 5.2.6.

in our implementation (Section 5.2.4). The simplified version is a useful starting point, and the refined algorithm derives directly from the ideas discussed in Section 5.2.3. Section 5.2.5 details the dataflow analysis we use to implement the algorithm, and Section 5.2.6 discusses a limitation of our prototype implementation.

Throughout this section we will refer to "variables"; variables here refer to any memory location at run-time, including array elements, global variables, and so on. In our implementation, variables are SSA names within a compiler pass, so we are guaranteed that the variable always points to the same memory location at run-time. Because LLVM automatically converts non-address-taken local variables to registers, our analysis does not process local variables if their address is not taken.

5.2.1 Instrumentation

The static analysis inserts calls to the run-time to start and stop *monitors* for different variables.[5] A "strong" monitor for variable `x` indicates that the thread is currently in an IFR for a write to `x`. A "weak" monitor for variable `x` indicates that the thread is in an IFR for either a read or write to `x`. If monitors for the same variable are active in two different threads at the same time, and at least one of the monitors is strong, then there must be overlapping IFRs for conflicting, concurrent accesses, so the run-time reports a data race.

Initially, we will consider a simple instrumentation scheme in which we start and stop monitors for a single variable via three instrumentation calls:

```
start_strong_monitor(void *x);
start_weak_monitor(void *x);
stop_monitor(void *x);
```

For example, a simple critical section could be instrumented as follows:

```
mtx_lock(m);
start_strong_monitor(x);
...
*x = ...;
...
```

[5]Our use of the term "monitor" has nothing to do with the synchronization construct of the same name.

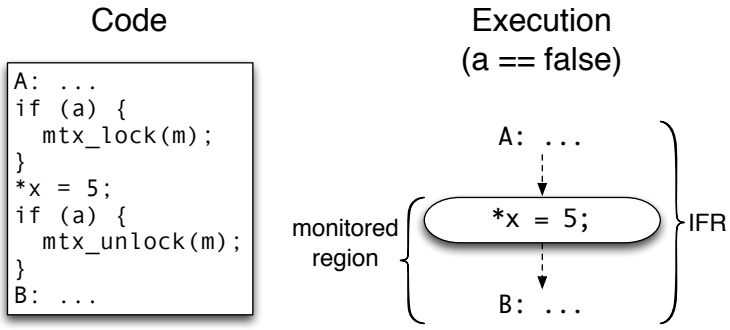

Figure 5.4: Monitored regions may be smaller than the actual IFR, due to conservatism in the static analysis.

```
stop_monitor(x);
mtx_unlock(m);
```

A downside of using static instrumentation is that the monitored region may not cover the entire dynamic IFR of an access; for instance, in Figure 5.4, our analysis does not insert the instrumentation to start the monitor until after the `if` statement.[6] On the plus side, since monitors are tied to variables, not accesses, we can use a single monitor to cover the IFRs for many accesses to the same variable. In Figure 5.5, the program may read `x` one or more times, but we only need to call `start_weak_monitor` once.

5.2.2 Correctness

Ideally, we would insert calls to `start_strong_monitor`, `start_weak_monitor` and `stop_monitor` such that a monitor would be active if and only if the corresponding point in the program's execution fell in an IFR for an access to the monitor's variable. In practice, we cannot statically determine the boundaries of every IFR, so we monitor a subset of the possible operations that fall into one or more IFRs in the execution. Crucially, we must not start a monitor unless an IFR for an access to the monitor's variable is active, and we must stop the monitor if no such IFRs are active. Formally, we meet the following two correctness conditions:

[6]Placing a call to `start_strong_monitor` before the `if` statement would violate the first correctness criterion in Section 5.2.2.

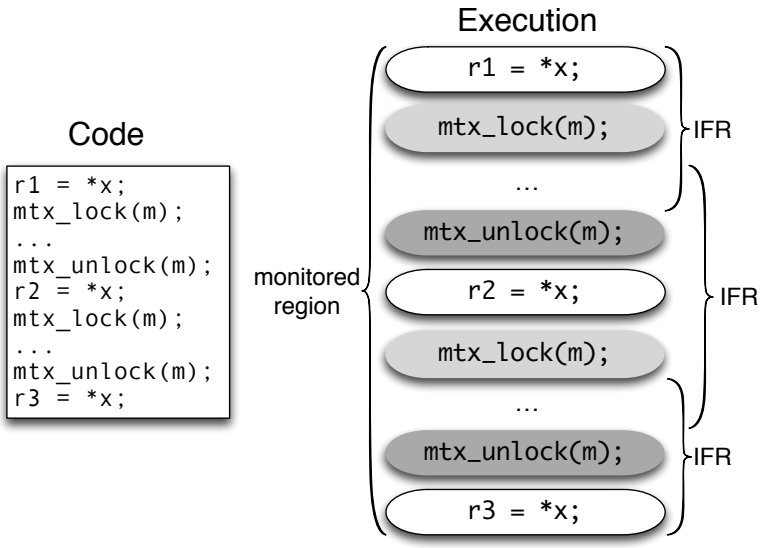

Figure 5.5: Monitored regions may combine IFRs for several accesses to the same variable.

1. Consider any execution trace from a call to `start_strong_monitor(x)` to `stop_monitor(x)`, with no intervening calls to `stop_monitor(x)`. Each operation in the trace must fall within an IFR for a write of `x`.
2. Consider any execution trace from a call to `start_weak_monitor(x)` to `stop_monitor(x)`, with no intervening calls to `stop_monitor(x)`. Each operation in the trace must fall within an IFR for a read or write of `x`.

5.2.3 Simplified Algorithm

This section presents a simple intraprocedural algorithm for inserting instrumentation calls.

First, for each program point p, we find two sets of variables:

1. WBNA[p]: the set of variables that must be written on any path through the current function's control-flow graph from p to the next acquire call (or the end of the function).
2. ABNA[p]: the set of variables that must be read or written on any path through the current function's control-flow graph from p to the next acquire call (or the end of the function).

This is the same information we gathered in Chapter 4 to find the interference-free variables at release calls, except in this case we have explicitly split the information into "write-only" and "read or write" sets. At each program point p, $\mathsf{WBNA}[p]$ represents the set of variables for which it is sound to start a strong monitor using `start_strong_monitor(x)`. The reason: If the variable will be written before the next acquire on every path from p, then all executions of p must be in an IFR for a write to the variable. Similarly, $\mathsf{ABNA}[p]$ represents the set of variables for which it is sound to start a weak monitor using `start_weak_monitor`.

Although it is sound to insert `start_*_monitor` calls at any program point, we try to minimize the number of calls by adding instrumentation in only three places: (1) after acquire calls; (2) after unknown function calls; and (3) at the beginning of basic blocks. As long as we insert all possible `start_*_monitor` calls at these three types of program points, inserting calls anywhere else in the program is redundant: every other program point is dominated either by an earlier call in the same basic block, or by the beginning of the basic block.

When inserting calls to `stop_monitor`, the problem changes from a must-analysis—which variables *must* be accessed after this program point—to a may-analysis: which monitors may have started before this program point? For each program point p, we need $\mathsf{ASLR}[p]$, the set of variables for which there exists a path from an access to the variable to p, with no intervening release calls. Again, this is the exact same information we needed in Chapter 4. IFRs always end at release calls, so we insert calls to `stop_monitor` just before release calls, as well as before unknown function calls (since the call may perform a release) and at the end of each function (to avoid interprocedural reasoning). At each of these locations, if we insert a call to `stop_monitor` for every variable in $\mathsf{ASLR}[p]$, we will have satisfied the correctness conditions in Section 5.2.2.

However, inserting calls for every variable in $\mathsf{ASLR}[p]$ is too conservative. We must take care not to stop monitors too early. For example, we should stop the monitor for `x` in Figure 5.6 at the end of the IFR for the second access to `x`, not the first. Therefore, at release calls, we insert `stop_monitor` calls only for variables in $\mathsf{ASLR}[p] - \mathsf{ABNA}[p]$ (i.e., variables for which this program point does not fall in an IFR). For variables in $\mathsf{ASLR}[p] \cap \mathsf{WBNA}[p]$ (i.e., variables for which this program point falls in a write IFR), we do not need to insert any

```
mtx_lock(m1);
start_strong_monitor(x);
...
*x = ...;
...
stop_monitor(x); // too early
mtx_unlock(m1);
...
*x = ...;
...
mtx_lock(m2);
...
stop_monitor(x);
mtx_unlock(m2);
```

Figure 5.6: Stopping a monitor too early.

```
mtx_lock(m1);
start_strong_monitor(x);
...
*x = ...;
...
downgrade_monitor(x);
mtx_unlock(m1);
...
r1 = *x;
...
mtx_lock(m2);
...
stop_monitor(x);
mtx_unlock(m2);
```

Figure 5.7: Downgrading a monitor from strong to weak.

instrumentation. For variables in $\mathsf{ASLR}[p] \cap (\mathsf{ABNA}[p] - \mathsf{WBNA}[p])$ (i.e., variables for which this program point falls in an IFR, but not necessarily a write IFR), instead of stopping the monitor, we "downgrade" it from strong to weak. This requires a fourth instrumentation function:

```
downgrade_monitor(void *x);
```

For example, in Figure 5.7 the strong monitor induced by the write to `x` is downgraded at the end of the critical section for `m1`.

In summary, we can insert instrumentation calls as follows to meet the correctness criteria of Section 5.2.2:

1. At acquire calls, unknown function calls, and the beginning of each basic block, we insert a call to `start_strong_monitor` for each variable `x` in $\mathsf{WBNA}[p]$.
2. At acquire calls, unknown function calls, and the beginning of each basic block, we insert a call to `start_weak_monitor` for each variable `x` in $\mathsf{ABNA}[p] - \mathsf{WBNA}[p]$.
3. At release calls, we insert a call to `stop_monitor` for variables in $\mathsf{ASLR}[p] - \mathsf{ABNA}[p]$.
4. At release calls, we insert a call to `downgrade_monitor` for variables in $\mathsf{ASLR}[p] \cap (\mathsf{ABNA}[p] - \mathsf{WBNA}[p])$.
5. At unknown function calls and the end of the function, we insert calls to `stop_monitor` for all variables in $\mathsf{ASLR}[p]$.

5.2.4 Refined Algorithm

In our actual implementation, instead of starting each monitor separately, we merge the `start_strong_monitor` and `start_weak_monitor` calls for different variables into a single call with a varargs argument:

```
start_monitors(int num_weak, int num_strong, ...);
```

The first `num_weak` arguments to the call after the two integers are the weak monitors to start (i.e., $\mathsf{ABNA}[p] - \mathsf{WBNA}[p]$), and the next `num_strong` arguments are the strong monitors to start ($\mathsf{WBNA}[p]$). Other than this change, which is useful because it allows the run-time to start several monitors at once, the algorithm for *starting* monitors is basically

as presented in Section 5.2.3. One difference is that we have a second helper analysis to identify redundant `start_monitors` calls; many calls are not necessary because they are dominated by a previous call to `start_monitors`.

In contrast, our approach to *stopping* monitors differs significantly from Section 5.2.3. Instead of stopping each individual monitor separately, we default to stopping all active monitors, except for a set of monitors which are permitted to continue through the call.

```
stop_all_monitors_except(int num_weak, int num_strong, ...);
```

The `num_weak` arguments correspond to calls to `downgrade_monitor`, since only weak monitors for these variables are permitted to continue through the call. As with `start_monitors`, the sets of variables for which we do not stop strong and weak monitors are $\mathsf{WBNA}[p]$ and $\mathsf{ABNA}[p] - \mathsf{WBNA}[p]$, respectively. In other words, we stop all monitors except those whose variables have active IFRs at that program point.

This inverted interface is an improvement over the simplified algorithm because now we do not need to add instrumentation before unknown function calls or at the end of a function. As long as we instrument every release call in the program, every monitor will be stopped at the first release call it encounters dynamically, unless the call is statically known to fall into an IFR for that variable. This means our detector has the surprising and useful quality that even though our compiler analysis is strictly intraprocedural, a dynamic monitor can start in one function and end in another. The other function might be the function's caller, a callee of the function, or even another function called long after the function returns.

This introduces a small soundness issue, since release calls in uninstrumented libraries, or indirect calls to primitive release functions, will not stop monitors. However, we have found that programs typically do not rely on synchronization in library code to protect shared data in the main program, so missing these release calls is a relatively minor problem. This problem is not fundamental to our algorithm; it would be possible to dynamically intercept release calls. In practice, we encountered only one case in our benchmarks where a program used function pointers for synchronization calls.

Statement type	Statement form	WBNA$[p]$	ABNA$[p]$
Load	p : `r` = ∗`x`;	WBNA$[p']$	ABNA$[p'] \cup \{x\}$
Store	p : ∗`x` = `r`;	WBNA$[p'] \cup \{x\}$	ABNA$[p'] \cup \{x\}$
Acquire	p : `mtx_lock`(`m`);	{}	{}
Release	p : `mtx_unlock`(`m`);	WBNA$[p']$	ABNA$[p']$
Call	p : `f`(...);	{}	{}
Other	p : . . . ;	WBNA$[p']$	ABNA$[p']$

Figure 5.8: Summary of our backwards data-flow analysis to insert instrumentation calls. p' is the program point after the statement at point p.

5.2.5 Data-Flow Analysis

Our compiler analysis is an intraprocedural backwards data-flow analysis. Working from the end of each function to the beginning, we identify variables that must be accessed on every path from a given program point to the next acquire call: WBNA and ABNA. (The refined analysis does not use ASLR.) The initial values (at the end of the function) are WBNA$[p_{\mathsf{end}}]$ = ABNA$[p_{\mathsf{end}}] = \{\}$. The sets propagate through statements as shown in Figure 5.8. At load and store operations we update the WBNA and ABNA sets. The sets get killed at acquire calls and unknown function calls. At control-flow merge points, we take the intersection of the incoming sets. We implemented this analysis in the LLVM compiler framework [26].

5.2.6 Short-Scope Monitors

In the previous section, we discussed inserting calls to `start_monitors` at the beginning of basic blocks or after unknown function calls. However, in some cases, instrumenting at these locations is not possible, because one or more variables for which a monitor is being started are not in scope. For example, consider the following loop:

```
int array[10]; // global variable

...
int i = 0;
int *x;
```

```
do {
  x = &array[i];
  *x = i;
  i++;
} while (i < 10);
```

Without analyzing the compiled version of this program, we can infer that there will be at least 10 different IFRs per execution: one for each value of `x`. Therefore we cannot simply insert a single `start_monitors` call for `x` before the loop. When translated to SSA form, `x`'s definition is inside the loop:

```
int *array; // global variable

entry:
  array = ...;
  goto loop;
loop:
  int i_1 = PHI [0, entry] [i_2, loop];
  int *x = array + i_1;
  store i_1 into x;
  int i_2 = i_1 + 1;
  if (i_2 < 10) goto loop else goto done;
done:
  return;
```

Our analysis will discover that the monitor for `x` should start at the beginning of the entry block; however, `x` is not in scope at the beginning of the entry block. Practically, the earliest we can start the monitor for `x` is after `x` is initialized:

```
...
int *x = array + i_2;
start_strong_monitor(x);
store i_2 into x;
...
```

Placing the call within the body of the loop has the effect of starting an IFR for each element in the array, which is what we expected from examining the source code.

Our instrumentation pass therefore works as follows: for every monitor start whose variable is not in scope, we insert a special call (either to `start_weak_monitor` or to `start_`

`strong_monitor`) that starts a single monitor right after the variable's definition. We call such monitors *short-scope monitors*, because the scope of the variable being monitored limits the duration of the monitor. We have found that there tend to be many short-scope monitor starts in program executions, since typically such calls cover exactly one memory load or store. Since handling all of these calls can be very expensive, our dynamic analysis can start monitors for only a subset of these calls in order to recover performance; this will be discussed in more detail in Section 5.3.4.

5.3 Dynamic Analysis

There are two parts to IFRit's dynamic analysis. First, IFRit tracks which threads have active monitors for which memory locations. Second, IFRit detects races by identifying conflicting monitors for the same location in different threads.

5.3.1 Dynamic Monitors

The static analysis informs the dynamic analysis of program points where monitors should start and stop. At run-time, IFRit maintains a data structure called the *Active Monitors Table* (AMT). The AMT maps each memory location to a set of *monitor records* for that location. There is one monitor record for each thread executing a monitor for a particular memory location. A monitor record stores the program counter where the monitor began, the thread ID of the thread executing the monitor, and whether the monitor is weak or strong. Each thread also maintains two thread-local sets of memory locations representing active weak and strong monitors.

Following the key insight of the FastTrack algorithm [18], the AMT holds at most one strong monitor per memory location at a time. In the data-race-free case, there is no need to store more than one monitor, since writes to a memory location are totally ordered. If more than one thread starts a strong monitor for a given location concurrently, the tool will report a data race. This optimization might result in fewer data-race detections, but only for executions where at least one data race is reported.

When a thread reaches a call to `start_monitors`, it looks up each argument in the AMT, adds a monitor record to the table's entry for each argument (unless one is already active),

and updates its local sets. When a thread encounters a call to `stop_all_monitors_except`, it iterates through its local sets, removing monitor records from the AMT for all memory locations in the local sets except those listed as arguments.

5.3.2 Detecting Data Races

IFRit detects data races using the information stored in the AMT. When a thread reaches a call to `start_monitors`, it performs a *race check* on every memory location passed to `start_monitors` (except those for which monitors are already active) before updating the AMT. To perform the race check, the thread looks at the set of monitor records associated with each memory location.

If the thread performing the race check is starting a strong monitor, and another thread already has an active monitor (weak or strong) for the location, IFRit concludes there is a data race. If the thread performing the check is starting a weak monitor, IFRit concludes there is a data race only if another thread has an active strong monitor for that location.

When a thread detects a data race, it reports its current program counter, and the program counter stored in the monitor record that the thread found in the AMT.

5.3.3 Implementation

We implemented IFRit's dynamic analysis from scratch in a run-time library. The library's API exposes the `start_monitors`, `start_strong_monitor`, `start_weak_monitor` and `stop_all_monitors_except` functions. The runtime implements the AMT as two arrays of 2^n hash tables, where n is a small positive integer—that is, 2^n pairs of hash tables, where each pair includes one hash table for strong monitor records and one hash table for weak monitor records.[7] Monitor records are assigned to the appropriate hash table in the array by masking off n bits in the monitor's associated memory location. We found that partitioning the AMT in this way was extremely valuable for regaining parallelism, as compared to earlier designs in our development process that used just two hash tables for all monitor records.

[7]The results presented in Section 5.5 use $n = 5$.

Each pair of hash tables in the AMT is synchronized using a mutex lock. In addition to preventing the hash tables from being corrupted by concurrent accesses, this simple synchronization scheme also has the effect of serializing monitor starts for each location.

The threads' sets of monitors are implemented as two thread-local hash tables, one for active strong monitors and one for active weak monitors. Because this information is stored locally, many calls to the runtime do not need to do any synchronization—they simply check to see whether the monitor is already active (in the case of `start_monitors` or its variants) or whether there are any active monitors that need to be stopped (in the case of `stop_all_monitors_except`).

5.3.4 Performance Considerations

IFRit has a strong correctness guarantee: even if not all monitors are started, we will report no false positives, as long as monitors are stopped at the appropriate time (or earlier). Therefore, we can ignore some calls to `start_monitors` without compromising soundness. We leverage this in two ways: limiting short-scope monitors, and sampling.

First, as discussed in Section 5.2.6, so-called "short-scope monitors" are numerous enough to be a burden on the runtime. A common case is that a thread will be iterating through a large array, which requires starting a new monitor on every iteration. The idea of our static instrumentation is to use a few calls to represent many accesses, so these small-scope calls are problematic. Therefore we have an optional mode for our detector that starts only a subset of these monitors. Specifically, we allow a maximum of k dynamic monitors per static call site to be live at the same time. This optimization is intended to exploit the observation that if one iteration of the loop is racy, it is likely that the rest will be racy as well. We have found that this optimization provides considerably better performance while catching almost as many races as the fully-monitored mode. We did find one race which was missed by this optimization: a loop in one of the PARSEC benchmarks (streamcluster) was not racy for its first 512 iterations, but was racy for the rest.

Second, we implemented sampling. Our runtime executes a *sampling period* for a window of execution every second. During a sampling period, the runtime executes all calls to

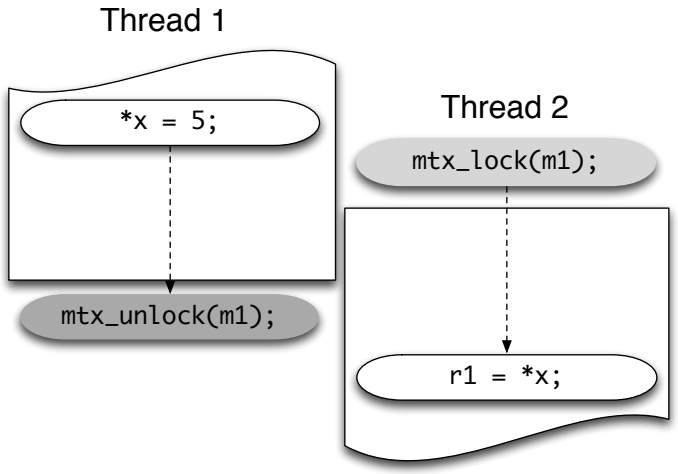

Figure 5.9: Even though neither access happens during the other access's IFR, we can detect the race in this case because the accesses' IFRs overlap.

`start_monitors` and its variants. For instance, with a sampling rate of 1%, IFRit monitors the execution for one one-hundredth of a second every second. When the period ends, we ignore calls to `start_monitors` and its variants. We chose this sampling technique because we suspect monitoring many memory locations simultaneously finds more bugs than sparsely sampling monitors at all times. Sampling is effective: at a sampling rate of 50%, overheads dropped an order of magnitude.

We also implemented an optimization for programs that have long single-threaded phases: if there is only one thread running, we ignore calls to `start_monitors`. This affects neither soundness nor completeness: once the thread finishes its work, it must call `thrd_create` to start a new thread. `thrd_create` is a release operation. Therefore any monitors collected during the single-threaded phase would be stopped before the `thrd_create` call anyway, so there is no point to starting these monitors.[8]

5.4 Formalism and Correctness

This section proves that the central idea of our detector is correct: if two interference-free regions for conflicting, concurrent accesses overlap, then the accesses must form a data race. The property we prove here is stronger than Theorems 3.1 and 3.2, because the racing access may not happen during the other access's IFR (see Figure 5.9 for an example).

As in Chapter 3, an execution of a program is a triple $(A, \leq_{\mathsf{po}}, <_{\mathsf{sw}})$: a set of actions A, the program order $\leq_{\mathsf{po}}$, and the synchronizes-with order $<_{\mathsf{sw}}$. The happens-before order $\leq_{\mathsf{hb}}$ is the reflexive transitive closure of $\leq_{\mathsf{po}}$ and $<_{\mathsf{sw}}$.

Our goal is to prove that two overlapping IFRs for conflicting, concurrent accesses to the same variable always imply that the accesses form a data race. This is stated in the following theorem:

Theorem 5.1. *Consider two IFRs I_1 and I_2 for actions (t_1, k_1, u_1) and (t_2, k_2, u_2). Assume that $t_1 \neq t_2$ and that k_1 and k_2 are either $\mathsf{read}(x)$ or $\mathsf{write}(x)$, and at least one is a write. Then if I_1 and I_2 overlap, (t_1, k_1, u_1) and (t_2, k_2, u_2) form a data race.*

First, we define interference-free regions and data races with respect to our formal model.

Definition 5.1 (IFR)**.** *An IFR is a triple $I = (u^{\text{begin}}, u^{\text{access}}, u^{\text{end}})$ where the following conditions hold:*

1. *There exist t, k^{access}, and x such that $(t, k^{\text{access}}, u^{\text{access}}) \in A$ and either $k^{\text{access}} = \mathsf{read}(x)$ or $k^{\text{access}} = \mathsf{write}(x)$.*
2. *$u^{\text{begin}} <_{\mathsf{po}} u^{\text{access}} <_{\mathsf{po}} u^{\text{end}}$.*
3. *For all u such that $u^{\text{begin}} <_{\mathsf{po}} u \leq_{\mathsf{po}} u^{\text{access}}$, u's associated kind is not an acquire synchronization.*
4. *For all u such that $u^{\text{access}} \leq_{\mathsf{po}} u \leq_{\mathsf{po}} u^{\text{end}}$, u's associated kind is not a release synchronization.*

Definition 5.2 (Data race)**.** *Two actions (t_1, k_1, u_1) and $(t_2, k_2, u_2) \in A$ form a data race if:*

[8] The `thrd_create` call might allow some monitors to continue through it, but we do not think this is a concern. Ignoring these monitors does not affect soundness, and it would be very easy to special-case calls to `stop_all_monitors_except` so that the monitors would be started before the `thrd_create` call.

1. $t_1 \neq t_2$;
2. k_1 and k_2 are either reads or writes of the same variable, and at least one is a write;
3. and the two actions are not ordered by happens-before: $u_1 \not\leq_{\mathsf{hb}} u_2$ and $u_2 \not\leq_{\mathsf{hb}} u_1$.

Suppose we have two IFRs I_1 and I_2 in a given execution. I_1 and I_2 do *not* overlap if either I_1 ends before I_2 begins, or I_2 ends before I_1 begins. Therefore, we say that two IFRs overlap if neither of these conditions holds:

Definition 5.3 (Overlapping IFRs)**.** *Two IFRs* $I_1 = (u_1^{\text{begin}}, u_1^{\text{access}}, u_1^{\text{end}})$ *and* $I_2 = (u_2^{\text{begin}}, u_2^{\text{access}}, u_2^{\text{end}})$ overlap *if* $u_1^{\text{end}} \not\leq_{\mathsf{hb}} u_2^{\text{begin}}$ *and* $u_2^{\text{end}} \not\leq_{\mathsf{hb}} u_1^{\text{begin}}$.

In order to prove our main theorem, we first prove a supporting lemma about the structure of happens-before edges. Effectively, we need to show that in order for there to be a happens-before edge between two actions in different threads, there must be a release synchronization action in the first thread that is sequenced after the first action, and an acquire synchronization action in the second thread that is sequenced before the second action.

Lemma 5.1. *Let* $(t_1, k_1, u_1), (t_2, k_2, u_2) \in A$ *such that* $t_1 \neq t_2$ *and* $u_1 \leq_{\mathsf{hb}} u_2$*. Then there exist* u_3 *and* u_4 *such that* $u_1 \leq_{\mathsf{po}} u_3 \leq_{\mathsf{hb}} u_4 \leq_{\mathsf{po}} u_2$*,* u_3*'s associated kind is a release synchronization, and* u_4*'s associated action is an acquire synchronization.*

Proof. Proof by induction on $u_1 \leq_{\mathsf{hb}} u_2$.

- As $t_1 \neq t_2$, $u_1 \not\leq_{\mathsf{po}} u_2$.
- If $u_1 <_{\mathsf{sw}} u_2$, let $u_3 = u_1$ and $u_4 = u_2$. As $t_1 \neq t_2$, u_3 has an outgoing edge and u_4 has an incoming edge. By reflexivity of $\leq_{\mathsf{po}}$, we have that $u_1 \leq_{\mathsf{po}} u_3 \leq_{\mathsf{hb}} u_4 \leq_{\mathsf{po}} u_2$.
- If $u_1 \leq_{\mathsf{hb}} u_5 \leq_{\mathsf{hb}} u_2$, let t_5 be the thread ID for u_5. Either $t_5 = t_1$, $t_5 = t_2$, or $t_5 \neq t_1$ and $t_5 \neq t_2$.
  - $t_5 = t_1$. Then $t_5 \neq t_2$, so by the inductive hypothesis there exist u_6 and u_7 such that $u_5 \leq_{\mathsf{po}} u_6 \leq_{\mathsf{hb}} u_7 \leq_{\mathsf{po}} u_4$, u_6 has an outgoing edge, and u_7 has an incoming edge. Let $u_3 = u_6$ and $u_4 = u_7$. As $t_5 = t_1$ and $u_1 \leq_{\mathsf{hb}} u_5$, it must be that $u_1 \leq_{\mathsf{po}} u_5$, and by transitivity of $\leq_{\mathsf{po}}$, $u_1 \leq_{\mathsf{po}} u_6$. Therefore $u_1 \leq_{\mathsf{po}} u_6 \leq_{\mathsf{hb}} u_7 \leq_{\mathsf{po}} u_2$.

- $t_5 = t_2$. Then $t_5 \neq t_1$, so by the inductive hypothesis there exist u_6 and u_7 such that $u_1 \leq_{\mathsf{po}} u_6 \leq_{\mathsf{hb}} u_7 \leq_{\mathsf{po}} u_5$, u_6 has an outgoing edge, and u_7 has an incoming edge. Let $u_3 = u_6$ and $u_4 = u_7$. As $t_5 = t_2$ and $u_5 \leq_{\mathsf{hb}} u_2$, it must be that $u_5 \leq_{\mathsf{po}} u_2$, and by transitivity of $\leq_{\mathsf{po}}$, $u_7 \leq_{\mathsf{po}} u_2$. Therefore $u_1 \leq_{\mathsf{po}} u_6 \leq_{\mathsf{hb}} u_7 \leq_{\mathsf{po}} u_2$.
- $t_5 \neq t_1$ and $t_5 \neq t_2$. We apply the inductive hypothesis twice. First, there exist u_6 and u_7 such that $u_1 \leq_{\mathsf{po}} u_6 \leq_{\mathsf{hb}} u_7 \leq_{\mathsf{po}} u_5$ and u_6 has an outgoing edge. Second, there exist u_8 and u_9 such that $u_5 \leq_{\mathsf{po}} u_8 \leq_{\mathsf{hb}} u_9 \leq_{\mathsf{po}} u_2$ and u_9 has an incoming edge. Let $u_3 = u_6$ and $u_4 = u_9$. By transitivity of $\leq_{\mathsf{hb}}$, we have that $u_1 \leq_{\mathsf{po}} u_6 \leq_{\mathsf{hb}} u_9 \leq_{\mathsf{po}} u_4$. □

Lemma 5.1 leads directly to the proof of Theorem 5.1.

Proof. Let $I_1 = (u_1^{\text{begin}}, u_1^{\text{access}}, u_1^{\text{end}})$ and $I_2 = (u_2^{\text{begin}}, u_2^{\text{access}}, u_2^{\text{end}})$. Assume that the two accesses do not form a data race; i.e. that either $u_1^{\text{access}} \leq_{\mathsf{hb}} u_2^{\text{access}}$ or $u_2^{\text{access}} \leq_{\mathsf{hb}} u_1^{\text{access}}$. Proceed by cases:

1. $u_1^{\text{access}} \leq_{\mathsf{hb}} u_2^{\text{access}}$. By Lemma 5.1, this happens-before edge must go through a release action in Thread t_1, and an acquire action in Thread t_2. Formally, there exist u_3 and u_4 such that $u_1^{\text{access}} \leq_{\mathsf{po}} u_3 \leq_{\mathsf{hb}} u_4 \leq_{\mathsf{po}} u_2^{\text{access}}$, u_3 is a release synchronization, and u_4 is an acquire synchronization. By Definition 5.1, it must be the case that the release and acquire actions do not fall in I_1 and I_2, respectively: $u_1^{\text{end}} \leq_{\mathsf{po}} u_3$ and $u_4 \leq_{\mathsf{po}} u_2^{\text{begin}}$. By transitivity of happens-before, we have that $u_1^{\text{end}} \leq_{\mathsf{hb}} u_2^{\text{begin}}$, contradicting our assumption that the two IFRs overlap.
2. $u_2^{\text{access}} \leq_{\mathsf{hb}} u_1^{\text{access}}$. This case is symmetric to the first. □

We have therefore proved that our algorithm produces no false positives.

5.5 Evaluation

There are four main goals to our evaluation of IFRit: (1) We highlight IFRit's low runtime overheads and characterize the impact of sampling on IFRit's overheads; (2) We demonstrate that IFRit effectively detects data races in several mature applications and assess the impact

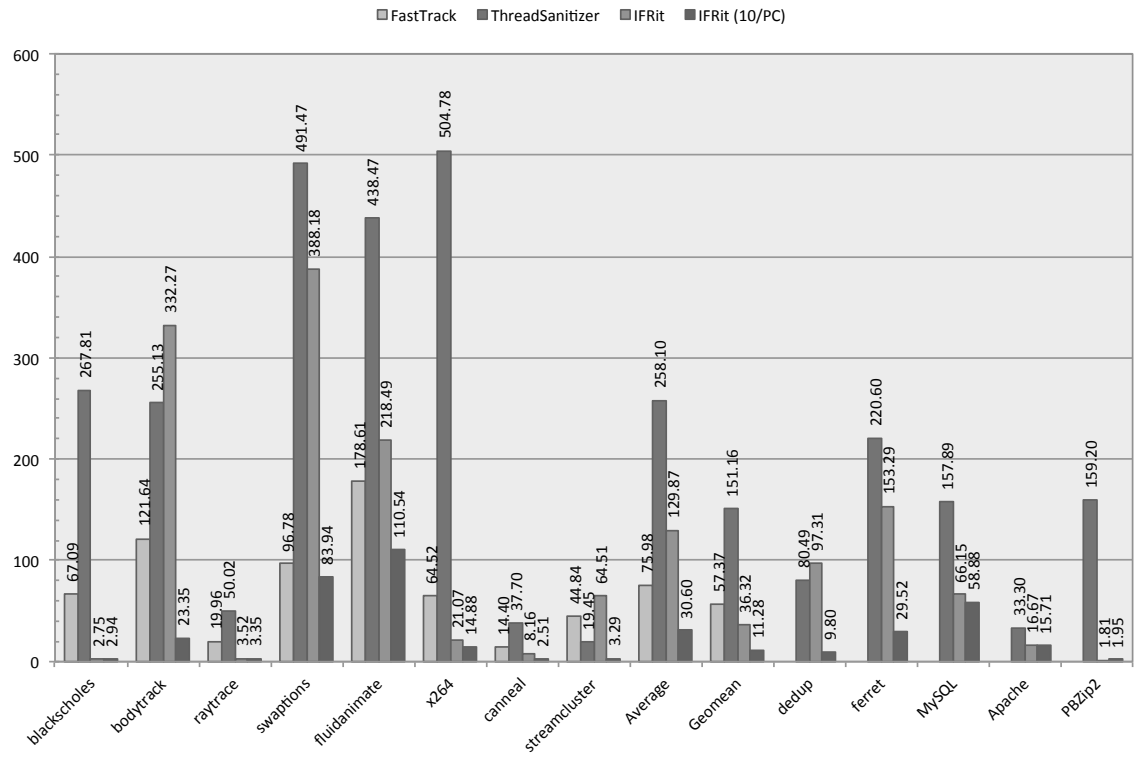

Figure 5.10: Overhead of IFRit compared to uninstrumented code for the PARSEC benchmarks and a suite of real applications. Average and geometric mean are over the first eight PARSEC benchmarks.

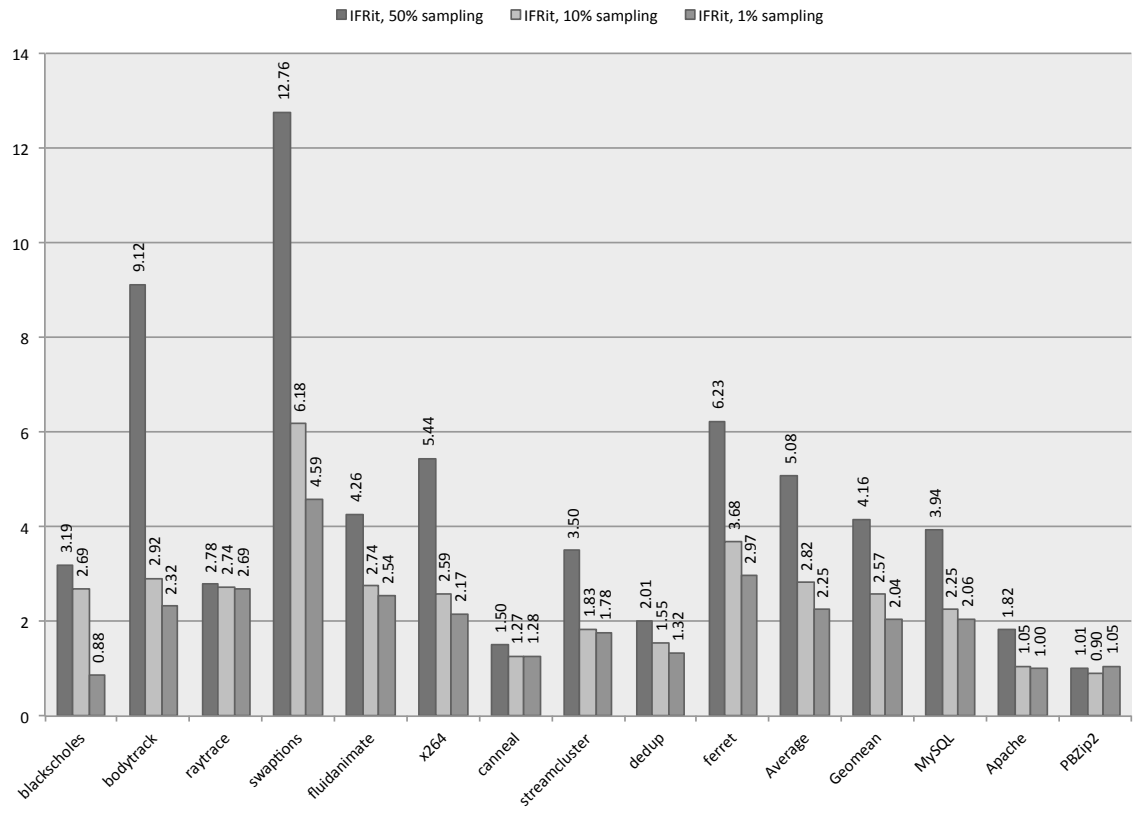

Figure 5.11: Effect of sampling on IFRit's performance overhead. Average and geometric mean are over the ten PARSEC benchmarks.

of sampling on IFRit's race-detection capability; (3) We qualitatively analyze the output of IFRit by examining several discovered races; and (4) Throughout our evaluation, we provide a head-to-head comparison with ThreadSanitizer, a state-of-practice happens-before data-race detection tool with widespread commercial adoption and FastTrack, a state-of-the-art happens-before data-race detection tool.

5.5.1 Experimental Setup

To benchmark IFRit, we used the PARSEC-2.1 benchmark suite [7] and a set of real applications. We ran the PARSEC benchmarks with their 8 threaded `pthreads` configuration on the simsmall input set. We excluded three of the 13 benchmarks: one, freqmine, used OpenMP for synchronization, and our runtime currently suports only pthreads; a second, vips, used GLib for synchronization, which our runtime uses for hash tables and therefore cannot be instrumented; a third, facesim, crashed during our tests due to memory requirements.

To evaluate IFRit further, we used unmodified versions of MySQL, Apache, and PBZip2. MySQL is an industrial-strength database server. We used MySQL-5.5.15, running with its default configuration. To benchmark MySQL, we used the sysbench OLTP benchmark running under its default configuration. Apache is a webserver. We used version httpd-2.0.48 with its "worker" thread configuration. We ran tests using ApacheBench, issuing 10000 requests from 8 request threads. PBZip2 is a parallel file compression/decompression tool. We used PBZip2-0.9.1, running with 8 threads. To benchmark PBZip2, we decompressed a 150MB text file.

We compiled applications using LLVM 3.0 and our instrumenting compiler pass. For our baseline, we compiled applications using LLVM 3.0, but without our instrumenting pass. The applications were run on a machine with two 4-core Intel Xeon processors clocked at 2.8 GHz with 16GB of RAM.

In our evaluation we directly compared IFRit to ThreadSanitizer's Valgrind implementation [43] and an implementation of FastTrack for C/C++ using DynamoRio [39]. We ran experiments with ThreadSanitizer on our machines. The authors of [39] provided us with

data from their experiments with their FastTrack implementation.

5.5.2 Overheads

PARSEC Figure 5.10 shows the overheads imposed by IFRit on the PARSEC benchmarks compared to FastTrack and ThreadSanitizer. FastTrack data was available for only the first eight PARSEC benchmarks, so we have listed the average and geometric mean for those programs only. The geometric mean de-emphasizes the effect of outliers. We ran each PARSEC program three times for each sampling rate and used the mean of the three execution times. In addition to the fully instrumented IFRit data, we show the overheads for a variant of IFRit where short-scope monitors are limited to a maximum of ten monitors per static call site at a time.

IFRit's overheads are low. On all but three PARSEC benchmarks, the fully instrumented version of IFRit outperforms ThreadSanitizer. For four of the eight benchmarks for which we have FastTrack numbers, IFRit outperforms FastTrack. The geometric mean of IFRIT's slowdown across the entire PARSEC suite is 46.3x, compared to 147.4x for ThreadSanitizer and 57.3x for FastTrack. If we enable the short-scope optimization, which limits the number of monitors per static call site, IFRit performs better than both FastTrack and ThreadSanitizer on every PARSEC benchmark, with an overall geometric mean of 12.2x.

Overall, these data show that IFRit's overheads are comparable to prior race detection tools [39, 43, 18]. In most of our benchmarks, monitors are started and stopped infrequently enough that the cost of our instrumentation is amortized by program execution. In these cases, IFRit's low overhead results from not having to instrument every memory access. For benchmarks with a large number of short-scope monitors, selectively omitting some monitors on a per-call-site basis is extremely effective in recovering performance without sacrificing much coverage (we discuss coverage more in Section 5.5.3).

Real applications Figure 5.10 also shows overheads for the real applications compared to uninstrumented execution. For these applications, we ran the benchmarking code only once per configuration. IFRit's overhead running on real applications is similar to the overheads we saw for PARSEC. Our best case is PBZip, with overheads around 4x. IFRit's worst case

full application is MySQL, which incurs a 66X overhead. While higher than the overheads in Apache and PBZip, IFRit's overhead is far lower than ThreadSanitizer's overhead of around 160X. The short-scope monitor optimization reduces MySQL's overhead to 59X. The difference indicates that short-scope monitors contribute to MySQL's overhead. Both PBZip and Apache saw little benefit from the short-scope monitor optimization, suggesting their performance is not limited by starting and stopping short-scope monitors.

Impact of Sampling on Performance

Figure 5.11 gives the overheads for IFRit with sampling enabled for 1%, 10% and 50% of the execution time. We give the average and geometric mean for all 10 PARSEC benchmarks. Sampling is very effective at reducing IFRit's overheads for PARSEC, with a geometric mean of 4.2x, 2.6x, and 2.0x slowdown for 50%, 10% and 1% sampling, respectively. Sampling also helps a great deal for the some of the real applications. MySQL runs much faster under sampling (15-30 times faster), but under sampling, no data races are detected (see Table 5.1). Apache, on the other hand, runs with nearly no overhead under sampling, and still detects many data races – half of the races reported without sampling are reported with a 50% sampling rate, and 30% of the races reported without sampling are still reported with a 1% sampling rate. PBZip also enjoys nearly no overhead with 50% sampling and still detects all the races reported by IFRit without sampling.

5.5.3 Race-Detection Coverage

Table 5.1 lists the number of unique races reported by our tool for the benchmarks. We found races in all three real applications and in four of the 13 PARSEC benchmarks. To assess the coverage of IFRit, we directly compare to the coverage of ThreadSanitizer. We discuss the races found by IFRit and ThreadSanitizer in Section 5.5.4. The data show that in each of the PARSEC programs that had any races reported, ThreadSanitizer detects some races that IFRit did not detect. The programs with the biggest difference in coverage are ferret and x264. In x264, ThreadSanitizer found 72 races while IFRit found only 3. In ferret, IFRit missed all of the races ThreadSanitizer reported. As we shall discuss in

	IFRit					
	1%	10%	50%	10/PC	Full	ThreadSanitizer
bodytrack	1	1	1	5	5	10
x264	–	–	2	3	3	72
streamcluster	1	1	2	2	3	24
ferret	–	–	–	–	–	38
Apache	6	8	10	19	19	21
PBZip	–	–	2	2	2	2
MySQL	–	–	–	11	11	14

Table 5.1: Number of unique races found by IFRit in various configurations and by ThreadSanitizer. Omitted benchmarks had no detected races.

Section 5.5.4, many of these races are related to memory accesses in code not instrumented by IFRit. Note that missing these races is a limitation of our prototype, not a fundamental limitation of our IFR-based approach.

In contrast, in the real application benchmarks we used, IFRit's coverage is nearly identical to ThreadSanitizer's coverage. IFRit and ThreadSanitizer detect the same races as PBZip. IFRit misses two races in Apache, and three races in MySQL.

Impact of Short-Scope Monitor Optimizaton on Coverage

When we limit the number of short-scope monitors per code point, IFRit's coverage is identical in all cases except streamcluster. Streamcluster executes a loop that starts short-scope monitors. The memory accesses in the first 512 iterations of the loop are not racy, but the remaining accesses are racy. The accesses occur at the same code point, so we miss these races with this optimization enabled.

Looking back to Figure 5.10, the data show that the reduction in overhead resulting from this optimization is very large. The data in Table 5.1 show that the degradation of coverage is almost negligible. Together these results demonstrate that limiting the number of short-scope monitor's per code point is beneficial.

Impact of Sampling on Coverage

Sampling reduces IFRit's coverage, but even with sampling IFRit detects many data races. Sampling 50% of the execution, IFRit detects some races in all programs in which it detected races without sampling, except MySQL. Using even sparser sampling further reduces IFRit's coverage. However, even with a sample rate of 1% IFRit still detects races in streamcluster, bodytrack, and Apache.

The data in Figure 5.11 show that sampling reduces overheads considerably—the geometric mean overhead at 1% sampling rate is about 2X, and only slightly higher at 10% sampling rate. The data in Table 5.1 show that IFRit is still useful for finding data races when sampling is active. Together, these results show that sampling is one way to trade off precision for increased performance.

5.5.4 Analysis of Detected Races

In order to track down these reported data races, we compiled and ran a second version of each racy benchmark with debugging information and less aggressive optimization. Our tool prints out the program counter for the `start_monitors` call for each side of the data race, as well as a stack trace for the call that triggered the report. The static analysis also prints a list of instrumentation calls and their associated accesses.

Races in PARSEC

Most of the PARSEC benchmarks had no races reported by either IFRit or ThreadSanitizer. (We did not have access to the DynamoRIO FastTrack race reports, but the paper mentions a race in canneal which neither IFRit nor ThreadSanitizer reported.) Both tools found races in bodytrack, x264, and streamcluster. ThreadSanitizer also found races in ferret that were not detected by IFRit.

Bodytrack IFRit found five data races in bodytrack, four of which were caused by the same bug involving the misuse of condition variables. The last race was caused by threads reading a structure that had not been fully initialized.

ThreadSanitizer reported 10 unique races, including the two problems identified by IFRit. ThreadSanitizer also found a race involving an unprotected counter that was not reported in IFRit. However, that race did show up in IFRit during runs run with a larger input (simmedium), and running IFRit on the simmedium input was faster than running ThreadSanitizer on the simsmall input. Fixing these three root causes resolved all of the race reports from both ThreadSanitizer and IFRit.

X264 IFRit reported three data races in x264, one of which was confirmed by ThreadSanitizer. ThreadSanitizer reported 72 races, most of them within memcpy in libc, which was not instrumented by IFRit's static analysis and therefore was not monitored for races.

Streamcluster IFRit reported three data races in streamcluster. Two of the races were on local variables declared `static`. `static` local variables are scoped to their function or block, but correspond to a single global object, so threads executing the function simultaneously can race on the variable. The third race was caused by a missing barrier call. It appears that the pthreads code was improperly translated from code that used Intel's TBB (Threading Building Blocks) Library.[9]

ThreadSanitizer reported 23 unique races in streamcluster, including the three reported by IFRit. We determined that the remaining races reported by ThreadSanitizer were due to two root causes. First, a function passed its arguments by value rather than by reference; since pass-by-value arguments are not listed as loads in the LLVM IR, IFRit did not instrument those memory accesses. The second race was on a pointer being freed, which ThreadSanitizer counts as a write and IFRit does not.

Ferret ThreadSanitizer found 43 races in ferret that were not reported by IFRit. Two races, one on a shared counter and a second on a shared boolean flag, were not detected by IFRit because the racy monitors in IFRit were of very short duration, and never happened to overlap. The remaining races were in libc, which was not instrumented by IFRit's static analysis and therefore was not monitored.

---

[9] `http://threadingbuildingblocks.org/`

Races in Real Applications

MySQL IFRit reported 11 races in MySQL. Three races in MySQL were the result of unsynchronized accesses to flags written in the main program thread and read in a signal handling thread during server shutdown. Two reported races involved lock meta-data in MySQL's wrapper for pthread locks.

The remaining races are on unsynchronized flags and a linked list implementation in debugging code. These races are unsurprising. Debugging code is often disabled in production, so it may be less thoroughly tested than other code.

IFRit and ThreadSanitizer had comparable coverage for MySQL. ThreadSanitizer reported 14 races in MySQL, including 9 of the 11 races that IFRit reported. ThreadSanitizer did not report two races IFRit reported and IFRit did not report four races that ThreadSanitizer reported.

Apache IFRit reported 19 different races in Apache. Seven were caused by a well-known bug in Apache's logging code that can lead to garbled log output [32, 34, 51]. Five more were caused by races that nearby comments indicated were known or intentional. Intentional or not, these races should be reported because even "benign" races can result in incorrect behavior [10]. The other races were all on improperly synchronized flags.

IFRit has nearly the same race detection coverage as ThreadSanitizer. IFRit detected all the races reported by ThreadSanitizer except for two. ThreadSanitizer did not report one of the two flag races that IFRit detected.

PBZip IFRit reported two races in PBZip. One of the races involves unsynchronized accesses to a flag variable signaling a termination condition to worker threads.

The other race involves concurrent accesses to fields of an output buffer structure. One thread fills the buffer and writes the fields. Concurrently, the thread that empties the buffer reads the fields without synchronizing.

The races reported by IFRit were the same races reported by ThreadSanitizer.

5.5.5 Discussion: IFRit vs. Other Detectors

Throughout this evaluation, we have compared IFRit directly to FastTrack and ThreadSanitizer. Like these precise detectors, IFRit is sound, so for all three there are no false positive races reported. FastTrack and ThreadSanitizer are also complete, meaning they detect all races in an execution. IFRit is not complete, but the data show that IFRit exploits a critical tradeoff of completeness for performance.

Figure 5.10 shows that IFRit's overhead is much lower than FastTrack and ThreadSanitizer. For the PARSEC programs, IFRit consistently outperformed the other techniques with our short-scope monitor optimization enabled. Comparing IFRit's overhead on our real application benchmarks to the overhead of ThreadSanitizer, IFRit is the clear winner with overheads far less than ThreadSanitizer. IFRit's performance advantage is a key distinction from prior techniques.

Table 5.1 shows that IFRit detects most of the races detected by FastTrack and ThreadSanitizer in the application code of the programs we evaluated. While we provide no completeness guarantee, our data show that IFRit is a powerful tool for detecting data races.

Together our performance and coverage results illustrate that IFRit recovers a large amount of performance by trading off what we empirically found to be a small margin of completeness. We consider this tradeoff profitable, as the reduction in overhead makes data-race detection cheap enough for practical frequent use. FastTrack and ThreadSanitizer pay a very high performance cost to provide completeness guarantees. Their overhead may be a barrier to their frequent use by developers in practice.

5.6 Related Work

A variety of tools have been developed to help find data races. Static race detection tools [16, 38, 1] analyze program code, and attempt to prove the absence of data races in all program executions. Static techniques are useful in that they can statically prove a program is data-race-free, but they also must be conservative because they lack information that is available only during program execution. We will focus on dynamic techniques.

Dynamic race detectors mostly fall into two categories: happens-before detectors [18,

12, 36, 39, 43] and lockset detectors [42]. Lockset detectors like Eraser [42] track the locks held at each access and report a race if accesses to a location are not consistently protected by the same lock. These techniques are based on a heuristic—that every shared variable will be consistently protected by the same lock—which may lead to false positives. Although it is possible to reduce false positives by introducing more heuristics (e.g., read-only data), any false positives represent a waste of the developer's time. This problem with false positives also applies to hybrid techniques such as MultiRace [40], RaceTrack [52], and ThreadSanitizer's hybrid mode [43].

Happens-before detectors work by tracking the order of synchronization actions in order to determine if conflicting accesses are or are not ordered by happens-before. Typically, these algorithms use *vector clocks* [37], a data structure that tracks the relative timing between different threads of execution in a process. Such race detectors report a data race if two accesses to the same shared state occur are not ordered by the happens-before relation. ThreadSanitizer's non-hybrid mode (which we used for comparison to IFRit in Section 5.5) is a standard happens-before detector that uses valgrind to instrument binaries.

The current state-of-the-art implementation of vector clocks, FastTrack [18], achieves an average 8.5x slowdown and is fully precise — i.e., it produces no false positives and reports at least one race if the execution contained any races. FastTrack achieves this relatively low overhead by looking only for *shortest races*—i.e., if access A races with later accesses B and C, only the race with access B will be reported. Practically, this means that the algorithm only has to track the most recent writer for each shared variable. In its current form, IFRit performs comparably to FastTrack when either sampling enabled or the short-scope optimization are enabled. This is significant since FastTrack is implemented in a managed language (Java), while IFRit runs on unmanaged code (C/C++). Therefore the baseline for IFRit is much faster, meaning overheads for IFRit are more noticeable.

As dicussed in Section 5.5, a recent paper reimplemented FastTrack for x86 binaries using the DynamoRio instrumentation platform [39]. As expected, FastTrack is still more efficient than a standard happens-before detector (ThreadSanitizer), but its overheads are much more noticeable than the Java versions: a geometric mean of around 50x for a set of 10 PARSEC benchmarks. The authors of that paper reduce the overheads by about

50% using Aikido, a custom hypervisor that uses page faults to quickly detect conflicts. We compared IFRit to the non-Aikido version of FastTrack, since IFRit does not require a custom hypervisor. IFRit, even without any sampling enabled, outperforms FastTrack, with a geometric mean of 36.3x on eight of the ten benchmarks used for FastTrack (FastTrack's geometric mean for those eight was 57.4x). In turn, FastTrack detects more data races than IFRit, since FastTrack is a fully-precise algorithm.

Several other tools have been developed that use sampling to reduce the overhead of fully-precise vector-clock detectors. Pacer uses FastTrack during sampling periods, and also does a small amount of work during non-sampled periods to ensure *proportionality*: the number of races detected should scale linearly with the size of the sampling period [12]. Unlike Pacer, IFRit does not do any work during non-sampled periods (except to check a boolean flag), so we miss races where only one of the monitor starts is sampled. However, the relatively smaller number of instrumentation points in IFRit means that we can afford to sample for longer periods, which mitigates Pacer's concern about proportionality. Our overheads at 10% sampling are comparable to Pacer's at 10%, even though we are running on C/C++ code instead of Java.

LiteRace [36] also uses sampling to improve the performance of vector clocks. They use dynamic profiling to identify "cold" functions, which they hypothesize are more likely to contain unnoticed data races. This adaptive sampling is a technique we could adapt to IFR-based data race detection. LiteRace achieves low overheads via adaptive sampling and also by using logging to postpone race checks until after execution. Like us, LiteRace runs on unmanaged C/C++ code, although they instrument binaries rather than source code. IFRit has higher overheads than LiteRace, but we perform race checks at runtime instead of offline. IFRit's overheads with sampling are comparable to those for LiteRace with thread-local adaptive sampling.

DataCollider [17] is a heuristic detector that tries to catch data races in OS kernels "red-handed": it freezes one thread before a memory access, and sets a hardware watchpoint to trap writes to the memory location in other threads. This is similar to IFRit in that both try to identify accesses that happen at roughly "the same time." IFRit differs from DataCollider in that we do not require hardware watchpoints, so we can monitor many

variables simultaneously.

5.7 Conclusion

This chapter presented IFRit, a new dynamic data-race detection algorithm for arbitrary C and C++ programs based on the fundamental concept of interference-free regions. IFRit improves on prior work by coalescing the instrumentation for multiple accesses to the same variable, reducing runtime overhead, and by requiring no specialized hardware to detect races with no false positives.

IFRit is a natural approach to dynamic data-race detection without the overhead of tracking a full happens-before relation. Our prototype implementation of this algorithm indicates that we can detect races in real programs without inducing too much overhead.

Chapter 6

INTERPROCEDURAL IFR ANALYSIS

The previous chapters introduced interference-free regions and developed static analyses—the ABNA and ASLR analyses—that infer IFRs for the purposes of program optimization or static instrumentation. In this chapter, we extend these algorithms to be interprocedural; i.e., to consider more than one function at a time. As we shall demonstrate, this is a natural extension of the original analyses that neatly solves a shortcoming of those algorithms—namely, handling functions with internal synchronization.

6.1 Motivation

The compiler analysis presented in Chapter 4 is an *intraprocedural* (operating on one function at a time) analysis that prunes the modified/reference sets for explicit synchronization calls. However, there is a problem with this approach: the intraprocedural algorithms cannot handle calls that contain internal synchronization—that is, calls to functions that transitively call primitive synchronization functions. In particular, if a program uses wrapper functions for synchronization, an *interprocedural* (operating on multiple functions at a time) analysis is necessary for IFR-based optimization.

Consider the following example. Here a program uses wrapper functions for synchronization calls:

```
mtx_t *global_lock;

void acquire_global() {
  mtx_lock(global_lock);
}

void release_global() {
  mtx_unlock(global_lock);
}
```

```
void f() {
  r1 = *x;
  acquire_global();
  ...
  release_global();
  r2 = *x;
}
```

If we inline the calls to `acquire_global()` and `release_global()`, it becomes clear that the second read of `x` can be eliminated (as in Figure 3.2):

```
void f() {
  r1 = *x;
  mtx_lock(global_lock);
  ...
  mtx_unlock(global_lock);
  r2 = *x;
}
```

If we were to analyze this program (the inlined version) using the intraprocedural IFR analyses, the ASLR and ABNA analyses would find that `x` is interference-free for the calls to `mtx_lock()` and `mtx_unlock()`, respectively. Therefore a later optimization pass could safely eliminate the second read of x.

Now consider how the intraprocedural analyses described in Chapter 4 would behave for the *non-inlined* version of this program. Both the ASLR and ABNA analysis analyze each function separately. Since function `acquire_global()` contains no memory accesses, the ASLR analysis would conclude that no variables are interference-free at the call to `mtx_lock()`. (In a larger program, there may be other calls to `acquire_global()` that are not in an IFR for `x` or any other variables, so this conclusion is correct.) Symmetrically, the ABNA analysis would conclude that no variables are interference-free at the call to `mtx_unlock()` in function `release_global()`. Therefore our analysis from Chapter 4 would not remove the redundant load in the non-inlined code.

In general, with enough inlining, any intraprocedural analysis is as effective as an equivalent interprocedural analysis (except in the presence of recursion). We will focus instead on designing an interprocedural algorithm that allows us to apply IFR-based optimizations to both primitive synchronization calls and calls with internal synchronization.

To address this issue, we need a systematic mechanism for reasoning about the side effects of calls that may contain internal synchronization. This chapter presents such a mechanism, which distinguishes the local memory accesses of a function from its synchronization behavior. We will also discuss how to use interprocedural analysis to improve the instrumentation for IFR-based data race detection in Section 6.5.

Unfortunately, our experimental results (Section 6.4) are underwhelming. Specifically, there were only two benchmarks in which using interprocedural analysis improved the results of the optimization passes. As we shall discuss later, we suspect that the lack of sophisticated alias analyses for LLVM is negatively affecting our results, and therefore that the algorithms presented in this chapter are still of interest because they could be combined profitably with future alias analyses.

6.2 Synchronization Behavior

The example in Section 6.1 suggests a possible approach to this problem: it would be useful to characterize the *synchronization behavior* of each function. For example, function `acquire_global()` acts like an *acquire* call, and `release_global()` acts like a *release* call. With this information, we can determine which interference-free regions extend through a given function call. For example, if given the information that the synchronization behavior of `acquire_global()` is "acquire," we can determine that the IFR for access A includes the `acquire_global()` call and area B in this program:

```
A: r = *x;          ⎫
acquire_global();   ⎬ interference-free region for A
B: ...              ⎭
release_global();
```

Similarly, if given the information that the synchronization behavior of `release_global()` is "release," we can determine that the IFR for access B includes the release call and area A in this program:

```
acquire_global();
A: ...              ⎫
release_global();   ⎬ interference-free region for B
B: r = *x;          ⎭
```

As it turns out, determining which IFRs extend through user-defined function calls will not be quite enough information to perform IFR-based optimization across those calls. If the IFR of an access includes a user-defined function call, that guarantees that no *other* thread will modify the accessed variable during the IFR (without introducing a data race), but the *current* thread may modify the variable during the call. This was not a problem for primitive synchronization calls like `mtx_lock()` and `mtx_unlock()`, which do not do any visible thread-local writes. We will discuss how to solve this problem in Section 6.4. For now, we will focus on the first step: summarizing the synchronization behavior of each function.

6.3 Interprocedural Algorithm

The purpose of the synchronization behavior analysis is to assign to each user-defined function in the program a label describing the kind of synchronization that may be performed when the function is called (including synchronization in transitively-called functions). Ultimately, we will use this labeling to determine which variables might be concurrently modified by other threads during the call's execution.

6.3.1 Synchronization Behaviors

We distinguish five different synchronization behaviors:

1. NONE: no synchronization calls
2. ACQ: acquire calls only (no release calls)
3. REL: release calls only (no acquire calls)
4. ACQ/REL: acquire or release calls, and any release calls must be performed after all acquire calls
5. ANY: other (e.g., a release call followed by an acquire call)

We selected these five synchronization behaviors because they are the most useful for interference-freedom analysis. Specifically, variables that must be accessed before an ACQ call are interference-free for that call; variables that must be accessed after a REL function are interference-free for that call; and variables that must be accessed both before and after an ACQ/REL call are interference-free for that call.

```
void n() {
}

void a(mtx_t *m) {
  mtx_lock(m);
}

void r(mtx_t *m) {
  mtx_unlock(m);
}

void ar(mtx_t *m1, mtx_t *m2) {
  a(m1); a(m2); n(); r(m2); r(m1);
}

void x(mtx_t *m) {
  r(m); a(m);
}
```

Figure 6.1: Example of synchronization behavior in a program.

For example, in Figure 6.1, function `n()` has synchronization behavior NONE, `a()` has behavior ACQ, `r()` has behavior REL, `ar()` has behavior ACQ/REL, and `x()` has behavior ANY.

The five synchronization behaviors form the lattice shown in Figure 6.2. We use this lattice to join synchronization behaviors at merge points in the control flow of a function. For example, function `f()` (defined below) technically has synchronization behavior ACQ/REL: it may perform acquire or release calls, and any release calls will be performed after all acquire calls have been completed.

```
void f(mtx_t *m, bool b) {
  if (b) {
    mtx_lock(m);
  } else {
    mtx_unlock(m);
  }
}
```

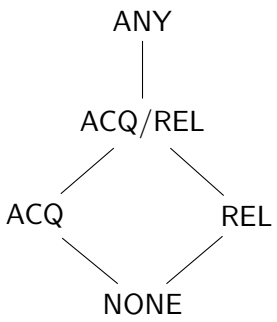

Figure 6.2: Lattice for synchronization behaviors.

Synchronization behavior	Bit representation
NONE	000
ACQ	001
REL	010
ACQ/REL	011
ANY	111

Table 6.1: Bit encodings for synchronization behaviors.

The first branch of the conditional has synchronization behavior ACQ; the second has synchronization behavior REL; overall, the function has behavior $\mathsf{ACQ} \sqcup \mathsf{REL} = \mathsf{ACQ/REL}$. Although this observation that "acquire OR release" can be treated like "acquire THEN release" is pleasingly elegant, we do not expect this pattern to occur often in real code. However, it simplifies our implementation because we can encode synchronization behavior using only 3 bits, with the join operator implemented as bitwise OR. Specifically, synchronization behaviors map to bit representations as shown in Table 6.1.

6.3.2 Interprocedural algorithm

Algorithm 6.1 (with helper algorithms 6.2, 6.3, and 6.4) gives the algorithm for determining the synchronization behavior for each user-defined function in a program. The final output of the algorithm is a map func_sync from functions to synchronization behaviors. We

Algorithm 6.1 analyze()

```
for each function f do
  if f is a user-defined function then
    Add f to func_worklist
    func_sync[f] ← NONE
  else if f is mtx_lock or thread_join or ... then
    func_sync[f] ← ACQ
  else if f is mtx_unlock or thread_create or ... then
    func_sync[f] ← REL
  else
    func_sync[f] ← NONE
  end if
end for
while func_worklist is not empty do
  f ← remove from func_worklist
  analyze_function(f)
end while
```

Algorithm 6.2 analyze_function(f)

```
for each basic block b in f do
  visited[b] ← false
  bb_sync[b] ← NONE
end for
Add entry block to bb_worklist
while bb_worklist is not empty do
  b ← remove from bb_worklist
  analyze_basic_block(b)
end while
s ← bb_sync[exit block of f]
if (s ≠ func_sync[f]) then
  Add callers[f] to func_worklist
  func_sync[f] ← s
end if
```

Algorithm 6.3 analyze_basic_block(b)

```
s ← bitwise OR of bb_sync of b's predecessors
for each statement S in b, in order do
  if S is an external call to function g then
    s ← update_sync(s, func_sync[g])
  else if S is a call to function g then
    f ← function being analyzed
    callers[g] ← callers[g] ∪ {f}
    s ← update_sync(s, func_sync[g])
  end if
end for
if (¬visited[b]) or (s ≠ bb_sync[b]) then
  Add b's successors to bb_worklist
  visited[b] ← true
  bb_sync[b] ← s
end if
```

Algorithm 6.4 update_sync(s_1, s_2)

```
if s2 = ACQ or s2 = ACQ/REL then
  if s1 ⊒ REL then
    s1 ← ANY
  else
    s1 ← s2
  end if
else if s2 = REL then
  s1 ← s1 ⊔ REL
else if s2 = ANY then
  s1 ← ANY
end if
return s1
```

initialize func_sync such that every function has behavior NONE. The algorithm maintains the following pieces of state, the last three of which should be reinitialized at each call to function analyze_function:

- func_worklist: a list of functions to be processed
- func_sync: a map from functions to their synchronization behavior
- bb_worklist: a list of basic blocks to be processed in the current function
- visited: a map from basic blocks to a boolean indicating whether the block has been visited during the current call to analyze_function
- bb_sync: a map from basic blocks to synchronization behaviors at the end of the block

Note that we initialize non-synchronization external functions to have synchronization behavior NONE. This includes even library calls that may perform synchronization, meaning the synchronization analysis is "unsound" in the sense that some calls may be labeled as having a weaker synchronization behavior than they do in reality. However, in the context of IFR-based compiler optimization, if an unknown external function could possibly contain synchronization, any sound alias analysis will treat calls of that function as opaque (potentially modifying any variable). Therefore, since the synchronization behavior analysis is always used in conjunction with a sound alias analysis, the combined result of the two analyses is sound. Many functions are known not to perform synchronization or have local side effects—e.g., `sin` and `cos`—so labeling them as having NONE synchronization behavior means that more optimization will be allowed. A more conservative approach would be to label unknown external functions as having ANY synchronization behavior; this variant ends up being a reasonable choice for IFR-based data-race detection, since in that context the synchronization behavior analysis is not combined with alias analysis and therefore its results should stand on their own. We will discuss this more in Section 6.5.

The algorithm is a standard interprocedural flow-sensitive data-flow analysis. As we process each function in the worklist, if its synchronization behavior changes from the behavior recorded in sync_func, we add its callers back into the worklist so their synchronization behavior can be updated to incorporate the newly-discovered behavior of their callees. We construct the list of callers for each function dynamically: whenever we encounter a function call, we add the current function to the callee's set of callers. (Note that the set of callers

is actually updated within Algorithm 6.3.)

Processing individual functions The intraprocedural portion of the algorithm, given in Algorithms 6.2 and 6.3, is a standard forward dataflow analysis, in which the data being flowed is the synchronization behavior. We maintain a worklist of basic blocks and a map from basic blocks to synchronization behaviors, where the synchronization behavior of a block respresents the synchronization from the beginning of the function to the end of the block. The algorithm proceeds by iteratively processing blocks in the worklist, adding their successors to the worklist if the block's synchronization behavior changes, until the algorithm reaches a fixpoint.

We use Algorithm 6.4 (update_sync) to update the current synchronization behavior if, in the course of analyzing a basic block, we encounter a statement that may perform synchronization. update_sync(s_1, s_2) returns the overall synchronization behavior of performing synchronization s_1 followed by s_2. If we have already seen acquire calls and we encounter a release call, update_sync returns ACQ/REL; if we have already seen release calls and we encounter an acquire call, update_sync returns ANY. Here are a few examples:

$$\text{update\_sync}(\text{ACQ}, \text{REL}) = \text{ACQ/REL}$$

$$\text{update\_sync}(\text{REL}, \text{ACQ}) = \text{ANY}$$

$$\text{update\_sync}(\text{NONE}, \text{REL}) = \text{REL}$$

Example Consider running Algorithm 6.1 on the program listed in Figure 6.1. For the purposes of the example, we process the functions in a non-optimal order; practically, it would be best to process them in bottom-up order in the call graph.

- First, we add all functions to the func_worklist, and set each function's initial synchronization behavior to NONE.
- We remove a function from the worklist to process. Suppose it is function `ar()`. Because none of its callees have been processed, Algorithm 6.2 determines that its synchronization behavior is NONE. In the course of processing `ar()`, we add it to the set of callers for `a()`, `n()` and `r()`.

- Next, suppose we process `n()`. We (correctly) determine that its synchronization behavior is NONE. Because the synchronization behavior of `n()` has not changed as a result of processing it, we do not add `n()`'s callers to the worklist.
- Next, suppose we process `a()`. We (correctly) determine that its synchronization behavior is ACQ. Because the synchronization behavior of `a()` has changed from its initial value of NONE, we add `a()`'s callers (currently just `ar()`) to the worklist.
- Next, suppose we process `r()`. As with `a()`, we update its synchronization behavior (this time to REL) and add its callers to the worklist. `ar()` is already in the worklist, so this has no effect.
- Next, suppose we process `ar()` again. Now that we have the correct synchronization behaviors for `a()` and `r()`, Algorithm 6.2 correctly determines that the synchronization behavior of `ar()` is ACQ/REL.
- Finally, we process `x()`. Algorithm 6.2 correctly determines that the synchronization behavior of `x()` is ANY. We do update the set of callers for `a()` and `r()` to include `x()`, but since the function worklist is empty, we are done.

6.4 Compiler Optimization

This section explains how we integrated the synchronization behavior analysis into IFR-based compiler optimization.

6.4.1 Handling Thread-Local Side Effects

The synchronization behavior analysis lets us determine when IFRs may propagate through calls of defined functions. However, this analysis alone is insufficient to allow elimination of redundant instructions across such calls. For example, consider the following:

```
r = *x;
...
update_shared_data();
...
r = *x;
```

where function `update_shared_data()` has behavior ACQ/REL. Given this synchronization behavior, we can infer that the variable `x` is interference-free in the region between the two

reads. Therefore it seems reasonable to remove the second read of `x`, since it appears to be redundant. However, what if the function itself writes to `x`? For example,

```
void update_shared_data() {
  acquire_global();
  ...
  release_global();
  ...
  *x = ...;
}
```

Now it is not safe to remove the second load of `x`. The same problem appears if a callee of `update_shared_data` may modify `x`. Such writes do not cause a data race because they are thread-local.

Let us assume for now that `update_shared_data()` does not modify `x`. To prove that the second read of `x` may be safely removed, we need information about the call's thread-local side effects. We therefore divide the problem into two subproblems:

1. Determining each call's possible synchronization behavior
2. Determining each call's possible thread-local side effects

Of course, the first subproblem is simply the synchronization behavior analysis presented in Section 6.2. By combining the results of these two analyses, we have enough information to enable interprocedural IFR-based optimization. In particular, we will implement a synchronization-aware *memory dependency* pass, which determines that the second read is *dependent* on the first read because (1) `x` is interference-free for the entire region between the two reads and (2) the intervening instructions, including the call to `update_shared_data()`, do not thread-locally modify `x`. We will discuss how to combine these two analyses more in Section 6.4.2.

Solving the second subproblem is conceptually quite simple, but practically more difficult. This problem is related to the standard compiler problem of determining the side effects (thread-local and non-thread-local) of a call. Typically, the alias analysis pass collects side-effect information for each call, and provides an interface for querying whether a call modifies a given variable. This includes both thread-local writes—such as store statements in the callee or its transitive callees—or potential writes in other threads, which are

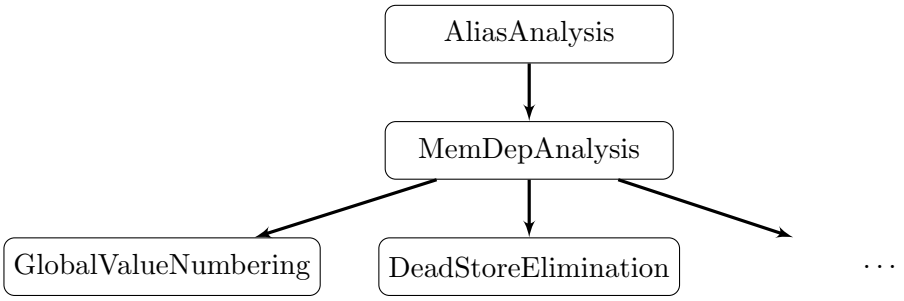

Figure 6.3: A subset of the LLVM analysis and optimization pipeline. Arrows indicate information flowing from one analysis to another.

indicated by the presence of synchronization calls. Therefore we can find thread-local side effects simply by modifying an existing alias analysis such that it ignores the synchronization calls when querying side effect information.

The practical difficulty arises in selecting an existing alias analysis to modify. We need an interprocedural alias analysis, so that the non-thread-local effects of calls with internal synchronization will be appropriately ignored in their callers. Alias analysis is an enormously complex problem, and sound, interprocedural alias analyses are difficult to find. In particular, we did not find any sophisticated interprocedural alias analysis for LLVM that was also sound for multithreaded programs [27, 20, 29].

LLVM is distributed with a simple interprocedural analysis called *GlobalsModRef*, which handles global variables whose addresses are never taken (and therefore all accesses of the variables are trivially statically identifiable). We modified GlobalsModRef to ignore synchronization calls when called with a special flag. This alias analysis, although simple, is interprocedural and therefore was sufficient for testing our synchronization behavior analysis. In the future, we would like to combine the synchronization behavior analysis with a more sophisticated alias analysis that is sound for multithreaded programs (e.g., [50], which was not available for public use at the time of writing).

```
class MemDepResult {
  enum DepType { Def, Clobber, NonFuncLocal };
  DepType type;
  Instruction *I;
  ...
}

/* Get the set of dependent instructions for an instruction. */
vector<MemDepResult> getDependency(LoadInst *L);
```

Figure 6.4: Interface for LLVM's Memory Dependence Analysis.

6.4.2 LLVM Implementation

Figure 6.3 shows the current analysis and optimization pipeline in LLVM. Optimizations such as global value numbering, dead store elimination, and so on indirectly query the alias analysis through a helper analysis called the memory dependency ("MemDep") analysis. MemDepAnalysis queries alias information to determine the dependencies of a load instruction (or other instructions, but we consider only loads here), using the interface given in Figure 6.4. Dependencies consist of an instruction and a descriptor for the instruction: *Def* for instructions that definitely access the same memory location as the load, and *Clobber* for instructions that may change the value of the memory location. In other words, a load of location `x` may have a Clobber dependency on a store to a location that may-aliases with `x` or a function call that may write to `x`, and may have a Def dependency on a load or store to a location that must-aliases with `x`. If there are any paths from the beginning of the function to the queried load instruction with no dependencies, MemDepAnalysis returns a single *NonFuncLocal* dependency.

In this example, the load at line C is has a Def dependence on the load at line A and a Clobber dependence on the store at line B.

```
if (random()) {
A:  int r1 = *x;
} else {
  int *z = random() ? x : y;
B:  *z = 5;
```

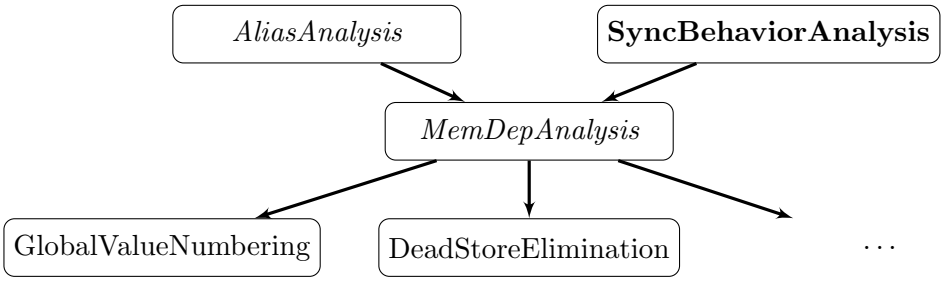

Figure 6.5: The LLVM analysis and optimization pipeline, modified to be synchronization-aware. Italicized analyses were modified; bolded analyses are new.

```
}
C: int r2 = *x;
```

Figure 6.5 shows our modified LLVM pipeline. Integrating the sync analysis into LLVM's optimization pipeline involved two main modifications. First, we modified the GlobalsMod-Ref analysis to ignore the side effects of synchronization calls. Second, we modified Mem-DepAnalysis to take advantage of synchronization behavior information and thereby avoid returning unnecessary Clobber results. Specifically, our modified version of MemDepAnal-ysis allows calls that perform synchronization on paths between a Def instruction and its dependent instruction, *as long as all release calls come after all acquire calls.* In such cases, the Def instruction's interference-free region will overlap with the dependent instruction's interference-free region, so no other thread can modify the variable without introducing a data race. On the other hand, if there is a release call followed by an acquire call between the Def instruction and the dependent instruction, the memory location is not interference-free between the Def and the dependent instruction, so another thread may modify its value. In this case, we return (one of) the release call(s) as a Clobber result.

Modifying MemDepAnalysis was quite straightforward. The algorithm is implemented as a backwards analysis from the dependent instruction. We added an extra piece of state to the analysis—a boolean indicating whether a call with behavior ACQ or ACQ/REL has been encountered yet on this path. If the analysis encounters a call with behavior REL or ACQ/REL when the boolean is true, it reports the release call as a Clobber dependency. It also reports any call with behavior ANY as a Clobber dependency.

The modified MemDepAnalysis replaces the ASLR and ABNA analyses from Chapter 4. We found that this approach was simpler and more effective, for the following three reasons:

1. The MemDepAnalysis approach catches more redundant loads that the ASLR/ABNA approach, because it is strictly more precise. Specifically, the MemDep analysis catches situations like this:

```
r1 = *x;
mtx_lock(...);
...
mtx_unlock(...);
if (...) {
  r2 = *x;
}
```

   Here the second load of `x` is redundant, since the call to `mtx_unlock` is interference-free for `x` if the conditional branch executes. However the ABNA analysis would not list `x` as interference-free for the `mtx_unlock()` call because `x` may not be read after the release call. MemDepAnalysis correctly determines that the first load of `x` is a Def for the second load, since the only path between the two loads does not have a release call followed by an acquire call.

2. MemDepAnalysis handles PHI nodes, which dictate how variable names should be translated at basic block boundaries. This is a tricky technical detail that can impact the precision of an analysis.

3. MemDepAnalysis includes a robust caching mechanism, which we were able to take advantage of with only a few small tweaks.

6.4.3 Experimental Results

Table 6.2 shows the result of running this analysis on the PARSEC and SPLASH-2 benchmarks. We used LLVM 2.9's clang frontend with link-time optimization in order to have the whole program available at analysis time. We disabled inlining in order to better evaluate the interprocedural aspect of our analysis. With inlining enabled, the interprocedural

	Package	Total static loads (LLVM bytecode)	Loads removed by GVN	Loads removed across user-defined function calls
SPLASH-2	barnes	460	4	0
	fmm	772	21	0
	ocean-c	3068	36	0
	ocean-n	985	94	0
	radiosity	1280	26	1
	raytrace	1357	10	0
	volrend	481	27	0
	water-n	706	15	0
	water-s	712	15	0
	cholesky	1975	9	0
	fft	197	3	0
	lu-c	219	3	0
	lu-n	170	3	0
	radix	202	11	0
PARSEC 2.1	blackscholes	42	0	0
	bodytrack	1949	0	0
	ferret	22398	0	0
	fluidanimate	398	7	0
	swaptions	129	1	0
	vips	58764	28	28
	x264	9616	0	0
	canneal	262	0	0
	dedup	1679	0	0
	streamcluster	270	10	0
	Total	108091	323	29

Table 6.2: Results of running LLVM's global value numbering (GVN) pass with the inter-procedural synchronization behavior analysis, a modified version of LLVM's GlobalsModRef alias analysis, and the modified version of LLVM's MemDepAnalysis on the SPLASH-2 and PARSEC benchmarks.

and intraprocedural versions of the analysis behave identically. This is partially due to the imprecise alias analysis, since the functions that were not inlined had poor thread-local side effect information.

The first column in Table 6.2 gives the package name. The second gives the total number of static loads per program, in order to put the last two columns in perspective. The third column gives the total number of new redundant loads identified and removed by GVN (global value numbering) using the synchronization-aware versions of the alias and memory dependence analyses. The fourth column gives the number of loads removed with at least one dependency across a user-defined function call—i.e., loads for which interprocedural synchronization behavior analysis is necessary.

Our results indicate that our analysis does uncover new loads to remove. As we already learned in Chapter 4, there are numerous loads that can be removed by enabling simply the intraprocedural analysis. As for the interprocedural analysis, there were two benchmarks, radiosity and vips, where loads were removed across user-defined function calls. For radiosity, the load in question was of a pointer to a global structure; the load was separated from its Def by a call to a function with ACQ/REL synchronization behavior. libxml2, a library called by vips, uses wrapper functions for synchronization, so our synchronization summaries allowed us to remove 28 extra loads across synchronization wrapper calls. We expect these results will improve with more accurate alias analysis.

6.5 Dynamic Data-Race Detection

This section explains how to extend IFRit, the dynamic data-race detection tool described in Chapter 5, to make use of the interprocedural synchronization behavior analysis described in Section 6.3.

As discussed in Section 6.1, interprocedural analysis can increase the size of inferred IFRs. On the data-race-detection side, the benefit of inferring bigger IFRs is that we are more likely to detect data races. Recall that we detect data races by detecting overlapping, conflicting IFRs in different threads. If the IFRs are bigger, then they are more likely to overlap. To use the "monitor" terminology from Section 5.2, monitors will be active for longer periods of time during execution, both because they are started earlier (instead

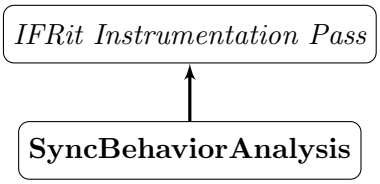

Figure 6.6: The IFRit instrumentation pipeline, modified to make use of the synchronization behavior analysis. Italicized analyses were modified; bolded analyses are new.

of conservatively started after unknown function calls) and because they may be able to propagate past more release calls before being stopped.

We will use the same synchronization behavior analysis described in Section 6.3, modulo the treatment of external function calls (see Section 6.5.2). Figure 6.6 shows the pass structure, which is very simple: the result of the synchronization behavior analysis feeds into the instrumentation pass (i.e., the static analysis).

6.5.1 Modified Static Analysis

Recall that the core of IFRit's static analysis are two backwards-dataflow analyses used to find the sets of variables (WBNA and ABNA) that must be accessed between each program point and either the next acquire call or the end of the function containing that program point.

In order to integrate the interprocedural synchronization analysis into IFRit, we modify how the two IFRit analyses handle call statements. In Figure 5.8, the parts of the analyses handling call statements are shown in the "Acquire," "Release," and "Call" lines. Specifically, the WBNA and ABNA sets propagate through release calls, and are killed at acquire and other calls. Figure 6.7 presents the modified dataflow analyses. This version is very similar, except that it relies on the synchronization behavior analysis. The WBNA and ABNA sets propagate through calls with synchronization behavior weaker than or equal to REL (i.e., REL or *syncnone*), since there is no chance that those calls will perform an acquire. For calls that may perform one or more acquire calls (i.e., *syncacq*, ACQ/REL, or ANY), the WBNA and ABNA sets are killed.

Statement type	Statement form	WBNA$[p]$	ABNA$[p]$
Load	p : r = *x;	WBNA$[p']$	ABNA$[p'] \cup \{x\}$
Store	p : *x = r;	WBNA$[p'] \cup \{x\}$	ABNA$[p'] \cup \{x\}$
Call	p : f(...); where sync(f) $\sqsupseteq$ ACQ	$\{\}$	$\{\}$
Call	p : f(...); where sync(f) $\sqsubseteq$ REL	WBNA$[p']$	ABNA$[p']$
Other	p : ...;	WBNA$[p']$	ABNA$[p']$

Figure 6.7: The dataflow analysis given in Figure 5.8, modified to use the results of the interprocedural synchronization behavior analysis. p' is the program point after the statement at point p.

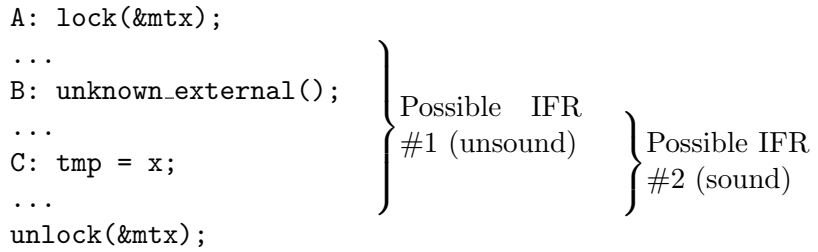

Figure 6.8: Possible inferred IFRs for a program with an unknown external function call. Possibility #1 assumes that `unknown_external` has synchronization behavior NONE. Possibility #2 assumes that `unknown_external` has synchronization behavior ANY.

6.5.2 Handling External Function Calls

In Section 6.3.2, we discussed how to handle external function calls in the context of compiler optimization. In that case, we argued that treating unknown function calls as having no synchronization was safe, because the complementary thread-local alias analysis would prevent unsafe optimizations across such calls. In the case of dynamic data-race detection, the correct approach is less clear, so we will simply present the two possible approaches and explain the advantages and disadvantages of each. As we will discuss in Section 6.5.3, the two approaches yielded no significant difference in our experiments.

Figure 6.8 illustrates the two possibilities, which depend upon the assumed synchronization behavior of the call to `unknown_external()` at line B. The IFR for access C extends

through call B if and only if call B performs an acquire call. If we assume that the call does not perform any synchronization (has synchronization behavior NONE), the instrumentation pass infers that access C's IFR extends up to line A (possible IFR #1 in the figure). (In other words, a `start_monitors` call will be inserted after line A.) If instead we assume that the call has synchronization behavior ANY, the inferred IFR is conservatively shortened to start after line B (possible IFR #2 in the figure). The second option is sound, but possibly overly conservative if B does not perform synchronization. The first option is unsound, but improves IFRit's chances of finding data races for the access.

The ANY approach is appropriate in the case where we use separate compilation to compile the program. In that case, it is possible that "external" calls are not library calls, but calls to user-defined functions in another file. Therefore the calls could contain relevant synchronization. In the opposite case where the whole program is available at compile time (e.g., link-time optimization), all external calls are library calls. As discussed in Section 5.2.4, we have observed that it is unlikely that user data will be protected by internal synchronization in library calls. Therefore the NONE approach, although unsound, seems reasonable when whole-program compilation is used.

We use whole-program compilation (link-time optimization) in our tests, so we chose to treat external calls as having behavior NONE. As expected, this did not introduce any false positives, because the library calls did not do any relevant synchronization. We also tested with the ANY approach and did not observe any significant difference between the two for the programs we tested.

6.5.3 Updated Results

We experimentally evaluated two versions of IFRit on Parsec 2.1. The *interprocedural* version used the results of the synchronization analysis for all function calls during the instrumentation pass. The *intraprocedural* version used the results of the synchronization behavior analysis only for calls to external (non-user-defined) functions, and assumed that user-defined functions had ANY behavior. In both cases, we compiled the benchmark programs using clang and link-time optimization so that the whole program would be available.

	IFRit (intraprocedural)	IFRit (interprocedural)	ThreadSanitizer
bodytrack	5	5	10
ferret	–	–	38
streamcluster	3	3	24
x264	3	2	72

Table 6.3: Number of unique races found by the intra- and interprocedural versions of IFRit on PARSEC. ThreadSanitizer results are repeated from Table 5.1 for comparison. Omitted benchmarks had no detected races.

As in Section 6.4, several of the benchmarks (dedup, facesim, raytrace, and vips[1]) did not compile properly in this environment. The baseline was also compiled using clang and link-time optimization, in order to better isolate the effects of the interprocedural synchronization analysis. For both versions of IFRit, we configured the runtime to start all short-scope monitors and not to use sampling.

Figure 6.9 gives the relative overheads for the two different versions of the runtime, as well as the old results from Chapter 5. The intraprocedural and interprocedural versions perform roughly the same on all benchmarks. There is no significant change from the numbers for IFRit in Chapter 5.

Table 6.3 shows the number of unique data races reported by the intraprocedural and interprocedural versions IFRit across 3 runs. The two versions reported the same races, except that the intraprocedural version reported one extra race in x264. There were only two dynamic reports of that extra race in x264, so most likely that race did not happen to manifest in the interprocedural runs.

These results are disappointing, since using the interprocedural synchronization behavior analysis does not noticeably affect the tool's performance or race-detection efficacy. A possible reason is that not enough functions have sufficiently simple synchronization behaviors (i.e., REL or NONE) to significantly change the results of the synchronization pass.

[1]vips compiled properly on its own, but failed to link properly with IFRit's runtime (see Section 5.5).

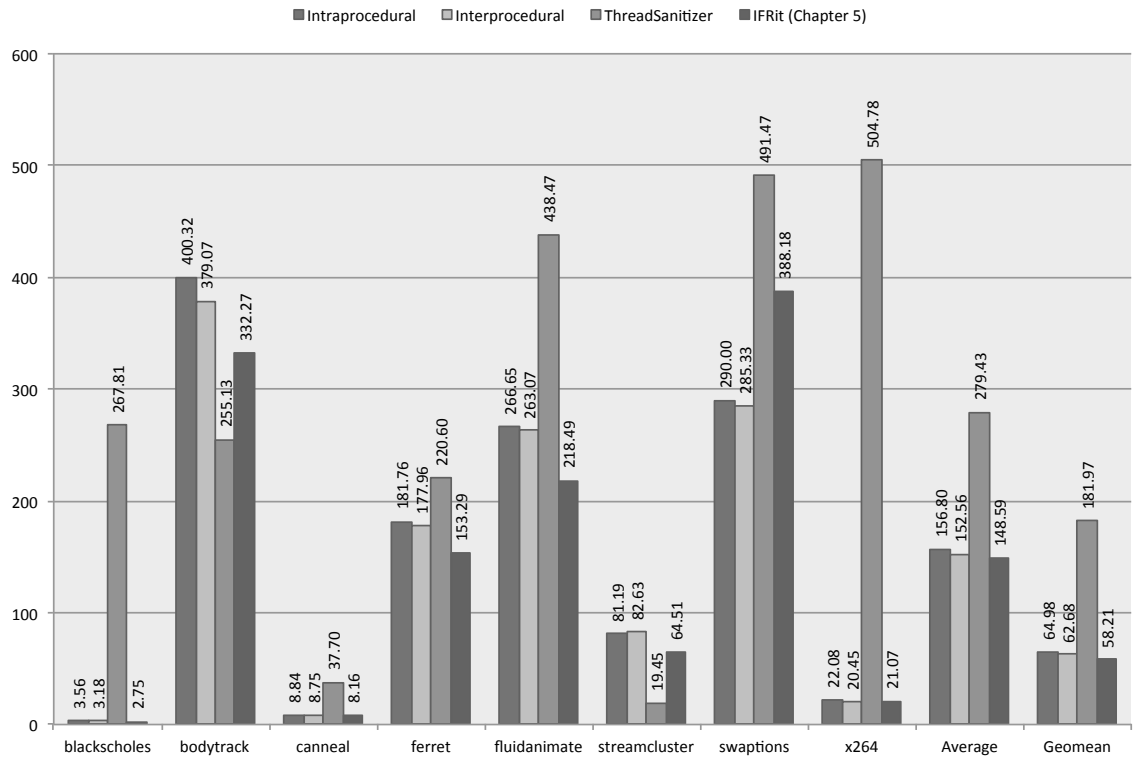

Figure 6.9: Updated overheads for PARSEC. ThreadSanitizer and IFRit results are repeated from Figure 5.10 for comparison. The average and geomean for ThreadSanitizer and the previous version of IFRit are for the benchmarks included in this figure only.

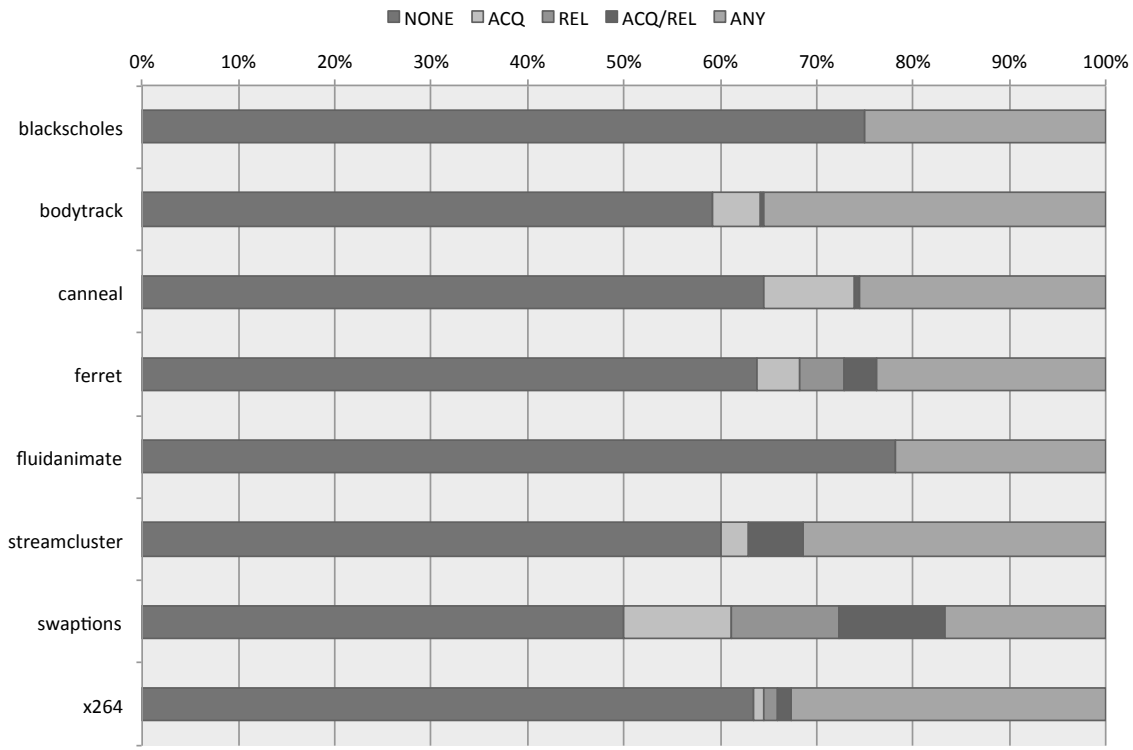

Figure 6.10: Distribution of synchronization behaviors for defined functions in the PARSEC benchmarks. This distribution is for the case where non-synchronization external functions are assumed to have synchronization behavior NONE.

However, Figure 6.10 gives the distribution of the five synchronization behaviors for defined (non-external) functions in the benchmarks, and for all benchmarks, at least half of the functions have synchronization behavior NONE. Therefore the instrumentation pass should be able to propagate WBNA and ABNA through many more function calls.

More likely is that, for the races in these benchmarks, the IFRs for the racy accesses include few, if any, calls to user-defined functions. One issue is the "short-scope monitors" discussed in Section 5.2.6—these monitors are inserted after the declaration of the accessed variable, meaning that the inferred IFR is already started as early as possible. Of the 11 races reported by the intraprocedural version of IFRit in bodytrack, streamcluster and x264, 7 involved a short-scope monitor, so the monitor's starting point could not be improved via interprocedural analysis. Manual inspection of all of the `start_monitors` calls involved in the 11 race reports confirmed that none of them could be moved earlier by reasoning about the synchronization behavior of calls. Our hope was to find new races for which the opposite was true, but for these seven benchmarks we were unsuccessful.

Chapter 7

CONCLUSIONS AND FUTURE WORK

Interference-free regions are a new abstraction for reasoning about data-race-freedom in multithreaded programs. This dissertation defines interference-free regions as the region around a memory access in a single thread's execution that extends from the most recent acquire call before the access and to the next release call after the access. Within this region, any modification of the accessed variable by another thread constitutes a data race. We argued that interference-free regions are useful by developing two novel applications of interference-free regions: compiler optimization and data-race detection.

The compiler optimization work, developed in Chapter 4 and published in MSPC 2011 [14], is one of the first projects to allow optimization across synchronization calls. In the future, we would like to see this kind of optimization integrated into real compilers. Chapter 6 discussed potentially extending this analysis to be interprocedural, but was hampered by the unavailability of sound interprocedural alias analyses for multithreaded programs. We plan to continue this work, either making use of stronger alias analyses as they become available, or developing new alias analyses specifically targeted to this problem.

In particular, we think it may be possible to build an alias analysis that ignores the effects of a synchronization call on a pointer if it is known that the call is interference-free for the pointer. This would be a new type of *synchronization-aware*, flow-sensitive alias analysis and would be useful both as a general multithreaded alias analysis and as a starting point for the interprocedural IFR analysis discussed in Chapter 6.

Our data-race detection tool (IFRit), described in Chapter 5 and published in OOPSLA 2012 [15], is also potentially extensible. Since this work does not assume data-race-freedom, it is not restricted to C and C++; in fact, we believe it would fit very well with other data-race detection work in Java [18]. The algorithm requires static instrumentation, so we are planning to use a framework like Soot [48] to instrument Java bytecode. We are

also interested in instrumenting binaries, which would fix some of the limitations of the prototype implementation in Chapter 5 (namely, not catching races on memory accesses in uninstrumented library code).

In summary, this work improves the community's understanding of multithreaded programs. We build on the newly-published C11 and C++11 standards [22, 23], particularly their requirement that programs be data-race-free, to develop several new algorithms based on the core idea of interference-free regions. As compilers begin to implement the new standards, the existence of a comprehensive and unambiguous specification for C/C++ multithreading will make it easier for researchers to develop new tools for writing, compiling, and debugging multithreaded programs. Our work on interference-free regions is an example of how reasoning formally about concurrency can lead to useful new algorithms for compilers and debugging tools.

BIBLIOGRAPHY

[1] Martin Abadi, Cormac Flanagan, and Stephen N. Freund. Types for safe locking: Static race detection for Java. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(2):207–255, March 2006.

[2] Sarita V Adve and Kourosh Gharachorloo. Shared memory consistency models: a tutorial. *IEEE Computer*, 29(12):66–76, December 1996.

[3] Sarita V Adve and Mark D Hill. Weak ordering—a new definition. In *ACM IEEE International Symposium on Computer Architecture (ISCA)*, 1990.

[4] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. *Compilers: Principles, Techniques, and Tools (Second Edition)*. Prentice Hall, 2011.

[5] David Aspinall and Jaroslav Ševčík. Formalising Java's data race free guarantee. In *International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, 2007.

[6] David Aspinall and Jaroslav Ševčík. Java memory model examples: good, bad and ugly. In *International Workshop on Verification and Analysis of Multi-threaded Java-like Programs (VAMP)*, 2007.

[7] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[8] Hans-J. Boehm. Reordering constraints for pthread-style locks. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2007.

[9] Hans-J. Boehm. Email to the Java Memory Model mailing list, May 2008. `https://mailman.cs.umd.edu/mailman/private/javamemorymodel-discussion/2008-May/000140.html`.

[10] Hans-J. Boehm. How to miscompile programs with "benign" data races. In *USENIX Workshop on Hot Topics in Parallelism (HotPar)*, 2011.

[11] Hans-J. Boehm and Sarita V. Adve. Foundations of the C++ concurrency memory model. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2008.

[12] Michael D. Bond, Katherine E. Coons, and Kathryn S. McKinley. Pacer: proportional detection of data races. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2010.

[13] Pietro Cenciarelli, Alexander Knapp, and Eleonora Sibilio. The Java memory model: operationally, denotationally, axiomatically. In *European Symposium on Programming (ESOP)*, 2007.

[14] Laura Effinger-Dean, Hans-J. Boehm, Dhruva Chakrabarti, and Pramod Joisha. Extended sequential reasoning for data-race-free programs. In *ACM SIGPLAN Workshop on Memory Systems Performance and Correctness (MSPC)*, 2011.

[15] Laura Effinger-Dean, Brandon Lucia, Luis Ceze, Dan Grossman, and Hans-J. Boehm. IFRit: Interference-free regions for dynamic data-race detection. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2012.

[16] Dawson Engler and Ken Ashcroft. RacerX: effective, static detection of race conditions and deadlocks. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[17] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. Effective data-race detection for the kernel. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[18] Cormac Flanagan and Stephen N Freund. FastTrack: efficient and precise dynamic race detection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[19] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. The Java Language Specification, Java SE 7 Edition, February 2012. `http://docs.oracle.com/javase/specs/jls/se7/jls7.pdf`.

[20] Ben Hardekopf and Calvin Lin. Semi-sparse flow-sensitive pointer analysis. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2009.

[21] Marieke Huisman and Gustavo Petri. The Java memory model: a formal explanation. In *International Workshop on Verification and Analysis of Multi-threaded Java-like Programs (VAMP)*, 2007.

[22] ISO JTC1/SC22/WG14. ISO/IEC 9899:2011, Information technology — Programming languages — C. `http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57853`. Draft available at `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf`.

[23] ISO JTC1/SC22/WG21. ISO/IEC 14882:2011, Information technology — Programming languages — C++. `http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372`. Draft available at `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3337.pdf`.

[24] Pramod Joisha, Robert S. Schreiber, Prithviraj Banerjee, Hans-J. Boehm, and Dhruva R. Chakrabarti. A technique for the effective and automatic reuse of classical compiler optimizations on multithreaded code. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2011.

[25] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, September 1979.

[26] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization (CGO)*, 2004.

[27] Chris Lattner and Vikram Adve. Automatic pool allocation: Improving performance by controlling data structure layout in the heap. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[28] Doug Lea. The JSR-133 cookbook for compiler writers. `http://gee.cs.oswego.edu/dl/jmm/cookbook.html`.

[29] Ondřej Lhoták and Kwok-Chiang Andrew Chung. Points-to analysis with efficient strong updates. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2011.

[30] Linux man page. `pthread_barrier_wait(3)`. `http://linux.die.net/man/3/pthread_barrier_wait`.

[31] Linux man page. `pthread_mutex_lock(3)`. `http://linux.die.net/man/3/pthread_mutex_lock`.

[32] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[33] Brandon Lucia, Luis Ceze, Karin Strauss, Shaz Qadeer, and Hans-J. Boehm. Conflict exceptions: Simplifying concurrent language semantics with precise hardware exceptions for data-races. In *ACM IEEE International Symposium on Computer Architecture (ISCA)*, 2010.

[34] Brandon Lucia, Benjamin P. Wood, and Luis Ceze. Isolating and understanding concurrency errors using reconstructed execution fragments. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011.

[35] Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2005.

[36] Daniel Marino, Madanlal Musuvathi, and Satish Narayanasamy. LiteRace: effective sampling for lightweight data-race detection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[37] Friedmann Mattern. Virtual time and global states of distributed systems. In *International Workshop on Parallel and Distributed Algorithms and Applications (PDAA)*, 1988.

[38] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for Java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[39] Marek Olszewski, Qin Zhao, David Koh, Jason Ansel, and Saman Amarasinghe. Aikido: accelerating shared data dynamic analyses. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[40] Eli Pozniansky and Assaf Schuster. MultiRace: Efficient on-the-fly data race detection in multithreaded C++ programs. *Concurrency and Computation: Practice and Experience*, 19(3):327–340, March 2007.

[41] Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus O Myreeen, and Jade Algave. The semantics of x86-CC multiprocessor machine code. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2009.

[42] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)*, 15(4):391–411, November 1997.

[43] Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitizer—data race detection in practice. In *Workshop on Binary Instrumentation and Applications*, 2009.

[44] Jaroslav Ševčík. *Program Transformations in Weak Memory Models*. PhD thesis, University of Edinburgh, 2008.

[45] Jaroslav Ševčík. Safe optimisations for shared-memory concurrent programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011.

[46] Jaroslav Ševčík and David Aspinall. On the validity of program transformations in the Java memory model. In *European Conference on Object-Oriented Programming (ECOOP)*, 2008.

[47] UPC Consortium. UPC language specications v1.2. Technical Report LBNL-59208, Lawrence Berkeley National Laboratory, May 2005.

[48] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot—a Java bytecode optimization framework. In *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, 1999.

[49] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *ACM IEEE International Symposium on Computer Architecture (ISCA)*, 1995.

[50] Jingyue Wu, Yang Tang, Gang Hu, Heming Cui, and Junfeng Yang. Sound and precise analysis of parallel programs through schedule specialization. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2012.

[51] Jie Yu and Satish Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. In *ACM IEEE International Symposium on Computer Architecture (ISCA)*, 2009.

[52] Yuan Yu, Tom Rodeheffer, and Wei Chen. RaceTrack: efficient detection of data race conditions via adaptive tracking. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2005.