Precise and Maximal Race Detection from Incomplete Traces

Jeff Huang
Texas A&M University, USA
jeff@cse.tamu.edu

Arun K. Rajagopalan
Texas A&M University, USA
arunxls@tamu.edu

Abstract

We present RDIT, a novel dynamic technique to detect data races in multithreaded programs with incomplete trace information, *i.e.*, in the presence of missing events. RDIT is both *precise* and *maximal*: it does not report any false alarms and it detects a maximal set of true traces from the observed incomplete trace. RDIT is underpinned by a sound Barrier-Pair model that abstracts away the missing events by capturing the invocation data of their enclosing methods. By making the least conservative abstraction that a missing method introduces synchronization only when it has a memory address in scope that overlaps with other events or other missing methods, and by formulating maximal thread causality as logical constraints, RDIT guarantees to precisely detect races with maximal capability. RDIT has been applied in seven real-world large concurrent systems and has detected dozens of true races with zero false alarms. Comparatively, existing algorithms such as Happens-Before, Causal-Precedes, and Maximal-Causality which are known to be precise all report many false alarms when missing synchronizations.

Categories and Subject Descriptors: D.2.5 **[Software Engineering]**: Testing and Debugging - *Diagnostics; Debugging aids*

General Terms: Algorithms, Design, Theory

Keywords: Incomplete Trace, Data Race, Maximal, Precise

1. Introduction

Data races are an important class of concurrency errors plaguing software systems today. A data race is commonly defined as two unordered, conflicting accesses without intervening synchronization. Because the two racy accesses may be executed in different orders, programs with data races are often non-deterministic, making testing and debugging them notoriously challenging. What is worse is that data races make it extremely difficult to reason about program correctness, because in high-level languages such as Java and C/C++, the semantics of data races are usually subtle or undefined. Even though a data race may look benign in the source code, compilers and hardware may transform it into harmful bugs [6, 12, 13].

Researchers have proposed a wide spectrum of race detection techniques [8, 10, 14, 21–23, 26, 33, 34, 42, 50, 52, 54, 57]. However, eliminating races in real-world programs remains impractical. A crucial issue is that existing techniques tend to not only detect true races, but also produce many false alarms or false positives.[1]

False alarms are particularly problematic for race detection tools, because races are surprisingly difficult to diagnose and validate. To correctly determine if a reported race is a false alarm, the developer would need to analyze all possible orderings of computations from different threads in all feasible paths, the space of which is often enormous for realistic programs. Even if a race looks suspicious, it may still be a false alarm due to certain subtle synchronizations that are not (yet) understood by the programmer. Worse, real bugs such as deadlocks could be added while attempting to fix a spurious race [26]. Consequently, any false alarms could significantly decrease programmer productivity and make the tool less useful.

The reasons for the false alarms are twofold. First, the problem of precisely detecting all races in a program in general is undecidable [49]. To scale to large programs, existing techniques often overly approximate races. For example, the *LockSet* algorithm [54] implemented in state of the art race detectors [47, 56] is known to be imprecise. Moreover, the challenge is rooted not only in the algorithmic complexity, but also from various practical issues:

- **Unavailability of whole program.** Real-world systems often rely on external libraries, proprietary code, and/or are composed from layers of frameworks and extended by third-party plugins. These programs may even be loaded on the fly over the network. Analyzing the whole

OOPSLA'16, November 2–4, 2016, Amsterdam, Netherlands
© 2016 ACM. 978-1-4503-4444-9/16/11...$15.00
http://dx.doi.org/10.1145/2983990.2984024

[1] In this paper we do not distinguish between benign and harmful data races. Any false positive race is considered as a false alarm. See Section 6 for more discussions on benign and harmful races.

program to find all synchronizations is difficult or impossible.

- **Limitation of logging facilities.** Many dynamic techniques require capturing a full program execution trace through static or dynamic instrumentation. The logging facilities may be limited to certain languages or cannot handle certain language features, such as builtin libraries (*e.g.*, `java.*`) or code written in a lower level language (*e.g.*, Java Native Interface (JNI) [2]).
- **Performance slowdown.** Many applications or components are performance sensitive or have resource constraints such that they cannot tolerate any runtime slowdown or memory overhead incurred by tracing, otherwise they would fail to function properly. In some scenarios, we may even desire to exclude certain code from tracing to improve performance. For example, when debugging code which implements a new feature, developers may be interested in detecting races in that specific code region and would want to skip the others.

In all these situations, we may end up missing vital program trace information. When only partial program information is available or observed, all existing precise algorithms (*i.e.*, *Happens-Before* (HB) [40], *Causally-Precedes* (CP) [57], *Maximal-Causality* (MC) [33]) become imprecise. For example, the classical HB algorithm [40] is precise, given that all critical events in the program execution are captured. However, this requirement can hardly be satisfied in practice, and HB-based tools [26, 56] tend to report many false alarms [5] on real-world programs. For example, as quoted from ThreadSanitizer (an industrial strength race detector) [4]: "*blacklisted functions are not instrumented. This can lead to false positives due to missed synchronization*".

In this paper, we present a new dynamic race detection technique, **RDIT** (***R**ace **D**etection from **I**ncomplete **T**races*), that is both *precise* and *maximal* even when the trace is incomplete, *i.e.*, certain events in the program execution are not tracked or are missed. "Preciseness" means no false positives. "Maximality" means that no more precise races can be detected based on *the same incomplete trace*. In other words, RDIT detects a maximal set of true races that can happen in all possible schedules inferred from an observed (complete or incomplete) trace.

RDIT is underpinned by a novel BarrierPair model of an incomplete trace, which soundly abstracts the behavior of missing events through the invocation data of their enclosing methods. BarrierPair is safe since it conservatively[2] assumes that all runtime data at the invocation sites of a method that is not logged will be accessed inside the method and may introduce synchronization. Meanwhile, it is the least conservative approach, in that any data non-reachable from the method's runtime arguments will not be accessed inside the method and hence does not introduce synchronization. Moreover, inspired by *Maximal-Causality* (MC) [33], we integrate BarrierPair with MC by formulating the race detection problem as a constraint solving problem. By solving a set of quantifier-free first-order logic formulas using an off-the-shelf SMT solver, RDIT is able to detect races precisely with maximal capability. However, different from previous work [33], RDIT allows arbitrary events to be missed in the trace without reporting any false alarms.

RDIT is built upon an initial idea developed in our prior work published in a short paper [51]. This paper significantly improves [51] with substantially deeper technical depth and extensive evaluation of BarrierPair. Moreover, the race detection algorithm developed in this paper (based on BarrierPair and MC) is completely different from that in [51] in order to achieve soundness and maximality for general traces with arbitrary number of threads.

We anticipate that RDIT will be useful in several practical scenarios. First, RDIT can be applied in systems (*e.g.*, multi-language programs) where it is difficult to trace certain computations. Second, RDIT can be used in programs with third party libraries or user extensions of which the complete code is unavailable. Third, RDIT is useful in performance sensitive applications that cannot tolerate any instrumentation slowdown. Users of RDIT can selectively exclude or include code sections/modules from the instrumentation. Fourth, RDIT can speed up the runtime for localized debugging where developers are only interested in certain code regions (*e.g.*, new features) and can skip logging code that they believe is race-free.

We have implemented RDIT for Java and evaluated it on seven real-world large multithreaded applications including *Eclipse IDE*, *Apache Derby Database*, and *Floodlight SDN controller*. RDIT detects a total of 85 true races in these systems with zero false positives, though it detects 27 fewer true races than MC due to its conservativeness. In contrast, existing precise algorithms (HB, CP, and MC) report hundreds of false alarms (149, 149, and 213, respectively) when synchronization methods are missing from the trace. Moreover, RDIT improves the overall program performance significantly when used for capturing incomplete traces in practice, and capturing the BarrierPairs incurs only 4%-13% runtime overhead after applying an optimization to compute reachable runtime data of missing methods.

Limitations. We note that the precision of RDIT relies on two assumptions. In theory, the RDIT tool may still report false positives if these two assumptions are not satisfied. First, direct accesses to globals in a missing method must be annotated by users (see Section 2.2 Caveat 5). Second, the optimization used by RDIT to compute reachable runtime addresses of missing methods (see Section 3.2.1) approximates heap reachability. It is sound only when all addresses used for synchronization are reachable from the missing methods' runtime arguments at the time of invoca-

[2] By "conservative", in this paper we mean that the technique makes sure not to detect false alarms.

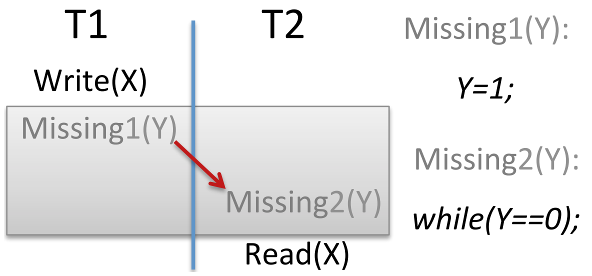


Figure 1: Ad-hoc synchronization in the missing methods results in false alarms reported by Happens-Before.

tion. If a new reachable address is introduced inside a missing method, and that new address is used in a subsequent missing method for synchronization, it may lead to missing synchronizations.

In summary, this work makes the following contributions:

- We present a precise and maximal dynamic race detection technique that detects a maximal set of true races from incomplete traces without any false alarms.
- We present a novel model of an incomplete trace that abstracts away the missing events by capturing the invocation data of their enclosing methods. This model forms a foundation for capturing maximal thread causality in the presence of missing events.
- We present an extensive evaluation of our technique on a range of real-world concurrent systems and demonstrate the race detection effectiveness and runtime performance.

2. Overview

We start by illustrating the problem of incomplete traces with an example. We then introduce the BarrierPair model and discuss the key technical challenges of realizing a precise and maximal race detection technique based on this model.

Example. Consider the trace in Figure 1. We have two threads $T1$ and $T2$ performing a *Write* and a *Read* on a common address X. The greyed out region in between the two events is the region of interest where we would like to check for any synchronization. The synchronization can either be in the form of a *Happens-Before* (HB) edge inducing event such as *Lock*/*Unlock*, *ThreadFork*/*ThreadJoin*, or an ad hoc synchronization, which causes an ordering in the program execution. In the absence of any such synchronization, we will flag the two events as a race. Therefore, when all computations in this region are missed, existing precise algorithms [33, 40, 57] will all report a race between the two accesses. However, this is a false alarm when the two missing methods in the region introduce a synchronization (any standard or ad hoc synchronization) on a shared address Y (assume Y is a volatile flag and the value is initialized to 0). Thread $T1$, after performing the *Write* to X, sets Y to 1, while Thread $T2$ waits until Y is set before it can perform the *Read* on X. The shared address Y is used as a barrier in Thread $T2$ to induce a desired HB ordering.

Caveat 0: Simple Barrier. A simple approach to avoiding false alarms in the presence of these missing events would be to consider each missing event (or a sequence of continuous missing events) as a barrier, and add HB edges between barriers in the observed order. For now we dismiss global variables, which will be discussed in Section 2.2. This approach would guarantee to detect no false alarm, because it strictly serializes the missing events. However, it is also overly conservative in that it would miss many true data races. For instance, if the two missing methods in Figure 1 are *empty* or *access different data*, there will be a true race on the two accesses to X, but this simple barrier approach will miss it.

Key Idea. Our technique provides the same precision guarantee as the simple barrier approach, however, at the minimal cost of missing true races. Our key observation is that although the computations inside the missing methods are unknown, the invocation of those missing methods can usually be captured. The runtime data at the invocation sites actually provides valuable information to approximate the behavior of the missing computations. For example, consider our example in Figure 1 again. Both of the two missing methods in threads $T1$ and $T2$ have accesses to the same memory address Y. In the absence of this shared address, there is no possibility for these two missing methods to introduce any synchronization. More generally, if the two missing methods have addresses A and B, respectively, in their scope, and if $A \wedge B = \emptyset$, then we can safely conclude that no ordering can be induced through this pair of missing methods. Meanwhile, if $A \wedge B \neq \emptyset$, without knowing any other information, the missing methods may use the overlapped addresses to synchronize. This observation leads to our first contribution in this work – the BarrierPair model, explained next.

2.1 The BarrierPair Model

Instead of abstracting each missing event as a barrier, we introduce two events for each *missing method call* – (***MethodBegin***, ***MethodEnd***), and refer to this pair of events as a BarrierPair. Specifically, a BarrierPair is associated with the following attributes:

- *Tid*: a thread ID denoting the thread that calls the missing method.
- *Begin*: a *MethodBegin* event corresponding to the invocation of the missing method.
- *End*: a *MethodEnd* event corresponding to the return of the missing method.
- *D*: a set of memory addresses that can be reached by the missing method.
- *Between*: a (possibly empty) set of observed events that occur in-between the *MethodBegin* and *MethodEnd* events from the particular thread. These events can be introduced by callback functions.

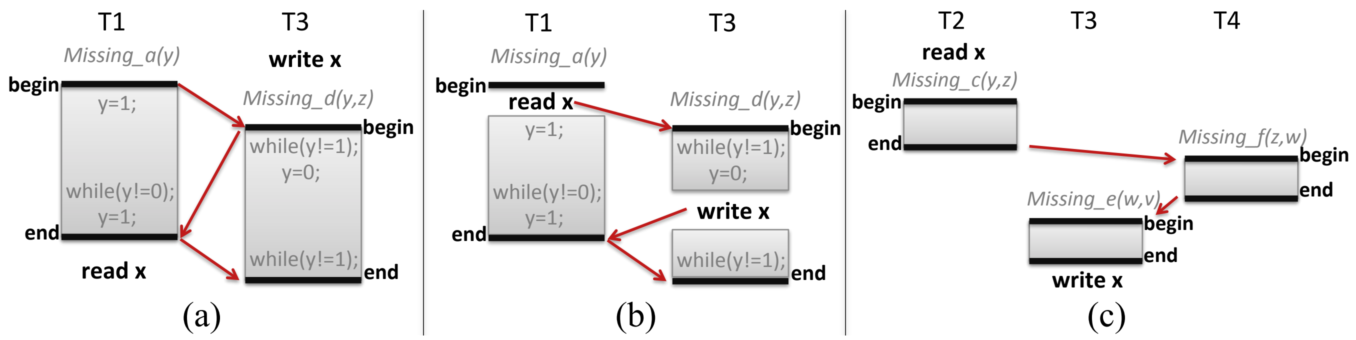


Figure 3: Examples illustrating the caveats (explained in Section 2.2). The threads and BarrierPairs correspond to that in Figure 2 with minor modification on the reachable addresses. To make the synchronization valid, we assume that the variables y, z, w used in the synchronizations are declared volatile. A false alarm race on the two observed accesses to x would be reported if any of the HB edges (denoted by the red arrows) is missed. The illustrated caveats are: (a) Overlapping BarrierPairs can incur multiple HB edges; (b) Events in between BarrierPairs may be observed and can introduce HB edges; (c) Multiple BarrierPairs can introduce HB edges transitively.

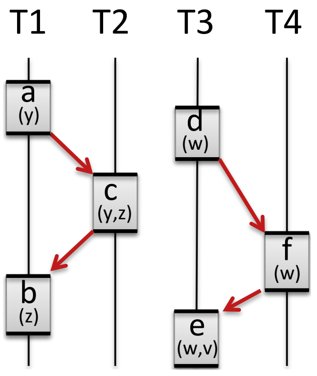


Figure 2: A program trace consisting of four threads and six BarrierPairs ($a - f$), each denoting a missing method call with its reachable memory addresses. For example, the BarrierPair $a(y)$ denotes that the corresponding missing method a may access address y. Four HB edges ($a \to c$, $c \to b$, $d \to f$, $f \to e$) are added between those BarrierPairs with overlapping reachable addresses.

The two events *MethodBegin* and *MethodEnd* are similar to the other types of events in the trace (we will present a formal model in Section 3) and all such events are globally ordered. We require that for each missing method these two events are always paired. In the occurrence of uncaught exceptions during a missing method call, we enclose the method by a *try-catch* block and re-throw the exceptions. Other events can also occur in-between a BarrierPair and be recorded in the trace, and multiple BarrierPairs may be nested.

The attributes of a BarrierPair can be recorded and computed at runtime without knowing the computation in the missing methods. This information can be used to *approximate* the synchronization behavior between missing methods. For example, if the memory addresses that can be reached by two BarrierPairs from different threads do not overlap, we can safely conclude that no ordering can be induced through this pair of missing methods. If they do overlap, they may be synchronized and we should then add HB edges to denote their ordering. Figure 2 illustrates six BarrierPairs in a trace and four added HB edges between them. With this enhancement, the same HB algorithm [26, 40] or other precise algorithms [33, 57] can be directly applied to detect races without any change.

Moreover, the BarrierPair model matches with the real-world usages naturally. The user can choose to exclude certain methods, classes, or packages from tracing with command line options such as "`--exclude=java.*,sun.*`" to instruct the instrumentation tool not to trace methods in these packages. This is actually a standard step used in many existing analysis frameworks [3, 33, 39]. It reduces both the trace size and runtime overhead, and also avoids the problem of tracing native code used in those excluded methods. Furthermore, a BarrierPair can be used to approximate the computation inside the missing method. For example, if a missing method is deterministic, the same invocation data recorded in a BarrierPair will always produce the same computation by calling this missing method.

2.2 Technical Challenges

The BarrierPair model paves the foundation for precise race detection from incomplete traces. However, there are several tough challenges we must tackle to develop a race detection technique that is both precise and maximal:

1. How to add HB edges that are both sufficent to guarantee precision and minimal to guarantee maximality?
2. How to compute (and compute efficiently) the full set of reachable memory addresses for each BarrierPair?
3. How to perform race detection that can maximize the detection power given an incomplete trace?

The first two challenges are fundamental to the soundness of our technique; we describe next a few caveats to illustrate these challenges. We then present our race detection technique in detail in Section 3. We note that these caveats are not exhaustive. However, our race detection technique is both precise and maximal, powered by the maximal causality model and the BarrierPair model.

Caveat 1: Overlapping BarrierPairs. Intuitively, we can enforce orderings between BarrierPairs with overlapping reachable addresses by adding a HB edge from one BarrierPair to another in the observed order in the trace. For example, in Figure 2, we add the HB edge $a \rightarrow c$ from the *MethodEnd* of a to *MethodBegin* of c, because a and c have an overlapping reachable address, y, and a occured before c. However, this naive method does not work when two BarrierPairs overlap in time. For instance, the BarrierPair d in Figure 2 overlaps with a. Suppose d also accesses address y, we cannot simply add a single HB edge from a to d or from d to a, but multiple HB edges may be needed. The reason is that the overlapping region may incur multiple HB edges between events in the missing methods. Consider an example in Figure 3(a). Three HB edges must be added between the *MethodBegin* and *MethodEnd* events of the two BarrierPairs, because of the ad hoc synchronizations incurred by the missing events on y. For instance, the HB edge $d.begin \rightarrow a.end$ must be added, because the *MethodEnd* event of BarrierPair a cannot happen until y is set to 0 by Thread $T3$, which is after the *MethodBegin* of BarrierPair d. Otherwise, a false alarm would be reported between the *Read* to x in Thread $T1$ and *Write* to x in Thread $T3$.

Caveat 2: Observed Events in-between BarrierPairs. Although computations inside missing methods are opaque, events from a missing method call may still be observed, for example, through callback functions. When events appear between the *MethodBegin* and *MethodEnd* events of a BarrierPair, their orderings with other BarrierPairs must be correctly enforced. Consider a trace in Figure 3(b) (slightly modified from Figure 3(a)). The *Read* and *Write* events to x in the two missing methods are both observed in the trace. We would report a race between them if we consider the same HB edges as that in Figure 3(a). However, this is a false alarm because the *Write* cannot happen until x is set to 1 by Thread $T1$, which is after the *Read*. Therefore, we must add HB edges between these observed events and the BarrierPair events.

Caveat 3: Orderings Between BarrierPairs and Ordinary Events. A BarrierPair can introduce HB orderings not only with other BarrierPairs and events in-between them, but also with those ordinary events outside missing methods. Consider again Figure 3(b). Suppose the method d in Thread $T3$ is not missing, the events at "*while y!=1*" are ordinary events. We must add a HB edge from the event *Read*(x) in Thread $T1$ to these ordinary events. Otherwise, similar to Caveat 2, a false alarm would be reported between *Read*(x) and *Write*(x).

Caveat 4: Transitive Orderings Over Multiple BarrierPairs. HB orderings are transitive. Two BarrierPairs without any common reachable address does not mean that they cannot be ordered, because they may be ordered transitively through other events or BarrierPairs. False alarms might be reported if we only consider BarrierPairs pairwisely. For example, consider the three BarrierPairs c, e, and f shown in Figure 3(c), and suppose f can also access address y. Because y is also accessed by c, a HB edge $c \rightarrow f$ from BarrierPair c to f must be added. And also because $f \rightarrow e$, we have $c \rightarrow e$. That is, the BarrierPair c must happen before e, though they do not have any common reachable address. Hence, the two accesses to x by threads $T2$ and $T3$ are ordered by HB edges and are not a race.

Caveat 5: Global Variables. In the BarrierPair model, we have made the assumption that the addresses used to perform synchronizations are local in scope, *i.e.*, they are passed in as runtime parameters at the missing method's invocation site. For addresses that are global in scope, such as `public static` variables in Java, their contribution to synchronization is ignored. However, if such global variables are *directly accessed*[3] in missing methods, false alarms may be introduced.

One way to address this issue is to use the simple barrier approach which, as explained earlier in *Caveat 0*, is overly conservative such that it would miss many true races. Instead, we propose a language extension that allows the users of RDIT to annotate direct global variable accesses at the call sites of missing methods. Specifically, we provide a custom Java annotation `@Global(X)` that users can insert before invocations of missing methods to specify that the global variable `X` (which is either an object reference or a volatile primitive) may be directly accessed in a missing method. For example, static synchronizations to a class `C` are defined as `@Global(C.class)`. At runtime, `X` is added to the set of reachable memory addresses of the BarrierPair. This method guarantees soundness, though reducing automation.

Nevertheless, we note that directly accessing global variables in external methods is rarely seen in real-world production systems. In our studied real-world systems, the only such cases are those to immutable global variables through singleton, which do not introduce any synchronization at all. In other words, to use RDIT annotations are almost never needed in practice.

3. The RDIT Technique

We first present in Section 3.1 a formal model of maximal thread causality with missing events, following the approach

[3] Note that in the BarrierPair model global variables are allowed to be accessed in missing methods, and as long as their accesses are visible (e.g., through callbacks), no false alarm will be introduced.

introduced in MC [33] (there without missing events). A key advancement of this new model is that it incorporates the notion of BarrierPair to guarantee both soundness and maximality from incomplete traces. We then present our RDIT algorithm in Section 3.2, including how to compute the reachable memory addresses of BarrierPairs and how to encode the new model with constraints. Our constraint encoding shares the same spirit with prior work [33] to guarantee soundness and maximality. Differently, we must consider the additional constraints introduced by the BarrierPairs.

3.1 Maximal Causality Model with Missing Events

Consider an arbitrary multithreaded program $\mathcal{P}$. It can be abstracted as a set of finite traces that it can produce when completely or partially executed, called *$\mathcal{P}$-feasible* traces. A *trace* is a sequence of events, which are operations performed by threads on concurrent objects. The following common event types are often considered in previous race detection work [26, 33, 57]:

- *Read*(t,x,v)/*Write*(t,x,v): read/write x with value v;
- *Lock*(t,l)/*Unlock*(t,l): acquire/release a lock l;
- *ThreadBegin*(t): the first event of thread t;
- *ThreadEnd*(t): the last event of thread t;
- *ThreadFork*(t,t'): fork a new thread t';
- *ThreadJoin*(t,t'): block until thread t' terminates;

Note that the event *value* is also a part of the definition. For example, if the value returned by a read is changed, it becomes a different read event, such that a conditional after the event may produce a different trace.

In this work, in addition to the usual events above, we include two new events:

- *MethodBegin*(t,m,D): invoking a method m that is missing with a set of reachable addresses D.
- *MethodEnd*(t,m): returning from a missing method m.

Similar to *Lock* and *Unlock* events, *MethodBegin* and *MethodEnd* events can appear anywhere in the trace and can be nested, but they are alway paired for the same thread t and method m. Each pair of *MethodBegin* and *MethodEnd* events forms a BarrierPair, which indicates that certain events in between these two events from the same thread are missed in the trace, and those events can perform arbitrary operations on any objects in D.

The sets of $\mathcal{P}$-feasible traces must obey two basic consistency axioms: *prefix closedness* and *local determinism*. The former says that the prefixes of a $\mathcal{P}$-feasible trace are also $\mathcal{P}$-feasible. The latter says that each thread has deterministic behavior, that is, only the previous events of a thread (and not other events of other threads) determine the next event of the thread, although if that event is a read then it is allowed to get its value from the latest write. For any consistent trace τ, these two axioms allow us to associate it with a *maximal causal model*, *MCM*(τ), which comprises precisely those traces that can be generated by any program that can generate τ. Specifically, from τ, we can infer a sound and maximal set of traces *MCM*(τ) by checking the two axioms, such that (1) any program that can generate τ can also generate all traces in *MCM*(τ), and (2) for any trace τ' not in *MCM*(τ) there exists a program generating τ which cannot generate τ'. Note that *MCM*(τ) here is different from that in prior work [33], because τ is incomplete and contains BarrierPairs that abstract missing events.

Algorithm 1 The RDIT Algorithm

```
1: $\tau$: input trace;
2: $O_e$: order variable for event $e$.
3: $BP$ = ComputeBarrierPairs($\tau$);
4: $\Phi_{mcm}$ = ConstructMCMFormula($\tau$, $BP$);
5: for all conflicting events $(a, b)$ in $\tau$ do
6:     if $\Phi_{mcm} \wedge (O_a = O_b)$ is satisfiable then
7:         report race $(a, b)$.
8:     end if
9: end for
```

3.2 Data Race Detection Algorithm

To perform precise and maximal race detection, intuitively, we can generate *MCM*(τ) and detect races in every trace in the set. However, generating *MCM*(τ) is challenging. Exhaustively enumerating all reorderings of τ and checking against the two axioms is impractical. Moreover, the semantics of BarrierPairs must be correctly modeled to ensure soundness (recall the caveats in Section 2.2). In RDIT, following [33], we encode *MCM*(τ) as a series of quantifier-free first-order logic formulas, Φ_{mcm}, such that any solution to Φ_{mcm} represents a trace in *MCM*(τ). By modeling races as additional constraints, we formulate the race detection problem as a constraint solving problem.

Specifically, given an input trace τ, the goal of RDIT is to find a trace τ' in *MCM*(τ) with two conflicting events (*i.e.*, *Read*/*Write* events, accessing the same memory address, at least one is a *Write*) a and b from different threads, such that a and b are next to each other in τ'. Algorithm 1 outlines our race detection algorithm. A key step is to introduce an order variable O for each event e in τ, denoting the order of e in τ', and use these order variables to encode Φ_{mcm}. We first compute the set of all BarrierPairs from τ. This step is mostly straightforward except that we need to efficiently compute the set of reachable memory addresses for each BarrierPair (explained shortly). We then construct the formula Φ_{mcm} from τ and the BarrierPairs. Finally, for each pair of conflicting events (a, b) from different threads, we invoke an SMT solver to solve Φ_{mcm} conjuncted with the race constraint $O_a = O_b$. If the solver returns a solution, it means that there exists a trace in *MCM*(τ) in which the two events a and b are unordered, and hence (a, b) is a true race.

3.2.1 Computing Reachable Memory Addresses

The set of *reachable memory addresses* of a BarrierPair is the union of all reachable addresses from runtime parameters passed at the invocation of the corresponding missing method. For Java programs, the reachable addresses of an object can be represented by a tree whose nodes are objects and edges denote field references (back edges are removed). The object tree can be generated using the Java reflection mechanism. To compute a complete set, for each *MethodBegin* event, we would need to track every method parameter object and iterate through its declared fields and inheritance stack to compute the object tree. However, this may incur a large runtime overhead and produce huge logs when calls to missing methods are frequent and the object tree is large. We will present the performance results in Section 5.3.

An Optimization. We develop an efficient method that does not compute the complete set of reachable addresses for every object upon every missing method call, but only *once* for each object for *all* missing method calls. This optimization is unsound in theory, but works well in practice. The key observation is that the object tree is static most of the time. It is only changed when write operations to field references (*i.e.*, $o_1.f = o_2$) are performed. Before such a write operation, the object tree of o_1 needs to be computed only once and can be *reused*, and upon an update operation, only the subtree from $o_1.f$ needs to be updated. Moreover, any such operation is either recorded in the trace or missed because it is from a missing method. If the former, we can recover o_2 by analyzing the trace. For the latter, we may ignore the update because o_2 might be already included in the set of reachable addresses, D, of the missing method. The only condition is that if o_2 is not in D, it should *not* be used for synchronization. In fact, this condition is never violated in our study of real-world applications (see Section 5). Therefore, in this optimization, for each object at runtime, we compute and log its object tree only once, and we recover the updates made by object field *Write* events in the trace analysis phase, which is offline.

3.2.2 Constraint Encoding of *MCM*(τ)

Algorithm 2 shows our constraint encoding algorithm for *MCM*(τ). Φ_{mcm} is constructed with three kinds of operators, "$<$" (less than), "$\wedge$" (conjunction), and "$\vee$" (disjunction), over the order variables O, and "$<$" is transitive. Φ_{mcm} is a conjunction of the following five types of constraints:

1. **BarrierPair Constraints** (Lines 7-20). This type of constraints capture the HB edges between the missing events themselves and between the missing events and the observed events. For each pair of BarrierPairs, if their reachable addresses overlap, we linearize all of their associated events (including both *MethodBegin*/*MethodEnd* events and the *Between* events associated with the BarrierPair), and construct constraints to enforce HB orderings between them. The rationale is that a missing event may exist anywhere in a BarrierPair and may introduce synchronization with any other event (either observed or not) accessing the overlapped

Algorithm 2 ConstructMCMFormula(τ, *BP*)

```
1:  $\tau$: input trace;
2:  $BP$: all BarrierPairs in $\tau$.
3:  $T \leftarrow$ GetAllThreads($\tau$);
4:  $L \leftarrow$ GetAllLocks($\tau$);
5:  $\Phi_{mcm}$ = true; // initialized to true
6:  // 1. Construct BarrierPair Constraints
7:  for $bp_1, bp_2 \in BP$ do
8:      if $bp_1.D \wedge bp_2.D \neq \emptyset$ then
9:          $S \leftarrow$ UnionEvents($bp_1$, $bp_2$);
10:         $\Phi_{mcm}$ $\wedge$= GetLinearizationConstraints($S$);
11:     end if
12: end for
13: for $bp \in BP$ do
14:     for $x \in bp.D$ do
15:         for $e \in$ GetAllReadWritesOnAddress($\tau$,$x$) do
16:             $S \leftarrow$ UnionEvents($bp$, $e$);
17:             $\Phi_{mcm}$ $\wedge$= GetLinearizationConstraints($S$);
18:         end for
19:     end for
20: end for
21: // 2. Construct Program Order Constraints
22: for $t \in T$ do
23:     $\tau_t$ = GetThreadEvents($\tau$, $t$); // events by Thread $t$
24:     for $i = 1{:}|\tau_t| - 1$ do
25:         // $O_{t,i}$: order variable of the $i$th event in $\tau_t$
26:         $\Phi_{mcm}$ $\wedge$= $O_{t,i} < O_{t,i+1}$;
27:     end for
28: end for
29: // 3. Construct Fork Join Constraints
30: for $e \in$ GetThreadForkJoinEvents($\tau$) do
31:     if $e$ =ThreadFork($t$, $t'$) then
32:         $\Phi_{mcm}$ $\wedge$=$O_e < O_{t',begin}$;
33:     else if $e$ =ThreadJoin($t$, $t'$) then
34:         $\Phi_{mcm}$ $\wedge$=$O_{t',end} < O_e$;
35:     end if
36: end for
37: // 4. Construct Locking Constraints
38: for $l \in L$ do
39:     // pairs of lock/unlock events on $l$
40:     $LP_l$ = GetLockPairs($\tau$, $l$);
41:     for $(e_a, e_b), (e_c, e_d) \in LP_l$ do
42:         $\Phi_{mcm}$ $\wedge$= $(O_{e_b} < O_{e_c} \vee O_{e_d} < O_{e_a})$;
43:     end for
44: end for
45: // 5. Construct Read Consistency Constraints
46: for $e$ =Read($t$, $x$, $v$) $\in \tau$ do
47:     $W^x \leftarrow$ GetAllWritesOnAddress($\tau$,$x$);
48:     $W^x_v \leftarrow$ GetAllWritesOnAddressValue($\tau$,$x$,$v$);
49:     $\Phi_{mcm}$ $\wedge$=$\bigvee_{w \in W^x_v} (O_w < O_e \bigwedge_{w \neq w' \in W^x} (O_{w'} < O_w \vee O_e < O_{w'}))$;
50: end for
```

Algorithm 3 GetLinearizationConstraints(S)

```
1: $S$: an input set of events;
2: $\phi$ = *true*;
3: $Z$ = **LinearizeByGlobalId**($S$);
4: **for** $i$ = 1:$|Z| - 1$ **do**
5:     $\phi \wedge= O_{Z[i]} < O_{Z[i+1]}$;
6: **end for**
7: return $\phi$
```

addresses. Specifically, the function **UnionEvents** first unions all these events into a set S. Then the function **GetLinearizationConstraints** (Algorithm 3) linearizes the events in S into an ordered list Z by their order (*i.e.*, GlobalId) in the input trace, and returns a formula in terms of "$O_{Z[i]} < O_{Z[i+1]}$" conjuncted over all events $Z[i]$. Similarly, for each BarrierPair and any ordinary *Read*/*Write* event accessing an overlapped address[4], we construct constraints to enforce their HB orderings.

2. **Program Order Constraints** (Lines 22-28). This type of constraints ensure sequential consistency, such that events from the same thread cannot be reordered. Specifically, we construct a constraint $O_{e_1} < O_{e_2}$ whenever e_1 and e_2 are events by the same thread and e_1 occurs before e_2. Note that because HB is transitive, it is sufficient to conjunct such constraints between consecutive events from the same thread. This type of constraints can also be weakened to reflect relaxed memory models such as TSO and PSO [36]. Nevertheless, we focus on sequential consistency in this work.
3. **Fork Join Constraints** (Lines 30-36). The semantics of *ThreadFork* and *ThreadJoin* events requires that a *ThreadBegin* event can happen only after the thread is forked by *ThreadFork* from another thread, and that a *ThreadJoin* event can happen only after the *ThreadEnd* event of the joined thread. We hence construct a constraint $O_{e_1} < O_{e_2}$ when e_1 is an event of the form *ThreadFork*(t, t') and e_2 of the form *ThreadBegin*(t'), or when e_1 is an event of the form *ThreadEnd*(t) and e_2 of the form *ThreadJoin*(t', t).
4. **Locking Constraints** (Lines 38-44). The locking semantics requires that any two code regions protected by the same lock are mutually exclusive. We first extract all pairs of *Lock*/*Unlock* events for each lock l, following the program order locking semantics: *Unlock* is paired with the most recent *Lock* on the same lock by the same thread. Then for each two such pairs, (e_a,e_b), (e_c,e_d), we construct the constraint ($O_{e_b} < O_{e_c} \vee O_{e_d} < O_{e_a}$) and conjunct them.
5. **Read Consistency Constraints** (Lines 46-50). This type of constraints ensures that the two basic axioms (recall Section 3.1) are satisfied by requiring that every event in the *inferred* trace τ' is feasible. Due to prefix closedness, τ' does not necessarily contain all the events in τ but may contain a subset of them. Due to local determinism, an event is feasible if every read it depends on gets the *same value* as that in τ. Each read, however, may read a value written by any write on the same address, as long as all the other constraints are satisfied. We hence construct a constraint (shown in Algorithm 2) for each *Read*$(t,\ x,\ v)$ event such that it is allowed to read the value v on x written by any *Write* event w, subject to the condition that w writes to x with v, and there is no other intervening *Write* to x with a different value. The size of read consistency constraints is cubic in the number of *Read*/*Write* events, and may dominate the size of Φ_{mcm}.

[4] At the programming language level, these addresses should be volatile for Java and C/C++ in order to introduce synchronization.

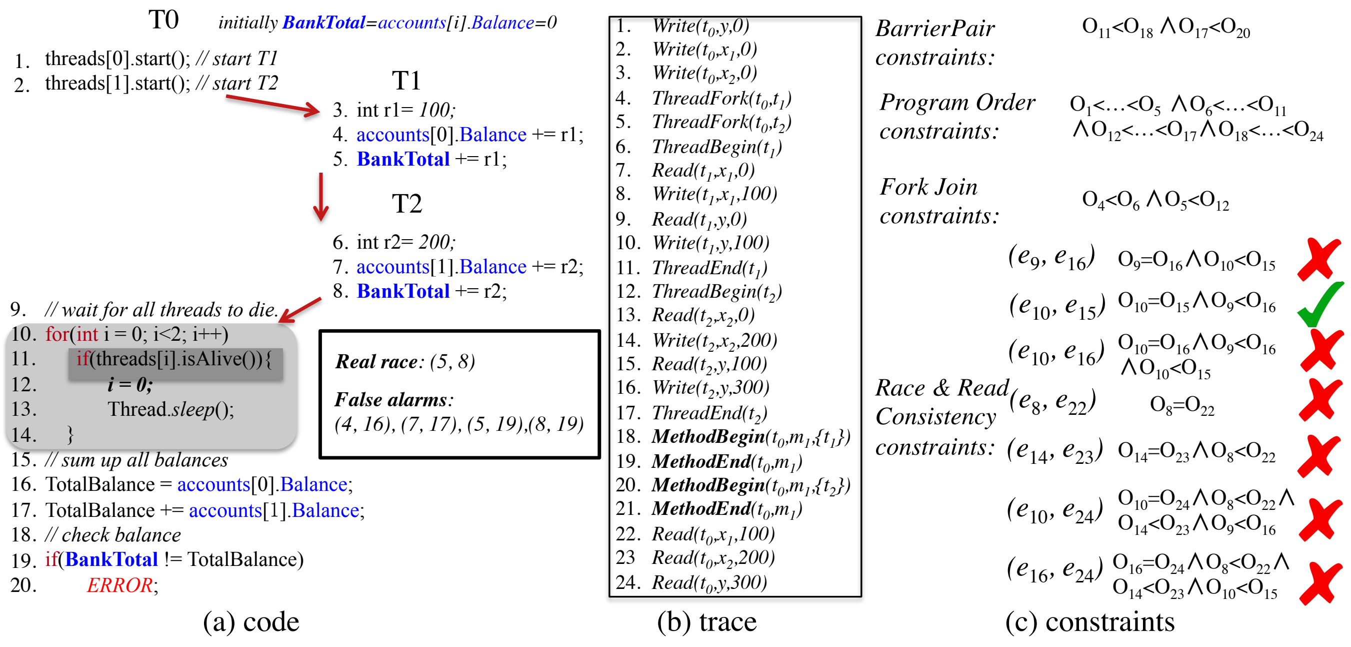
Soundness and Maximality. It is important to note that the constructed formula Φ_{mcm} is both sound and maximal (assuming sequential consistency). That is, Φ_{mcm} encodes precisely all the feasible traces in *MCM*(τ), and each solution of the order variables to Φ_{mcm} corresponds to a valid reordering of events in τ. The proof can be derived from the soundness and maximality proof of MC [33]. The key difference here is that the BarrierPair constraints should precisely capture all the necessary happens-before orderings that the missing events may incur. This can be proved by the same reasoning as we construct the BarrierPair constraints: a missing event may exist anywhere in a BarrierPair and it may introduce a happens-before ordering with any other event (either observed or not) accessing an overlapped address.

The complexity of *MCM*(τ) may be exponential in the trace size, as the number of unique solutions to Φ_{mcm} can be exponential. In RDIT, however, we do not need to directly solve Φ_{mcm} to produce all the traces in *MCM*(τ). Instead, it suffices to find one trace that satisfies the race condition.

4. A Case Study

In this section, we present a case study of race detection in a popular multithreaded benchmark – *Account* (Figure 4). We show that all existing precise algorithms [26, 33, 57] report several false alarms in this benchmark due to missing events in the native library. We illustrate how RDIT detects the only true race while suppressing all false alarms.

False Alarms in the Account Benchmark. This benchmark has been used frequently in previous race detection studies [26, 33, 38, 55, 57]. In this program, a number of bank accounts are simulated by concurrent threads to handle deposits. The sum of deposited amounts by all threads is tracked dynamically. At the end of the execution, the sum is compared with the total balance of all accounts. If they are not equal, it indicates a concurrency error. Figure 4(a) shows code snippets of the main thread ($T0$) and two account threads ($T1$ and $T2$). The loop at lines 10-14 in $T0$


T0
initially BankTotal=accounts[i].Balance=0
T1
T2
Real race: (5, 8)
False alarms: (4, 16), (7, 17), (5, 19),(8, 19)
BarrierPair constraints: $O_{11}<O_{18} \wedge O_{17}<O_{20}$
Program Order constraints: $O_1<\ldots<O_5 \wedge O_6<\ldots<O_{11} \wedge O_{12}<\ldots<O_{17} \wedge O_{18}<\ldots<O_{24}$
Fork Join constraints: $O_4<O_6 \wedge O_5<O_{12}$
Race & Read Consistency constraints:
(e_9, e_{16}) $O_9=O_{16} \wedge O_{10}<O_{15}$ ✗
(e_{10}, e_{15}) $O_{10}=O_{15} \wedge O_9<O_{16}$ ✓
(e_{10}, e_{16}) $O_{10}=O_{16} \wedge O_9<O_{16} \wedge O_{10}<O_{15}$ ✗
(e_8, e_{22}) $O_8=O_{22}$ ✗
(e_{14}, e_{23}) $O_{14}=O_{23} \wedge O_8<O_{22}$ ✗
(e_{10}, e_{24}) $O_{10}=O_{24} \wedge O_8<O_{22} \wedge O_{14}<O_{23} \wedge O_9<O_{16}$ ✗
(e_{16}, e_{24}) $O_{16}=O_{24} \wedge O_8<O_{22} \wedge O_{14}<O_{23} \wedge O_{10}<O_{15}$ ✗
(a) code
(b) trace
(c) constraints

```
1.  threads[0].start(); // start T1
2.  threads[1].start(); // start T2
        3. int r1= 100;
        4. accounts[0].Balance += r1;
        5. BankTotal += r1;
        6. int r2= 200;
        7. accounts[1].Balance += r2;
        8. BankTotal += r2;
9.  // wait for all threads to die.
10. for(int i = 0; i<2; i++)
11.     if(threads[i].isAlive()){
12.         i = 0;
13.         Thread.sleep();
14.     }
15. // sum up all balances
16. TotalBalance = accounts[0].Balance;
17. TotalBalance += accounts[1].Balance;
18. // check balance
19. if(BankTotal != TotalBalance)
20.     ERROR;
```

```
1.  Write(t0,y,0)
2.  Write(t0,x1,0)
3.  Write(t0,x2,0)
4.  ThreadFork(t0,t1)
5.  ThreadFork(t0,t2)
6.  ThreadBegin(t1)
7.  Read(t1,x1,0)
8.  Write(t1,x1,100)
9.  Read(t1,y,0)
10. Write(t1,y,100)
11. ThreadEnd(t1)
12. ThreadBegin(t2)
13. Read(t2,x2,0)
14. Write(t2,x2,200)
15. Read(t2,y,100)
16. Write(t2,y,300)
17. ThreadEnd(t2)
18. MethodBegin(t0,m1,{t1})
19. MethodEnd(t0,m1)
20. MethodBegin(t0,m1,{t2})
21. MethodEnd(t0,m1)
22. Read(t0,x1,100)
23  Read(t0,x2,200)
24. Read(t0,y,300)
```

Figure 4: The *Account* benchmark. Existing precise dynamic algorithms (*e.g.*, Happens-Before) all report four false alarms due to missing events caused by the native method call *Thread.isAlive()* at line 11. By incorporating BarrierPair events (e_{18}-e_{21}) into the trace and formulating maximal causality constraints, RDIT reports no false alarm and detects the only true race (5,8).

is important to note here. It behaves as a join for $T1$ and $T2$, though it contains no *Thread.join()* statement. Specifically, Line 11 calls *Thread.isAlive()* to check if $T1$ and $T2$ have terminated or not. If not, the loop variable i will be set to 0 at line 12 and the loop will iterate again after *Thread.sleep()* at line 13. However, because *Thread.isAlive()* is a native method implemented through JNI, it is difficult to trace the computations inside the method. As a result, existing dynamic race detectors [28, 33, 56] all report false alarms at lines `(4,16)`, `(7,17)`, `(5,19)`, `(8,19)` due to missing events in this method, even though the race detection algorithms [26, 33, 57] they use are precise. In fact, the only true race in this benchmark is between lines `(5,8)` (because $T1$ and $T2$ can execute concurrently and there is no lock protecting these two statements), and this race may cause the error at line 20 to occur.

How RDIT Detects the Only True Race. Suppose we observe an execution of the program following an order denoted by the line numbers. The corresponding trace is shown in Figure 4(b). To avoid clutter, we omit read-only events to *accounts[i]*, and we refer to *accounts[i].Balance* as x_i, *BankTotal* as y, and the missing method *Thread.isAlive()* as m_1. To instantiate our event model presented in Section 3.1, variable initialization events, e_1:*Write*(t_0,y,0) and $e_{2,3}$:*Write*(t_0,x_i,0) (i=1, 2), and thread begin/end events $e_{6,12}$:*ThreadBegin*(t_i)/$e_{11,17}$:*ThreadEnd*(t_i) are also included in the trace. For lines 4-5 and 6-7, each line corresponds to two events (a *Read* and a *Write*).

The trace has two BarrierPairs (both triggered at line 11): (e_{18}:*MethodBegin*(t_0,m_1,{t_1}), e_{19}:*MethodEnd*(t_0,m_1)), and (e_{20}:*MethodBegin*(t_0,m_1,{t_2}), e_{21}:*MethodEnd*(t_0,m_1)).

From the trace, the constraints formulated by RDIT are shown in Figure 4(c). Let O_i refer to the order variable of e_i. The *BarrierPair* constraints are written as $O_{11} < O_{18} \wedge O_{17} < O_{20}$, because the two BarrierPairs have overlapping reachable addresses, t_1 and t_2, with the two *ThreadEnd* events e_{11} and e_{17}, respectively. The *Program Order* constraints and *Fork Join* constraints are similarly constructed following Algorithm 2. The *Locking* constraints are empty because the trace contains no lock. The *Read Consistency* constraints are encoded together with the race constraint for each conflicting event pair from different threads to simplify our presentation (by avoiding redundant formulas). For instance, for the event pair e_9:*Read*(t_1,y,0) and e_{16}:*Write*(t_2,y,300), the constraints are written as $O_9 = O_{16} \wedge O_{10} < O_{15}$, because e_{16} depends on the read e_{15}:*Read*(t_2,y,100), which must happen after the write e_{10}:*Write*(t_1,y,100) that sets y to 100. Similarly, for (e_{10}, e_{24}:*Read*(t_0,y,300)), the constraints are written as $O_{10} = O_{24} \wedge O_8 < O_{22} \wedge O_{14} < O_{23} \wedge O_9 < O_{16}$, because e_{24} depends on two reads, e_{22}:*Write*(t_0,x_1,100) and e_{23}:*Write*(t_1,x_2,200), which must happen after the two writes, e_8:*Write*(t_1,x_1,100) and e_{14}:*Write*(t_2,x_2,200), respectively, to get the valid value.

Conjoining all these constraints, we invoke an SMT solver (Z3 [19] in our implementation) to compute a solution. Because all unknown variables in the constraints are integers, and for the race constraint $O_a = O_b$ we can replace O_a by O_b, the constraints can be efficiently solved by Integer Difference Logic (IDL). For (e_{10},e_{15}), the solver returns a solution, so lines (5,8) are a true race. However, for all the other four conflicting event pairs at lines `(4,16)`, `(7,17)`, `(5,19)`, `(8,19)`, the solver reports that no solution exists. Therefore, all of them are false alarms.

We note that another way to avoid the false alarms in this example is to properly model the happens-before effects of the *Thread.isAlive()* method at the language level. However, this would require nontrivial manual effort (*e.g.*, *Thread.isAlive()* does not incur the usual happens-before but can be executed anytime regardless of the thread's termination) and hence may not scale to large programs. In contrast, our technique is automatic and does not require such manual modeling of happens-before effects.

5. Evaluation

We have implemented the RDIT algorithm in RVPredict [33], a recent race detector for multithreaded Java programs based on ASM [1] and Z3 [19]. RVPredict allows us to perform a direct comparison between RDIT and three existing precise algorithms – *Happens-Before* (HB) [40], *Causally-Precedes* (CP) [57], *Maximal-Causality* (MC) [33]), all of which have been implemented in RVPredict. In addition, we have implemented the simple barrier (SB) approach described in Section 2 and compared it with the other approaches. RDIT aims to be useful for dynamic race detection in real-world programs where missing events are common due to instrumentation challenges and performance considerations.

In this section, we focus on answering two questions:

(1) **Race detection effectiveness**. How effective is RDIT in preventing false alarms and in detecting true races in real-world programs? While guaranteeing no false alarm, would RDIT also seriously limit the race detection ability?

(2) **Runtime performance**. How much performance improvement (or slowdown) overall does RDIT introduce for handling missing methods? What is the runtime overhead for capturing BarrierPair events?

5.1 Evaluation Methodology

We compare RDIT with HB, CP, MC and SB on seven real-world large multithreaded applications, including *Eclipse*, *Apache Derby*, *Jigsaw*, *Sunflow*, *Xalan*, and *Floodlight*. Most of these applications are collected from previous studies [33]. Table 1 summarizes these benchmarks and metrics of the corresponding traces. To perform a fair comparison, for each benchmark, we collect one trace and run different techniques on the same trace. Because all these traces are long (*e.g.,* most containing millions of events), we use the same windowing strategy developed in RVPredict [33] (*i.e.*, cutting the traces into smaller chunks, each with 10K events by default), so that all techniques can finish within a reasonable time (one hour). For each trace, we compare the total number of reported races and false alarms by each technique. For computing the reachable memory addresses of missing methods, we also compare the results with and without using the optimization in Section 3.2.1.

App	LoC	Thrd	Evnt	RW	Sync	BP
ftpserver	32K	12	48K	34K	3K	5K
floodlight	68K	9	58K	33K	3K	11K
jigsaw	101K	12	3.4M	3M	3K	205K
sunflow	109K	9	15.6M	11M	0.6K	2.3M
xalan	180K	9	15M	13M	62K	2M
derby	302K	3	2.2M	1.8M	64K	196K
eclipse	560K	10	16.6M	8.2M	1.4M	3.5M

Table 1: Benchmarks and traces. The total size of all benchmarks is over 1.3MLOC. *Thrd*: the number of threads; *Evnt*: events; *RW*: reads/writes; *Sync*: synchronizations; and *BP*: BarrierPairs in the trace. The BarrierPairs are set to all method calls that contain synchronizations.

One challenge in our evaluation is how to determine if a reported race is a false alarm. For evaluation purpose, we first collect a set of true races (*i.e.*, *ground truth*) for each benchmark by running MC on a full trace (except excluding certain JDK libraries in `java.*`,`javax.*`,`com.*`,`sun.*`, due to instrumentation limitations). In case the excluded JDK libraries introduce synchronization that leads to false alarms reported by MC, we also cross validate these races by running RDIT (with those excluded libraries set to missing methods). We ensure that the same set of races are reported by both MC and RDIT.

We then further exclude events in each trace to simulate missing events. Finally, the races reported by each technique on the remaining trace are compared with the ground truth and those not in the ground truth are classified as false alarms.

We simulate missing events in two different ways. First, we randomly exclude certain classes and packages from the trace to simulate the random condition of missing events, and we consider all method calls to the excluded classes as BarrierPairs. Second, we exclude methods that contain synchronization events (*e.g.*, *ThreadFork*/*ThreadJoin*, *Lock*/*Unlock* and *Read*/*Write* to volatile variables) and consider calls to them as BarrierPairs. The second way simulates an extreme situation where all synchronizations are missed in the trace. We use this scenario to obtain a set of false alarms (by the other competing techniques) and to show that RDIT is able to eliminate all the false alarms. Nevertheless, it is a valid situation that may happen in practice.

We evaluate the runtime performance of RDIT with the *Xalan* benchmark. We choose *Xalan* because it is CPU intensitive. We measure the execution time and memory consumption of the generated trace by RDIT and compare the performance data between several different configurations: before and after excluding certain methods from common

#MissingClass	Detected Races (#Total/#False Positive)				
	HB	CP	MC	**RDIT**	**SB**
0	2(**0**)	2(**0**)	9(**0**)	9(**0**)	9(**0**)
20	32(**30**)	34(**30**)	48(**39**)	8(**0**)	2(**0**)
30	55(**51**)	58(**52**)	69(**60**)	5(**0**)	2(**0**)
40	61(**57**)	66(**62**)	83(**74**)	4(**0**)	1(**0**)

Table 2: Results on *Eclipse* by randomly excluding classes.

JDK libraries and *Xalan* packages, with and without capturing BarrierPairs, and with and without using the reachable address optimization.

All experiments were conducted on an 8-processor 32-core 3.6GHz Intel i7 Linux with 8GB memory and JDK 1.8 8GB heap space. All data were averaged over three runs.

5.2 Race Detection Results

Table 2 reports the results by randomly excluding the executed classes in *Eclipse* (the other benchmarks have similar results), and Table 3 summarizes the results of race detection by excluding synchronization methods. For all the seven benchmarks, RDIT detects a total of 85 races, all of which are true races. Note that all reported races have a unique signature (*i.e.*, static program locations, not dynamic race pairs). Comparatively, SB only detects 23 races (though all of them are true races), and the other three techniques (HB, CP, and MC) report 149, 149, and 213 false alarms, respectively, after excluding the methods containing synchronizations. In Table 3, HB and CP report the same set of races for all benchmarks. The reason is that the two algorithms become equivalent when *Lock*/*Unlock* synchronizations are excluded. For the true races, HB and CP detect a total of 58, and MC detects 112. Surprisingly, even with missing methods, RDIT detects 27 more true races than HB and CP, due to the power of the maximal causality model. For MC, although it detects 27 (112 vs 85) more true races than RDIT, it also reports an excessive number (213) of false alarms. Moreover, the results are consistent with and without using the reachable address computing optimization described in Section 3.2.1, because the optimization condition always holds in these benchmarks. We next discuss the results of several interesting benchmarks.

Eclipse. This benchmark contains JDT tests for the Eclipse IDE (collected from Dacapo [11]). There are 9 true races detected on the full trace with ten threads and 16.6M events. When no class is excluded, both HB and CP report 2 races, and MC, SB, RDIT all report 9 races, with no false alarm. However, when 20 random classes (may not necessarily contain synchronization methods) are excluded, HB reports 32 races, but 30 of them are false alarms; CP is able to detect 4 true races, but also reports 30 false alarms; MC still reports 9 true races, but 39 false alarms. Nevertheless, RDIT reports 8 races, all of which are true races. In contrast, SB detects only 2 races and misses 7, because in SB missing events are all serialized, no matter they may introduce synchronizations or not. RDIT misses only one race due to the fact that the BarrierPair model approximates the synchronization behavior of missing methods, which adds additional happens-before edges. As more classes are excluded, HB, CP and MC all report more false alarms, but RDIT still reports no false alarm, albeit fewer true races. When 40 classes are excluded, the three other techniques report 57, 62 and 74 false alarms, respectively, whereas RDIT only reports 4 of the 9 true races and SB reports only one of them. RDIT misses 5 true races in total due to its conservative analysis.

When all synchronization methods are excluded, all the three previous precise techniques (HB, CP, and MC) report a large number of false alarms (68, 68, and 81, respectively), whereas RDIT and SB consistently report 4 and 1 true races only, respectively. The reason for the large number of false alarms is that most conflicting events in *Eclipse* are properly protected by synchronizations. When synchronizations are excluded, HB, CP and MC will all report false alarms.

Floodlight. This benchmark is an open source software-defined networking (SDN) controller. The trace corresponds to an execution of *Floodlight* starting up until it is ready to accept network requests. It contains 9 threads and 58K events. There are five true races identified by MC and RDIT when no synchronization is excluded. After excluding all synchronizations, HB and CP report 16 races but none of them is true, and MC reports 24 races but 19 of them are false alarms. In contrast, RDIT reports 4 races all of which are true races, and SB only reports one of them. RDIT misses one true race because the two race events are both inside a BarrierPair method.

Jigsaw. This benchmark is a web server application that has been studied frequently in previous work [31, 33, 57]. There are 8 true races detected on the full trace containing 12 threads and 3.4M events. RDIT only detects two true races because all the other six races are inside the missing methods (excluded in our experiment because they contain synchronization events), whereas SB does not detect any race due to the serialization of missing events.

Xalan. This benchmark transforms XML documents into HTML using multiple threads. It contains a large number of true races (56 detected on the full trace). Interestingly, RDIT is able to detect almost all (52) of the true races – a lot more than that detected by SB (5) and by HB and CP (22), though HB and CP report only two false alarms. MC, on the other hand, detects all the true races but also reports one false alarm. The number of false alarms is small because, unlike that in *Eclipse*, the majority of race events do not occur in synchronized methods.

5.3 Runtime Performance

Table 4 reports the runtime performance results. Overall, the runtime overhead of RDIT for capturing BarrierPairs in the

Application	#True Races					#False Positives					RDIT #Missed Races		
	HB	CP	MC	RDIT	SB	HB	CP	MC	RDIT	SB	Total	Exclusion	Conservativeness
ftpserver	20	20	24	**15**	12	17	17	32	**0**	0	9	2	**7**
floodlight	0	0	5	**4**	2	16	16	19	**0**	0	1	1	**0**
jigsaw	5	5	8	**2**	0	36	36	47	**0**	0	6	6	**0**
sunflow	1	1	1	**1**	0	5	5	5	**0**	0	0	0	**0**
xalan	20	20	56	**52**	5	2	2	1	**0**	0	4	0	**4**
derby	8	8	9	**7**	3	5	5	28	**0**	0	2	0	**2**
eclipse	4	4	9	**4**	1	68	68	81	**0**	0	5	0	**5**
Total:	58	58	112	**85**	23	149	149	213	**0**	0	27	9	**18**

Table 3: Results after excluding all methods that contain synchronization events. For each benchmark, the same incomplete trace is used in all the five techniques: Happens-Before (HB), Causal-Precedes (CP), Maximal-Causality (MC), Simple-Barrier (SB), and RDIT. Columns 2-11 report the number of true races and false positives reported by each technique on the incomplete traces. Columns 12-14 report the total number of missed true races by RDIT, those due to code exclusion, and those due to the analysis conservativeness, respectively. RDIT detects a total of 85 true races with no false positives, SB detects only 23, while the other three techniques report hundreds of false positives. Compared to MC, RDIT misses 27 races, 18 of them due to the analysis conservativeness.

Excluded	Log Size			Time		
	No-BP	**BP**	**BP+Opt**	**No-BP**	**BP**	**BP+Opt**
❶	1.3G	3.2G(**+1.9G**)	1.4G(**+0.1G**)	16.5s	44.2s(**+168%**)	18.6s(**+13%**)
❶+❷	1.3G	3.1G(**+1.8G**)	1.4G(**+0.1G**)	15.8s	39s(**+147%**)	16.6s(**+5%**)
❶+❷+❸	1.1G	2.5G(**+1.4G**)	1.2G(**+0.1G**)	13.6s	31.8s(**+134%**)	15.3s(**+12%**)
❶+❷+❹	652M	1.3G(**+0.7G**)	711M(**+60M**)	6.4s	11.3s(**+77%**)	6.7s(**+4%**)
❶+❷+❸+❹	548M	990M(**+0.4G**)	598M(**+50M**)	4.8s	8.6s(**+79%**)	5.2s(**+8%**)
❶+❷+❸+❹+❺	489M	823M(**+0.3G**)	537M(**+50M**)	4.3s	7.1s(**+65%**)	4.5s(**+7%**)
❶ common JDK libraries (`java.*`,`javax.*`,`com.*`,`sun.*`); ❷ `org.dacapo.harness.*` ❸ `org.apache.xpath.*`; ❹ `org.apache.xml.*`; ❺ `org.apache.xalan.*`.						

Table 4: Runtime performance of RDIT on *Xalan* when missing methods in certain packages, with and without capturing BarrierPairs, and with and without using the reachable address optimization. The native execution of *Xalan* takes 0.36s.

Xalan benchmark ranges from 4%-13% with the reachable address optimization (recall Section 3.2.1) and 65%-168% without. The space overhead for trace storage (mostly for storing the reachable addresses computed from the object tree) ranges between 0.3-1.9GB without the optimization, but is less than 100MB with the optimization. With the optimization, the runtime overhead for capturing BarrierPairs is almost negligible compared to that of tracing all the other events (*e.g.*, *Read*/*Write*). For instance, the native execution of *Xalan* without any logging takes only 0.36s, while the tracing execution excluding only the common JDK libraries takes 16.5s, more than 45X overhead. Moreover, because capturing BarrierPairs completely avoids the need to log events inside the missing methods, the overall performance improvement of RDIT is significant. For example, when excluding the packages `org.dacapo.harness` and `org.apache.xml`, the execution time is reduced from 18.6s to 6.7s with the optimization and from 44.2s to 11.3s without, and the trace size reduced from 1.4GB to 711MB with the optimization and from 3.2GB to 1.3GB without. When further excluding the package `org.apache.xpath`, the execution time is reduced to 4.8s, and trace size reduced to 598M, with the optimization.

Our performance results strongly support the application of RDIT in practice where logging certain methods or libraries is expensive, or the developer is only interested in certain specific code regions. For instance in *Xalan*, logging `org.apache.xml` is expensive but the developer may only be interested in detecting races in the package `org.apache.xalan`. The developer can then instruct RDIT to log only events in `org.apache.xalan` and model all method calls to `org.apache.xml` as BarrierPairs.

6. Related Work

Researchers have proposed a large number of race detection techniques, both static [47, 58] and dynamic [10, 22, 26], targeting different types of software [21, 23, 52, 60], memory models [16, 17, 27], application domains [8, 44], and at various stages of software development [24, 43]. Our technique is distinguished in that it addresses the practical problem of missing events. The BarrierPair model bridges the gap between existing precise race detection algorithms and the practical challenges in capturing a full execution trace, enabling dynamic race detectors to precisely detect races from incomplete traces with maximal detection capability.

Runtime Pruning False Alarms. Researchers have proposed several runtime validation techniques [9, 34, 55] to improve the accuracy of race detection. These techniques take a set of potential races as input and execute the program again attempting to simulate the schedules necessary to induce the race. If the conditions to reproduce the race are not met, the race is considered false and not reported. While these techniques can prune false alarms, they require multiple runs of the program, and may suffer from livelocks and hence miss true races.

Low-level Race Detection. LARD [60] considers the problem for low-level race detection that the JVM can emit extra events that may introduce false HB edges not present at the language level, and it proposes a method to remove those false HB edges by excluding the JVM events. This problem is opposite to the missing event problem addressed by RDIT in the sense that, the missing events may cause true HB edges to be missed, and RDIT adds those true HB edge back by approximating the missing events.

Sampling-based Race Detection. To improve runtime performance, several online sampling techniques [14, 42, 61] have been proposed to scale dynamic race detection to long running programs. LiteRace [42], Pacer [14], and RaceTrack [61] all use sampling to reduce the tracing overhead and may achieve negligible runtime slowdown, at the cost of reduced race detection ratio. However, when certain code is not instrumented, their algorithms may not be precise and do not guarantee the absence of false alarms.

Systematic State-space Exploration. Complementary to predictive race detection, which is based on a single trace, researchers have proposed several techniques [15, 29–31, 45, 46] to explore the program's state-space by re-executing the program multiple times following different schedules. The schedules are systematically explored with context bounding [45], probabilistic priorities [15, 46], redundancy reduction [29–31], etc. These techniques can be combined with RDIT to detect more races from multiple traces instead of a single trace.

Deterministic Execution. In contrast to detecting races under random schedules, deterministic execution techniques pioneered by DMP [20], DThreads [41], and Parrot [18] aim to make the execution deterministic by default, such that data races either manifest themselves, or do not, on every execution. However, deterministic execution techniques may also suffer from the problem of missing events. The techniques may fail to enforce deterministic execution when certain events that introduce non-determinism are missed or not instrumented.

Symbolic Constraint Analysis. Numerous symbolic analyses [25, 31–33, 35, 59] based on logical constraint solving have been proposed to detect and diagnose concurrency bugs, including a few race detection techniques [33, 53]. Nevertheless, none of the previous analyses considered the practical problem of missing events.

Race Detection for Relaxed Memory Models. Races in systems with weak consistency models can be more difficult to understand. Several approaches [16, 17, 27] have been proposed to detect races under relaxed memory models, such as TSO, PSO, and Java memory models. In this work we have focused on sequential consistency only. Nevertheless, the BarrierPair model can be extended to relaxed memory models by re-formulating the thread causality constraints, such as what is done in [36] for verifying concurrent programs under TSO and PSO.

Harmful and Benign Races. Not all true races may be considered harmful by developers. A few techniques [24, 37, 48] have been proposed to automatically classify benign and harmful races from true races through replay [48], symbolic analysis [37], or heuristics [24]. RDIT does not distinguish benign and harmful races. However, we note that races that look benign may still be harmful or become harmful, due to subtleties in memory models [6], compiler transformations, or hardware optimizations [12, 13].

7. Conclusion

We have presented a new technique, RDIT, that enhances the existing body of dynamic race detection by allowing events to be missed in the trace through missing methods. Powered by a sound BarrierPair model and a constraint encoding of maximal thread causality, RDIT is both precise and maximal for the sequential consistent memory model, that it does not report any false alarms and it detects a maximal set of true races from the observed incomplete trace. We have shown empirically that RDIT detects dozens of true races in a variety of real-world large multithreaded applications with zero false alarms, whereas existing precise algorithms report many false alarms due to missing events. We believe that RDIT will be valuable for the development of precise dynamic race detection tools in practice.

8. Acknowledgements

The authors would like to thank Daniel Jiménez and the anonymous reviewers for their constructive comments on earlier versions of this paper. This research is supported by faculty start-up funds from Texas A&M University, a Google Faculty Research Award, and NSF award CCF-1552935.

References

[1] ASM bytecode analysis framework. `http://asm.ow2.org`.

[2] Java native interface specification. `http://docs.oracle.com/javase/7/docs/technotes/guides/jni/\spec/jniTOC.html/`.

[3] T. J. Watson Libraries for Analysis (WALA). `http://wala.sourceforge.net/`.

[4] ThreadSanitizer Documentation. `http://clang.llvm.org/docs/ThreadSanitizer.html`.

[5] ThreadSanitizer issue #646. `https://github.com/google/sanitizers/issues/646`.

[6] S. V. Adve and H.-J. Boehm. Memory models: a case for rethinking parallel languages and hardware. *Communications of the ACM*, 53(8):90–101, 2010.

[7] T. S. architecture manual. Version 9. *SPARC International, Inc.* 1994.

[8] P. Bielik, V. Raychev, and M. Vechev. Scalable race detection for android applications. In *ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, 2015.

[9] S. Biswas, M. Zhang, and M. D. Bond. Lightweight data race detection for production runs.

[10] S. Biswas, M. Zhang, M. D. Bond, and B. Lucia. Valor: Efficient, software-only region conflict exceptions. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, 2015.

[11] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, 2006.

[12] H.-J. Boehm. How to miscompile programs with "benign" data races. In *Proceedings of the 3rd USENIX Conference on Hot Topic in Parallelism*, pages 3–3, 2011.

[13] H.-J. Boehm. Position paper: Nondeterminism is unavoidable, but data races are pure evil. In *RACES*, pages 9–14, 2012.

[14] 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*, pages 255–268, 2010.

[15] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 167–178, 2010.

[16] S. Burckhardt and M. Musuvathi. Effective program verification for relaxed memory models. In *International Conference on Computer Aided Verification*, 2008.

[17] J. Burnim, K. Sen, and C. Stergiou. Testing concurrent programs on relaxed memory models. In *ACM International Symposium on Software Testing and Analysis*, pages 122–132, 2011.

[18] H. Cui, J. Simsa, Y.-H. Lin, H. Li, B. Blum, X. Xu, J. Yang, G. A. Gibson, and R. E. Bryant. Parrot: a practical runtime for deterministic, stable, and reliable threads. In *ACM Symposium on Operating Systems Principles*, 2013.

[19] L. De Moura and N. Bjørner. Z3: an efficient SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[20] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. Dmp: deterministic shared memory multi-processing. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2009.

[21] D. Dimitrov, V. Raychev, M. Vechev, and E. Koskinen. Commutativity race detection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 305–315, 2014.

[22] L. Effinger-Dean, B. Lucia, L. Ceze, D. Grossman, and H.-J. Boehm. IFRit: Interference-free regions for dynamic data-race detection. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 467–484, 2012.

[23] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: a race and transaction-aware Java runtime. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007.

[24] J. Erickson, M. Musuvathi, sebastian burckhardt, and kirk olynyk. Effective data-race detection for the kernel. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 1–16, 2010.

[25] A. Farzan, P. Madhusudan, N. Razavi, and F. Sorrentino. Predicting null-pointer dereferences in concurrent programs. In *Joint European Software Engineering Conference and ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 47:1–47:11, 2012.

[26] C. Flanagan and S. N. Freund. FastTrack: efficient and precise dynamic race detection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 121–133, 2009.

[27] C. Flanagan and S. N. Freund. Adversarial memory for detecting destructive races. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 244–254, 2010.

[28] C. Flanagan and S. N. Freund. The roadrunner dynamic analysis framework for concurrent programs. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 1–8, 2010.

[29] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 110–121, 2005.

[30] P. Godefroid. Model checking for programming languages using verisoft. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 174–186, 1997.

[31] J. Huang. Stateless model checking concurrent programs with maximal causality reduction. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 165–174, 2015.

[32] J. Huang, Q. Luo, and G. Rosu. GPredict: Generic Predictive Concurrency Analysis. In *International Conference on Soft-*

ware Engineering, 2015.

[33] J. Huang, P. O. Meredith, and G. Rosu. Maximal sound predictive race detection with control flow abstraction. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 337–348, 2014.

[34] J. Huang and C. Zhang. PECAN: Persuasive Prediction of Concurrency Access Anomalies. In *ACM International Symposium on Software Testing and Analysis*, pages 144–154, 2011.

[35] J. Huang, C. Zhang, and J. Dolby. CLAP: Recording local executions to reproduce concurrency failures. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 141–152, 2013.

[36] S. Huang and J. Huang. Maximal causality reduction for tso and pso. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, 2016.

[37] B. Kasikci, C. Zamfir, and G. Candea. Data Races vs. Data Race Bugs: Telling the Difference with Portend. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012.

[38] Z. Lai, S. C. Cheung, and W. K. Chan. Detecting atomic-set serializability violations in multithreaded programs through active randomized testing. In *International Conference on Software Engineering*, pages 235–244, 2010.

[39] P. Lam, E. Bodden, and L. Hendren. The soot framework for Java program analysis: a retrospective, 2011.

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

[41] T. Liu, C. Curtsinger, and E. D. Berger. Dthreads: Efficient deterministic multithreading. In *ACM Symposium on Operating Systems Principles*, pages 327–336, 2011.

[42] 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*, pages 134–143, 2009.

[43] N. D. Matsakis and T. R. Gross. A time-aware type system for data-race protection and guaranteed initialization. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, 2010.

[44] J. Miserez, P. Bielik, A. El-Hassany, L. Vanbever, and M. Vechev. Sdnracer: Detecting concurrency violations in software-defined networks. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, pages 22:1–22:7, 2015.

[45] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 267–280, 2008.

[46] S. Nagarakatte, S. Burckhardt, M. M. Martin, and M. Musuvathi. Multicore acceleration of priority-based schedulers for concurrency bug detection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 543–554, 2012.

[47] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 308–319, 2006.

[48] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically classifying benign and harmful data racesallusing replay analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 22–31, 2007.

[49] R. H. B. Netzer and B. P. Miller. What are race conditions: Some issues and formalizations. *LOPLAS*, 1992.

[50] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2003.

[51] A. Rajagopalan and J. Huang. RDIT: Race detection from incomplete traces. In *Joint European Software Engineering Conference and ACM SIGSOFT Symposium on Foundations of Software Engineering, New Ideas and Emerging Results*, 2015.

[52] V. Raychev, M. Vechev, and M. Sridharan. Effective race detection for event-driven programs. In *ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, pages 151–166, 2013.

[53] M. Said, C. Wang, Z. Yang, and K. Sakallah. Generating data race witnesses by an SMT-based analysis. In *International Conference on NASA Formal Methods*, pages 313–327, 2011.

[54] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. In *ACM Symposium on Operating Systems Principles*, pages 27–37, 1997.

[55] K. Sen. Race directed random testing of concurrent programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 11–21, 2008.

[56] K. Serebryany and T. Iskhodzhanov. Threadsanitizer: data race detection in practice. In *WBIA*, 2009.

[57] Y. Smaragdakis, J. Evans, C. Sadowski, J. Yi, and C. Flanagan. Sound predictive race detection in polynomial time. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 387–400, 2012.

[58] J. W. Voung, R. Jhala, and S. Lerner. Relay: static race detection on millions of lines of code. ESEC-Joint European Software Engineering Conference and ACM SIGSOFT Symposium on Foundations of Software Engineering, 2007.

[59] C. Wang, S. Kundu, M. K. Ganai, and A. Gupta. Symbolic predictive analysis for concurrent programs. In *FM*, 2009.

[60] B. P. Wood, L. Ceze, and D. Grossman. Low-level detection of language-level data races with LARD. In *Architectural Support for Programming Languages and Operating Systems,*, pages 671–686, 2014.

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