IFRit: Interference-Free Regions for Dynamic Data-Race Detection *

Laura Effinger-Dean Brandon Lucia
Luis Ceze Dan Grossman
University of Washington
{effinger,blucia0a,luisceze,djg}@cs.washington.edu

Hans-J. Boehm
HP Labs
hans.boehm@hp.com

Abstract

We propose a new algorithm for dynamic data-race detection. Our algorithm reports no false positives and runs on arbitrary C and C++ code. Unlike previous algorithms, we do not have to instrument every memory access or track a full happens-before relation.

Our data-race detector, which we call *IFRit*, is based on a run-time abstraction called an *interference-free region* (IFR). An IFR is an interval of one thread's execution during which any write to a specific variable by a different thread is a data race. We insert instrumentation at compile time to monitor active IFRs at run-time. If the runtime observes overlapping IFRs for conflicting accesses to the same variable in two different threads, it reports a race. The static analysis aggregates information for multiple accesses to the same variable, avoiding the expense of having to instrument every memory access in the program.

We directly compare IFRit to FastTrack [10] and ThreadSanitizer [25], two state-of-the-art fully-precise data-race detectors. We show that IFRit imposes a fraction of the overhead of these detectors. We show that for the PARSEC benchmarks, and several real-world applications, IFRit finds many of the races detected by a fully-precise detector. We also demonstrate that sampling can further reduce IFRit's performance overhead without completely forfeiting precision.

* This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant DGE-0718124, the National Science Foundation under Grants CCF-1064497 and CCF-0811405, the University of Washington Royalty Research Fund, and gifts from Google.

OOPSLA'12, October 19–26, 2012, Tucson, Arizona, USA.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—parallel programming; D.2.5 [*Software Engineering*]: Testing and Debugging—monitors, testing tools

General Terms Languages, Algorithms, Reliability

Keywords data-race detection, concurrency, interference-free regions

1. Introduction

1.1 Motivation

Concurrent programming has become the norm, as multiprocessor computers require programmers to use parallelism to achieve high performance. In addition, many important application domains (*e.g.*, mobile computing, distributed sensing) are inherently concurrent. Unfortunately, concurrent programming is difficult, and concurrent programming errors are common.

A data race is a pathological concurrent program behavior that is often the result of a programming error. A data race occurs when multiple threads access the same memory location, at least one access is a write, and the accesses are not ordered by synchronization. Determining the meaning of a program's execution when a data race has occurred is deeply problematic. In C and C++, programs that permit data races have undefined semantics [2]—an execution can literally do anything. Many managed languages like Java have memory models that provide somewhat stronger guarantees about what a program with data races is allowed to do. Memory models that provide such strong guarantees and also permit efficient language implementations are complex and subtle. As a result, data races should almost always be avoided [2, 26].

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. 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 [10, 21] than uninstrumented execution, so it is typically useful to sacrifice precision in principled ways while still detecting many data races.

1.2 Our Approach

This paper describes *IFRit*, a dynamic data-race detector that never reports false data races but may miss data races.[1] 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:[2]

```
lock(m);
for(int i = 0; i < 1000; i++) {
  ...
  *x = ... ;
  ...
}
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 lock and 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 our recent work on interference-free regions [7], as explained in more detail in Section 2. Consider this example, where again we assume any code not shown is known to be synchronization-free:

```
*x = ...;
while(...) {
  *x = ...;
  lock(m);
  ...
  unlock(m);
}
*x = ...;
```

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

[2] Here, we assume that `x` is a local variable or register, and the location pointed to by `x` is a shared variable. Henceforth when we say "accessing `x`" or "writing to `x`," we are referring to the location pointed to by `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. It is surprising that doing so cannot lead to reporting data races that are not true data races, since the code above does have synchronization. But 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`.

1.3 Results

Our work is the first to use the notion of IFRs for data-race detection. We have implemented our technique in the LLVM compiler framework [14] 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 also compare our results with published results for other imprecise data-race detectors. We have developed a formal model of IFRs for data-race detection and use it prove correctness: Any data race reported by our approach is a true data race.

The remainder of this paper is organized as follows. Section 2 discusses the details of the IFR abstraction and explains our new technique for applying IFRs to data-race detection. Section 3 presents the static analysis used by IFRit to insert instrumentation calls. Section 4 discusses the runtime implementation. Section 5 formalizes IFRs and proves that our technique is sound. Section 6 gives empirical data illustrating IFRit's low overhead and high precision. Section 7 discusses prior work related to IFRit in more detail, and Section 8 concludes and lays out possible future work.

2. Interference-Free Regions

This section introduces interference-free regions and applies them to dynamic data-race detection.

2.1 Background

An interference-free region, or *IFR*, is a region of a single thread's execution trace surrounding an access to a shared variable [7]. The region extends from the first acquire action

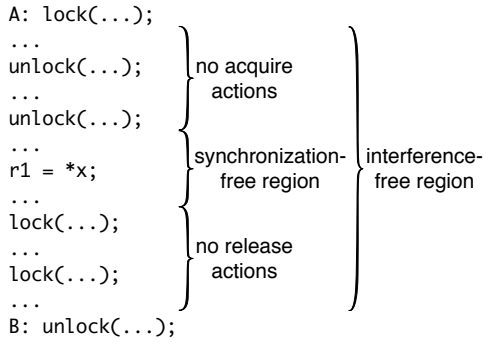


Figure 1. Interference-free region for an access.

prior to the access, to the first release action following the access, noninclusive. By an *acquire action*, we mean an action that is synchronized with earlier actions in the trace; for example, a lock acquire is an acquire action because earlier releases of the lock synchronize with the acquire. Similarly, a *release action* synchronizes with later actions in the trace. These "synchronizes-with" relationships, combined with the ordering on actions of each thread induced by program order, determine the *happens-before* order, a partial order over actions in the execution [13]. Other acquire actions include thread joins and reads of Java volatile variables or C11 atomic variables. Other release actions include thread create actions and writes to Java volatile variables or C11 atomic variables.

Note that IFRs are purely dynamic constructs, and therefore, for each memory access in an execution trace, there is exactly one IFR for that access, and all operations by the same thread either do or do not fall into the IFR for that access.

For example, Figure 1 shows the IFR for an access to shared variable `x`. The figure shows only the actions for a single trace. The IFR extends from the lock at line A to the unlock at line B. Note that the IFR is always at least as large as the access's *synchronization-free region*, which extends from the most recent synchronization action to the next synchronization action. We can characterize the IFR as the union of three regions: the synchronization-free region surrounding the access, the "acquire-free" region before the synchronization-free region, and the "release-free" region after the synchronization-free region.

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.

Interference-free regions are so called because while the thread is executing the interference-free region, no other thread can write to the shared variable accessed by the IFR's access without inducing a *data race*. A data race is a pair of accesses to the same variable by different threads, where at least one access is a write and the accesses are not ordered by synchronization. Therefore, if an execution has no data races, the variable (e.g., `x` in Figure 1) is "interference-free" for the duration of the execution of the region.

Prior work [7] used IFRs for compiler optimization. Compilers typically perform optimizations only within synchronization-free regions. Since the compiler may assume data-race-freedom for C and C++ programs [11, 12], IFRs extend the scope in which the compiler can reason sequentially about a variable's value. Our work takes the fundamental idea of interference-free regions and applies it to another purpose: dynamic data-race detection.

2.2 IFRs for Data-Race Detection

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.[3] 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 2. 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 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 3(a) 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 3(b), the two threads use different mutexes to protect variable `x`, but the critical 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

[3] Our implementation inserts instrumentation that serializes the beginnings and ends of IFRs for the same variable. See Section 4.3 for more details.

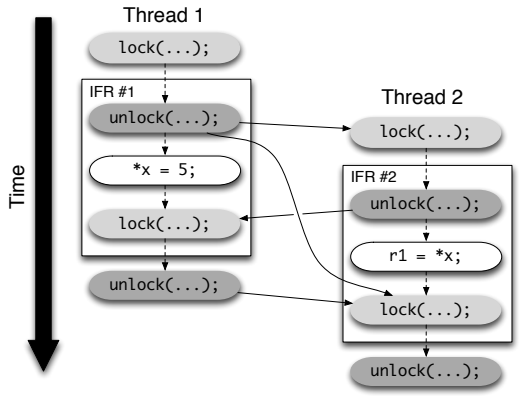
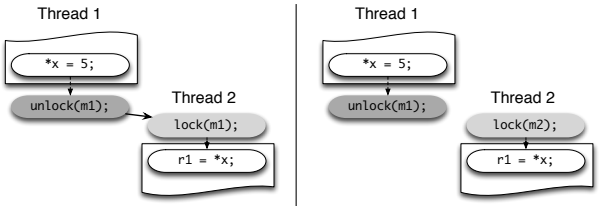


Figure 2. Overlapping IFRs for racy accesses in an execution. The solid blocks indicate interference-free regions. Dashed lines indicate program order; solid lines indicate possible happens-before edges between synchronization actions. The two accesses must form a data race.



(a) These two accesses do not form a data race, so their IFRs do not overlap.

(b) These two accesses form a data race, even though their IFRs do not overlap.

Figure 3. Non-overlapping IFRs.

in the program. This property follows from a standard theorem about happens-before memory models (e.g., Theorem 8.2 in [5]): 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.[4] 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 [24], 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 6).

[4] An execution is sequentially consistent if all threads see the same global order of actions, the actions of each thread are in program order, and every read sees the most recent write to the same location.

2.3 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 (SFRs) instead of overlapping interference-free regions. We believe that a scheme using interference-free regions is superior for two reasons. First, as shown in Figure 1, the interference-free region for an access always subsumes the synchronization-free region for an access, so monitoring interference-free regions will 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 this paper uses synchronization-free regions to implement a lightweight hardware concurrency exception model [16]. 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.[5] 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.

3. 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 3.1) and giving a simple correctness criterion for the analysis (Section 3.2). Then we present the algorithm in two steps. First, we describe a simplified algorithm for inserting instrumentation calls (Section 3.3). Second, we present the refined algorithm actually used in our implementation (Section 3.4).

[5] For an example of why this might happen, see Section 3.6.

The simplified version is a useful starting point, and the refined algorithm derives directly from the ideas discussed in Section 3.3. Section 3.5 details the dataflow analysis we use to implement the algorithm, and Section 3.6 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.

3.1 Instrumentation

The static analysis inserts calls to the run-time to start and stop *monitors* for different variables.[6] 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 consider a simple instrumentation scheme in which we start and stop monitors for a single variable via three instrumentation calls:

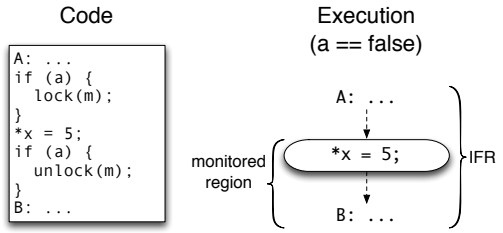
```
start_strong_monitor(void *x);
start_weak_monitor(void *x);
stop_monitor(void *x);
```

`start_strong_monitor` and `start_weak_monitor` have no effect if a monitor of the correct type is already active for the argument. `stop_monitor` has no effect if no monitor is active.

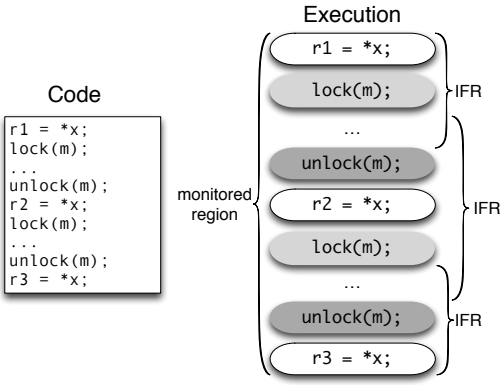
For example, a simple critical section could be instrumented as follows:

```
lock(m);
start_strong_monitor(x);
...
*x = ...;
...
stop_monitor(x);
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 4(a), our analysis does not insert the instrumentation to start the monitor until after the `if` statement.[7] 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 4(b), the program may read `x` one or more times, but we only need to call `start_weak_monitor` once.



(a) Monitored regions may be smaller than the actual IFR, due to conservatism in the static analysis.



(b) Monitored regions may combine IFRs for several accesses to the same variable.

Figure 4. Two examples of IFRs vs. monitors in execution.

3.2 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:

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`.

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

[7] Placing a call to `start_strong_monitor` before the `if` statement would violate the first correctness criterion in Section 3.2.

3.3 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. $\mathcal{W}[p]$: 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. $\mathcal{RW}[p]$: 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).

At each program point p, $\mathcal{W}[p]$ 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, $\mathcal{RW}[p]$ 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 $\mathcal{A}[p]$, the set of variables for which there exists a path from an access to the variable to p, with no intervening release calls. 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 $\mathcal{A}[p]$, we will have satisfied the correctness conditions in Section 3.2.

However, inserting calls for every variable in $\mathcal{A}[p]$ 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(a) 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 $\mathcal{A}[p] - \mathcal{RW}[p]$ (i.e., variables for which this program point does not fall in an IFR). For variables in $\mathcal{A}[p] \cap \mathcal{W}[p]$ (i.e., variables for which this program point falls in a write IFR), we do not need to insert any instrumentation. For variables in $\mathcal{A}[p] \cap (\mathcal{RW}[p] - \mathcal{W}[p])$ (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:

```
lock(m1);
start_strong_monitor(x);
...
*x = ...;
...
stop_monitor(x); // too early
unlock(m1);
...
*x = ...;
...
lock(m2);
...
stop_monitor(x);
unlock(m2);
```

(a) Stopping a monitor too early.

```
lock(m1);
start_strong_monitor(x);
...
*x = ...;
...
downgrade_monitor(x);
unlock(m1);
...
r1 = *x;
...
lock(m2);
...
stop_monitor(x);
unlock(m2);
```

(b) Downgrading a monitor from strong to weak.

Figure 5. Stopping and downgrading monitors.

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

For example, in Figure 5(b) 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 3.2:

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 $\mathcal{W}[p]$.
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 $\mathcal{RW}[p] - \mathcal{W}[p]$.
3. At release calls, we insert a call to `stop_monitor` for variables in $\mathcal{A}[p] - \mathcal{RW}[p]$.
4. At release calls, we insert a call to `downgrade_monitor` for variables in $\mathcal{A}[p] \cap (\mathcal{RW}[p] - \mathcal{W}[p])$.
5. At unknown function calls and the end of the function, we insert calls to `stop_monitor` for all variables in $\mathcal{A}[p]$.

3.4 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., $\mathcal{RW}[p] - \mathcal{W}[p]$), and the next `num_strong` arguments are the strong monitors

to start ($\mathcal{W}[p]$). 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 3.3. 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 3.3. 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 $\mathcal{W}[p]$ and $\mathcal{RW}[p] - \mathcal{W}[p]$, 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.

3.5 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: $\mathcal{W}$ and $\mathcal{RW}$. (The refined analysis does not use $\mathcal{A}$.) The initial values (at the end of the function) are $\mathcal{W}[p_{\mathsf{end}}] = \mathcal{RW}[p_{\mathsf{end}}] = \{\}$. The sets propagate through statements as shown in Figure 6. At load and store operations we update the $\mathcal{W}$ and $\mathcal{RW}$ 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 [14].

Statement type	Statement form	$\mathcal{W}[p]$	$\mathcal{RW}[p]$
Load	p : `r` = *`x`;	$\mathcal{W}[p']$	$\mathcal{RW}[p'] \cup \{x\}$
Store	p : *`x` = `r`;	$\mathcal{W}[p'] \cup \{x\}$	$\mathcal{RW}[p'] \cup \{x\}$
Acquire	p : `lock(m)`;	$\{\}$	$\{\}$
Release	p : `unlock(m)`;	$\mathcal{W}[p']$	$\mathcal{RW}[p']$
Call	p : `f(...)`;	$\{\}$	$\{\}$
Other	p : ...;	$\mathcal{W}[p']$	$\mathcal{RW}[p']$

Figure 6. Summary of our backwards data-flow analysis to insert instrumentation calls. p' is the program point after the statement at point p.

3.6 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 that 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 4.4.

4. 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.

4.1 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 [10], 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.

4.2 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.

4.3 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.[8] 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.

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

[8] The results presented in Section 6 use $n = 5$.

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`).

4.4 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 3.6, 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. 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 `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 went down by 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 `pthread_create` to start a new thread. `pthread_create` is a release operation. Therefore any monitors collected during the single-threaded phase would be stopped before the `pthread_create` call anyway, so there is no point to starting these monitors.[9]

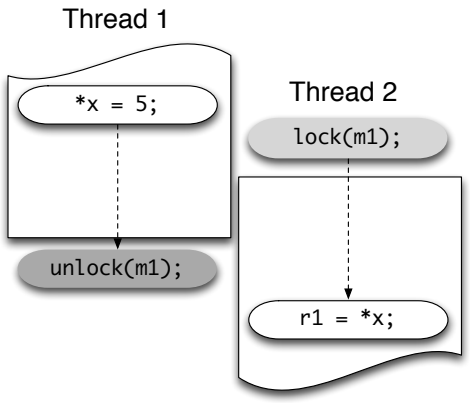


Figure 7. Even though neither access happens during the other access's IFR, we can detect the race in this case because the accesses' IFRs overlap.

5. 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 proof is based on a proof in prior work, which showed that other threads could not write to a variable during an IFR for that variable without inducing a data race [7]. The property we prove here is stronger, because the racing access may not happen during the other access's IFR (see Figure 7 for an example). We use similar notation to prior work to improve clarity. To simplify the presentation, we use a version of the C++11/C11 memory model that has been abstracted and simplified in unessential ways [11, 12].

An execution of a program is a quadruple $(A, \leq_{\mathsf{sb}}, <_{\mathsf{sw}}, \leq_{\mathsf{hb}})$. A is a set of actions, where each action a is a triple of a thread ID t, a kind of action k, and a unique ID u: $a = (t, k, u)$. We do not specify which kinds of actions are used in the execution, but we assume there are reads and writes of variables, as well as some form of synchronization. The sequenced-before relation $\leq_{\mathsf{sb}}$ totally orders unique IDs with the same thread ID. The synchronizes-with relation $<_{\mathsf{sw}}$ is a strict partial order over unique IDs, which orders synchronization actions: $u_1 <_{\mathsf{sw}} u_2$ implies that u_1 is the UID for a release action, and u_2 is the UID for an acquire action. The happens-before relation $\leq_{\mathsf{hb}}$ is the reflexive transitive closure of the union of $\leq_{\mathsf{sb}}$ and $<_{\mathsf{sw}}$: $\leq_{\mathsf{hb}} = (\leq_{\mathsf{sb}} \cup <_{\mathsf{sw}})^*$.

Our goal is to prove that two overlapping IFRs for conflicting, concurrent accesses to the same variable always im-

[9] The `pthread_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 `pthread_create` call.

ply that the accesses form a data race. This is stated in the following theorem:

Theorem 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 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. *$u^{\text{begin}} <_{\mathsf{sb}} u^{\text{access}} <_{\mathsf{sb}} u^{\text{end}}$.*
3. *For all u such that $u^{\text{begin}} <_{\mathsf{sb}} u \leq_{\mathsf{sb}} u^{\text{access}}$, u's associated kind is not an acquire synchronization.*
4. *For all u such that $u^{\text{access}} \leq_{\mathsf{sb}} u \leq_{\mathsf{sb}} u^{\text{end}}$, u's associated kind is not a release synchronization.*

Definition 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:*

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 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 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 $u_1 \leq_{\mathsf{sb}} u_3 \leq_{\mathsf{hb}} u_4 \leq_{\mathsf{sb}} u_2$, u_3's associated kind is a release synchronization, and u_4's associated action is an acquire synchronization.*

The proof of Lemma 1 is given in Appendix A . Lemma 1 leads directly to the proof of Theorem 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 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 $u_1^{\text{access}} \leq_{\mathsf{sb}} u_3 \leq_{\mathsf{hb}} u_4 \leq_{\mathsf{sb}} u_2^{\text{access}}$, u_3 is a release synchronization, and u_4 is an acquire synchronization. By Definition 1, it must be the case that the release and acquire actions do not fall in I_1 and I_2, respectively: $u_1^{\text{end}} \leq_{\mathsf{sb}} u_3$ and $u_4 \leq_{\mathsf{sb}} u_2^{\text{begin}}$. 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.

6. 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 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 [25] and FastTrack, a state-of-the-art happens-before data-race detection tool [10, 21].

This section focuses on comparison with precise data-race detectors; in Section 7 we discus other imprecise approaches, such as detectors that implement sophisticated sampling techniques [6, 18] or detectors that use hardware watchpoints [9].

6.1 Experimental Setup

To benchmark IFRit, we used the PARSEC-2.1 benchmark suite [3] and a set of real applications. PARSEC is comprised of a set of programs representative of emerging concurrent applications, such as data mining, vision, and video encoding. 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

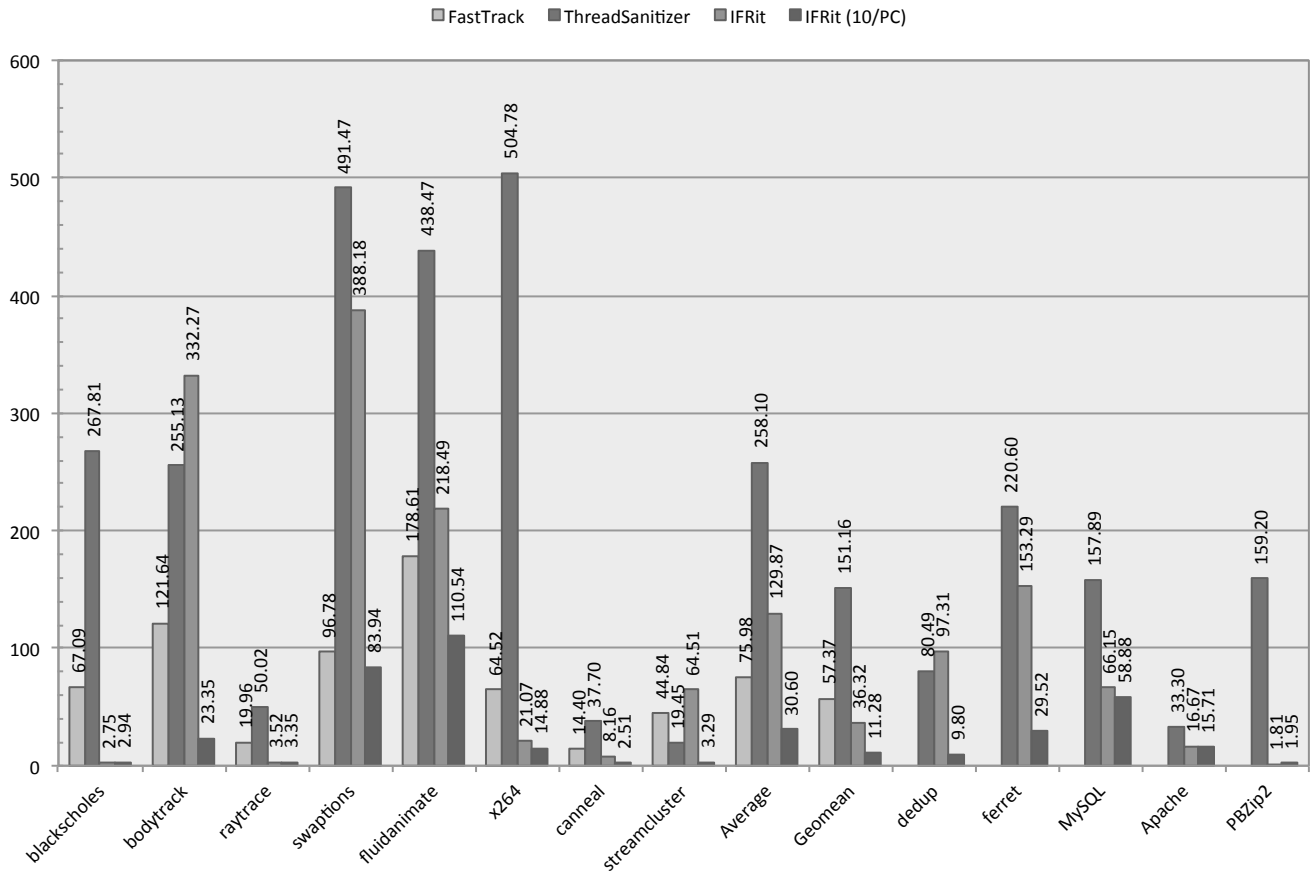


Figure 8. Overhead of IFRit compared to uninstrumented code for the PARSEC benchmarks and a suite of real applications. Average and geometric mean are over the first eight PARSEC benchmarks.

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 and our instrumenting compiler pass. For our baseline, we compiled applications using LLVM, but without our instrumenting pass. The PARSEC benchmarks were run on a dual 4-core 2.27 GHx Intel Xeon E5562 with 10GB of RAM. The real applications were run on a 4-core 2.8 GHz Intel Xeon E5462 with 16GB of RAM.

In our evaluation we directly compared IFRit to ThreadSanitizer's Valgrind implementation [25] and an implementation of FastTrack for C/C++ using DynamoRio [21]. We ran experiments with ThreadSanitizer on our machines. The authors of [21] provided us with data from their experiments with their FastTrack implementation; they used a quad-socket, 8-core 2.0 GHz Intel Xeon X7550 for their experiments.

6.2 Overheads

PARSEC Figure 8 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

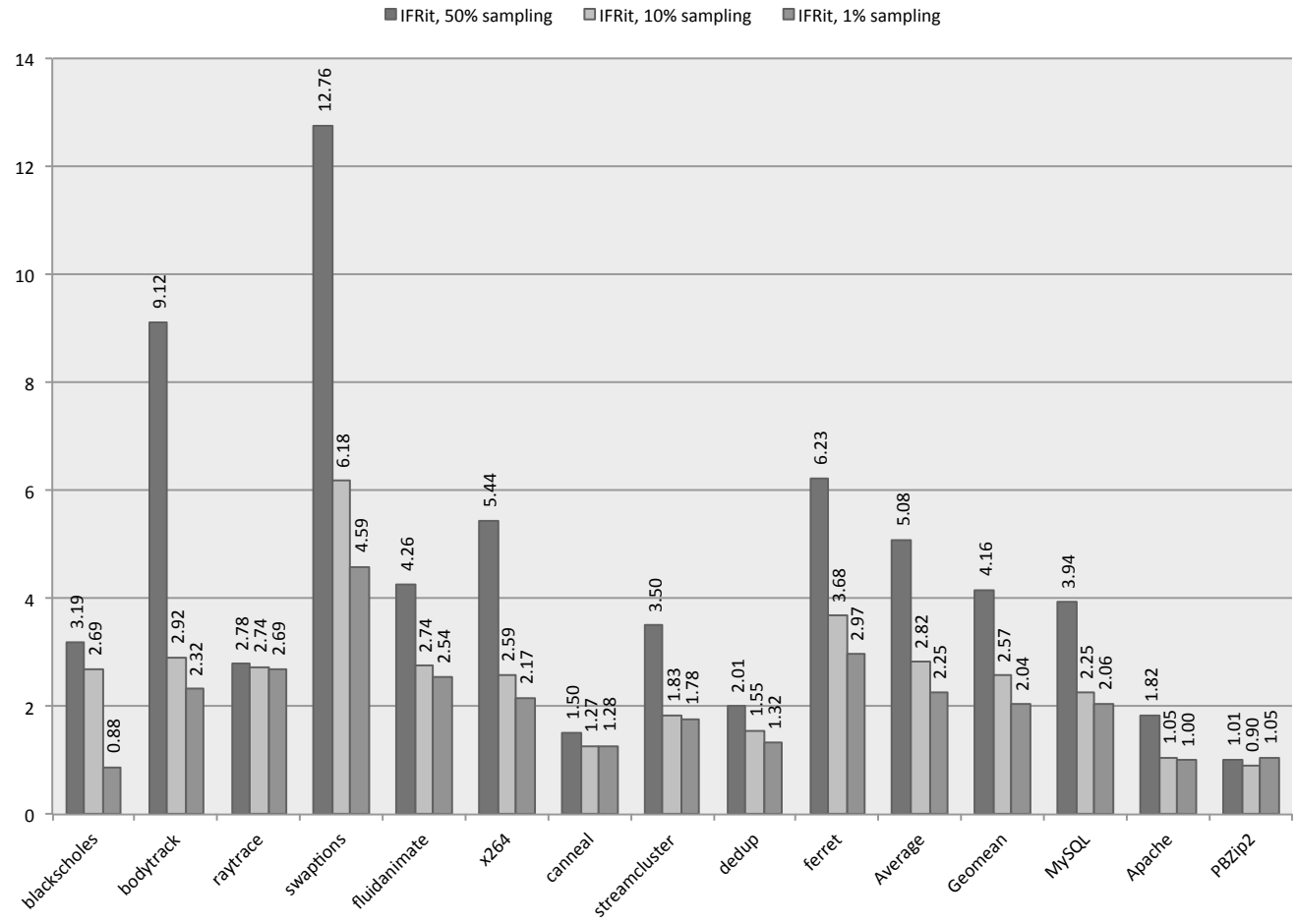


Figure 9. Effect of sampling on IFRit's performance overhead. Average and geometric mean are over the ten PARSEC benchmarks.

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.

Several of the benchmarks (blackscholes, raytrace, x264, canneal) have especially low overheads. The main reason for these low overheads is that the structure of the parallelism in these programs amortizes the analysis cost imposed when monitors start and end. Blackscholes has fork-join structure, so it does little sharing and few monitor starts and ends, relative to the amount of computation being performed by the program.

Bodytrack, swaptions, fluidanimate, streamcluster, dedup and ferret saw higher overheads. In these cases, more frequent starting and stopping of monitors leads to a larger fraction of the execution time spent running IFRit's instrumentation code. These applications also tend to have many short-scope monitors. For example, in streamcluster threads repeatedly iterate over arrays of points, coordinated via barrier synchronization calls. This results in many short-scope monitors, since most memory accesses happen within tight loops, as well as many calls to `stop_all_monitors_except`, since there are so many calls to `pthread_barrier_wait`. Note that streamcluster's performance greatly improves when we cap the number of short-scope monitors.

Overall, these data show that IFRit's overheads are comparable to prior race detection tools [10, 21, 25]. 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 6.3).

Real applications Figure 8 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. Like dedup and blackscholes, PBZip has low overhead because of its parallelism strategy. In PBZip workers share data with a main thread through a worklist, but synchronize infrequently. The infrequent starting and stopping of monitors that results leads to PBZip's low overhead. Our experiments with PBZip show how avoiding per-memory-operation overhead saves performance. In FastTrack and ThreadSanitizer, each data access is instrumented to check or update a vector clock. In contrast, IFRit only needs to update its state at the beginning of large regions during which many data accesses are performed.

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.

When we apply our short-scope monitor optimization MySQL's overhead is reduced 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.

6.2.1 Impact of Sampling on Performance

Figure 9 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 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.

6.3 Race-Detection Coverage

Table 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 6.4. 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 Section 6.4, 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.

	IFRit					
Races	1%	10%	50%	10/PC	Full	TS
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 1. Number of unique races found by IFRit in various configurations. TS shows races reported by ThreadSanitizer. Omitted benchmarks had no detected races.

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.

6.3.1 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 8, the data show that the reduction in overhead resulting from this optimization is very large. The data in Table 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.

6.3.2 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 9 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 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.

6.4 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.

6.4.1 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 which 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 which 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.[10]

ThreadSanitizer reported 23 unique races in streamcluster, including the three reported by IFRit. We determined that the remaining races reported by ThreadSanitizer boiled down 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.

6.4.2 Races in Real Applications

MySQL IFRit reported 11 races in MySQL. Three races in MySQL were the result of unsynchronized accesses to termination 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 [15, 17, 27]. 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 [4]. 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.

6.5 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

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

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 8 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 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.

Another important distinction between IFRit and both prior detectors is IFRit's ability to sample program execution. Sampling gives developers a knob to turn that scales back the overhead of race detection. FastTrack and ThreadSanitizer do not provide such a knob. IFRit's overheads with sampling enabled are sometimes less than 2X. Furthermore, as we describe in Section 6.3 IFRit still detects many races detected by the precise detectors with sampling enabled. IFRit's soundness guarantees, combined with such low overheads make it practical to integrate race detection with a development framework like continuous testing [23]. FastTrack's and ThreadSanitizer's overheads are likely to be too high for continuous use. In some cases (*e.g.*, Apache, PBZip, canneal, blackscholes) IFRit's overheads are low enough that they would be tolerable for use in deployed systems.

7. Related Work

A variety of tools have been developed to help find data races. Static race detection tools [1, 8, 20] 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 [6, 10, 18, 21, 25] and lockset detectors [24]. Lockset detectors like Eraser [24] 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 [22], RaceTrack [28], and ThreadSanitizer's hybrid mode [25].

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* [19], 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 6) is a standard happens-before detector that uses valgrind to instrument binaries.

The current state-of-the-art implementation of vector clocks, FastTrack [10], 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++).

As dicussed in Section 6, a recent paper reimplemented FastTrack for x86 binaries using the DynamoRio instrumentation platform [21]. 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 [6]. 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 [18] 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 [9] 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.

8. Conclusion & Future Work

We have presented IFRit, a new dynamic data-race detection algorithm for arbitrary C and C++ programs. IFRit improves on prior work by coalescing the instrumentation for multiple accesses to the same variable, reducing runtime overhead. We require no specialized hardware and 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.

A possible future improvement would be to improve our strategy for short-scope monitors. Our current scheme simply limits the number of short-scope monitors per static call site; a more adaptive strategy (say, starting a monitor with probability inversely proportional to the number of active monitors at that call site) would be more thorough.

We plan to extend our static analysis to be interprocedural. Currently, we treat function calls conservatively, sometimes starting monitors later than necessary. Interprocedural analysis allows us to propagate information through function calls, increasing the lengths of monitored regions, and finding data races. We also plan to implement IFRit for Java programs.

Acknowledgments

Thanks to the anonymous reviewers and to Mark Oskin for their valuable comments on earlier drafts of this paper.

References

[1] M. Abadi, C. Flanagan, and S. 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] S. V. Adve and H.-J. Boehm. Memory models: A case for rethinking parallel languages and hardware. *Communications of the ACM (CACM)*, 53(8):90–101, August 2010.

[3] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

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

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

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

[7] L. Effinger-Dean, H.-J. Boehm, P. Joisha, and D. Chakrabarti. Extended sequential reasoning for data-race-free programs. In *ACM SIGPLAN Workshop on Memory Systems Performance and Correctness (MSPC)*, 2011.

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

[9] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk. Effective data-race detection for the kernel. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

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

[11] 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`.

[12] 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.

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

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

[15] S. Lu, S. Park, E. Seo, and Y. 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.

[16] B. Lucia, L. Ceze, K. Strauss, S. Qadeer, and H.-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.

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

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

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

[20] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[21] M. Olszewski, Q. Zhao, D. Koh, J. Ansel, and S. Amarasinghe. Aikido: accelerating shared data dynamic analyses. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[22] E. Pozniansky and A. 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.

[23] D. Saff and M. D. Ernst. Reducing wasted development time via continuous testing. In *IEEE International Symposium on Software Reliability Engineering (ISSRE)*, 2003.

[24] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)*, 15(4):391–411, November 1997.

[25] K. Serebryany and T. Iskhodzhanov. ThreadSanitizer—data race detection in practice. In *Workshop on Binary Instrumentation and Applications*, 2009.

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

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

[28] Y. Yu, T. Rodeheffer, and W. Chen. RaceTrack: efficient detection of data race conditions via adaptive tracking. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2005.

A. Proof of Lemma 1

Lemma 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 $u_1 \leq_{\mathsf{sb}} u_3 \leq_{\mathsf{hb}} u_4 \leq_{\mathsf{sb}} u_2$, u_3's associated kind is a release synchronization, and u_4's associated action is an acquire synchronization.*

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

- As $t_1 \neq t_2$, $u_1 \not\leq_{\mathsf{sb}} u_2$.
- If $u_1 <_{\mathsf{sw}} u_2$. By the definition of $<_{\mathsf{sw}}$, u_1 is a release synchronization and u_2 is an acquire synchronization. Let $u_3 = u_1$ and $u_4 = u_2$. By reflexivity of $\leq_{\mathsf{sb}}$, we have that $u_1 \leq_{\mathsf{sb}} u_3 \leq_{\mathsf{hb}} u_4 \leq_{\mathsf{sb}} u_2$.
- 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 $u_5 \leq_{\mathsf{sb}} u_6 \leq_{\mathsf{hb}} u_7 \leq_{\mathsf{sb}} u_4$, u_6 is a release synchronization, and u_7 is an acquire synchronization. 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 $u_1 \leq_{\mathsf{sb}} u_5$, and by transitivity of $\leq_{\mathsf{sb}}$, $u_1 \leq_{\mathsf{sb}} u_6$. Therefore $u_1 \leq_{\mathsf{sb}} u_6 \leq_{\mathsf{hb}} u_7 \leq_{\mathsf{sb}} u_2$.
  - $t_5 = t_2$. Then $t_5 \neq t_1$, so by the inductive hypothesis there exist u_6 and u_7 such that $u_1 \leq_{\mathsf{sb}} u_6 \leq_{\mathsf{hb}} u_7 \leq_{\mathsf{sb}} u_5$, u_6 is a release synchronization, and u_7 is an acquire synchronization. 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 $u_5 \leq_{\mathsf{sb}} u_2$, and by transitivity of $\leq_{\mathsf{sb}}$, $u_7 \leq_{\mathsf{sb}} u_2$. Therefore $u_1 \leq_{\mathsf{sb}} u_6 \leq_{\mathsf{hb}} u_7 \leq_{\mathsf{sb}} u_2$.
  - $t_5 \neq t_1$ and $t_5 \neq t_2$. We apply the inductive hypothesis twice. First, there exist u_6 and u_7 such that $u_1 \leq_{\mathsf{sb}} u_6 \leq_{\mathsf{hb}} u_7 \leq_{\mathsf{sb}} u_5$ and u_6 is a release synchronization. Second, there exist u_8 and u_9 such that $u_5 \leq_{\mathsf{sb}} u_8 \leq_{\mathsf{hb}} u_9 \leq_{\mathsf{sb}} u_2$ and u_9 is an acquire synchronization. Let $u_3 = u_6$ and $u_4 = u_9$. By transitivity of $\leq_{\mathsf{hb}}$, we have that $u_1 \leq_{\mathsf{sb}} u_6 \leq_{\mathsf{hb}} u_9 \leq_{\mathsf{sb}} u_4$. □