FAST AND PRECISE ON-THE-FLY DATA RACE DETECTION

A Thesis

by

ARUN KRISHNAKUMAR RAJAGOPALAN

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

May 2016

Major Subject: Computer Science

ABSTRACT

While concurrent programming is quickly gaining popularity lately, developing bug-free programs is still challenging. Although developers have a wide choice of race detection tools available, we have found that the majority of these techniques do not scale well and developers are often forced to balance precision with speed. Additionally, various practical issues force even precise race detectors to produce spurious warnings, defeating their purpose and burdening their users. We design and implement a novel race detection technique that is both fast and precise, even in the face of missing program source information. Towards this goal, we have developed two separate tools, TREE and RDIT, that respectively improve performance and precision over existing techniques.

TREE, implemented in the RoadRunner framework, acts as a filter and sends through only those events that might add value to race detection while eliminating those events which are deemed redundant for this purpose. All the while, removing these redundant events does not affect its race detection capability. We have evaluated TREE against a whole set of standard benchmarks, including two large real-world applications. We have found that there exists a significant number of redundant events in all these applications and on an average, TREE saves somewhere between 15-25% of analysis time as compared to the state-of-the-art techniques.

Meanwhile, our next tool, RDIT, is able to precisely detect races in programs with incomplete source information, generating no false positives. RDIT is also maximal in the sense that it detects a maximal set of true races from the observed incomplete trace. It is underpinned by a sound BarrierPair model that abstracts away the missing events by capturing the invocation data of their enclosing methods. By making the least conservative assumption that a missing method introduces synchronization only when its invocation data overlaps with other missing methods, and by formulating maximal thread causality as a set of logical constraints, RDIT guarantees to precisely detect races with maximal capability. We tested RDIT against seven real-world large concurrent systems and have detected dozens of true races with zero false alarm. Comparatively, existing algorithms such as Happens-Before, Causal-Precede, and Maximal-Causality, which are all known to be precise, were observed reporting hundreds of false alarms due to trace incompleteness.

ACKNOWLEDGEMENTS

Firstly, I would like to express my sincere gratitude to Dr. Jeff Huang for his continued support towards my Master's study and related research, for his patience, motivation, and immense knowledge. His guidance during the research and writing of this thesis was invaluable. It was truly a pleasure working with him.

Besides my advisor, I would like to thank the rest of my thesis committee: Dr. Jennifer Welch and Dr. Jim Ji for their insightful comments and encouragement. My sincere thanks also goes to Dr. Dmitri Loguinov, Dr. Lawrence Rauchwerger, and all my other professors who imparted valuable knowledge and involved me in exciting project work that greatly helped my research.

I thank my fellow labmate Anirudh and my roommates for the stimulating discussions, the sleepless nights working before deadlines and all the fun we've had over the past two years. Last but not the least, I would like to thank my family: my parents, my brother and my grandparents for their love and support throughout my life.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

Concurrent programming is quickly becoming very popular due to inherent limitations of hardware. It is more economically viable and practical nowadays to have several moderately performing processor cores instead of a single high performing core, as in the early days of computing. Such multi-core processors are finding increasing popularity in a range of devices, from the powerful servers serving the world's populace to the ubiquitous smartphones that are carried in people's pockets everyday. They are even being employed in critical infrastructure applications such as health care, public utilities, defense etc. However, as it turns out, developing bug-free concurrent programs is quite hard due to the non-linear execution patterns that are typical of these programs. A few notable instances of concurrency bugs that caused significant damage and monetary loss include the 1985 Therac-25 medical radiation device malfunction [59] that lead to the death of 5 people and injured several others from overdose, the 2003 Northeast Blackout [41] that cost the government an estimated $4 billion and more recently, NASDAQ's Facebook IPO glitch [48] that resulted in a loss of $13 million to investors. It is evident then, that verifying the correctness of such concurrency programs is of paramount importance.

Researchers have proposed a wide variety of tools to aid in detection of concurrency errors [2, 3, 7, 16–18, 21, 26, 28, 34, 40, 47, 50, 54]. Among the various concurrency errors, data-races are a particularly challenging category since they appear non-deterministically and are often triggered by just the right combination of various conditions [41, 59]. 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 notoriously challenging. Making it worse is fact 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 can transform it into harmful bugs [1, 5, 6].

Tools to detect data races are of two kinds – static analysis based and dynamic analysis based. Tools that use static analysis form constrains on the data flow and use a solver to try and find data races hidden in the program. Dynamic data race tools, on the hand, analyze the program execution trace and report races from the observed execution. Although we now have a plethora of tools to help in detecting data-race bugs, eliminating races in real-world programs remains impractical. There are three primary factors that determine the effectiveness of a race detection tool:

1. number of races detected,

2. performance and

3. number of false alarms

In this thesis, we focus on addressing and improving the last two factors, that of performance and precision:

- **Improved performance:** Developers usually have to balance two factors when hunting for data-race bugs - whether to focus on precision or on performance. While precise race detectors reduce developer burden by detecting only true races, they are usually prohibitively expensive to use for large scale programs. Thus, developers end up choosing a less precise but faster approach and then verify the bugs manually. Improving performance and scalability concerns of these precise race detectors would enable developers to avoid false positives and improve their turnaround time.

- **Improved precision:** As we shall see in subsequent sections, even precise race detectors can generate false positives when trace information is incomplete. Our goal is to focus on elimination of these false positives in all conditions so that users of our tool are guaranteed that every race detected in a true race.

1.1 Redundancy Elimination

There are, in general, three broad categories of dynamic race tools – a) LockSet [50] based, b) Happens-Before [32] based and c) hybrid approaches [40] based on the LockSet and HB. LockSet based techniques [37, 53] deem a race to have occurred if two or more threads access a shared memory location without holding a common lock. As such, LockSet based techniques are very fast and have less overhead. Unfortunately, they tend to produce a lot of false alarms since perfectly valid race-free sections of execution can violate the locking principle. HB based tools [21, 53] are precise, typically producing no false positives. However, they run slower than the LockSet based approaches since they rely on expensive vector clock [42] computations.

In recent years, the performance of Happens-Before (HB) based race detectors has greatly improved thanks to techniques such as FastTrack [21]. For many small-scale programs their performance is now close to that of LockSet based tools [50]. This is primarily due to recent advancements that have greatly cut down the size of the vector clocks from $O(N_{threads})$ to almost $O(1)$, where $N_{threads}$ is the number of threads. However, we still have difficulties scaling these tools to large software applications. This is because these tools must still maintain state and check races for each memory access, which is in $O(N_{events})$, *i.e.*, the number of memory accesses. The $O(N_{events})$ time significantly impacts their scalability.

Previous research has proposed several techniques [7, 17, 22, 28, 34, 44, 60] to further improve performance of dynamic race detection tools by reducing N_{events}. However, most techniques are either incomplete or unsound, meaning that they either reduce the race detection capability of the tools or make them report false positives. For example, sampling-based techniques [7, 34] may lead to missing races, and static analysis based [17, 22, 44] may lead to false positives.

Our first contribution is a new technique, called TREE (**T**race **R**edundant **E**vent **E**limination), that significantly improves the native performance of dynamic race detectors, while still maintaining both precision and soundness. The key idea in our design stems from the fact that certain events in a typical program execution trace can be removed without affecting the capability and precision of the race detection tool. We term these events as *redundant.* Redundant events can be of two types (not mutually exclusive):

- Memory access events to addresses on which no races will be found.

- Memory access events to addresses on which *no new races* will be found.

Existing techniques such as [17, 44, 60] target specifically the first type of redundancy. For example, RaceTrack [60] adds more instrumentation to those regions that are more susceptible to races and lesser instrumentation to regions that are not. However, precisely analyzing the source code and determining such regions is hard and these tools may result in loss of precision or soundness.

In this work, we target the second type - *detecting only unique races.* We have evaluated the performance of TREE on a collection of benchmarks including two large real-world applications by running it as a pass before FastTrack [21](a precise race detector). TREE is able to remove more than half of the total events generated in these benchmarks. On the two real-world server applications, TREE identifies 35-70% redundant events, and improves the runtime performance of FastTrack by 15-25% with only a small memory cost. Memory performance could be further tuned by adjusting TREE parameters on an application by application basis by the user. More importantly, enabling TREE did not result in loss of any true data-races over what FastTrack might have reported.

1.2 Missing Trace Events

In real world usage of dynamic race detection tools, although they advertise themselves to be precise, we notice that they can generate false alarms in certain situations[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 enormous for real 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 [21]. 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 general problem of precisely identifying races is NP-hard [39]. To scale to large programs, existing techniques often overly approximate races. As discussed previously, the *LockSet* algorithm [50] implemented in state-of-the-art race detectors [37, 52] is known to be imprecise. Moreover, the challenge is rooted not only in the algorithmic complexity,

[1]We do not distinguish benign and harmful data races in this work. Any false positive race is considered as a false alarm. See Section 2 for more discussions on benign and harmful races.

but also from various practical issues:

1. **Performance slowdown.** Many applications or components are performance sensitive or have resource constraints such that they cannot tolerate too much runtime slowdown, otherwise they would fail to function properly. We may even desire to miss certain code in some scenarios for better performance. For example, when debugging code implementing a new feature, developers may be interested in detecting races in a specific code region and would want to skip the others.

2. **Unavailability of whole program.** Real-world systems often rely on external libraries, and/or are composed from layers of frameworks and extended by third-party plug-ins. These programs may even be loaded on the fly over the network. Analyzing the whole program to find all synchronizations is difficult or impossible.

3. **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. For example, built-in libraries (e.g., `java.*`) and code written in a lower level language (e.g., Java Native Interface (`JNI`) [29]).

In all these situations, we may end up missing vital program trace information. When only partial program information is available or observed, even existing precise algorithms (*i.e.,* Happens-Before (HB) [32], Causally-Precedes (CP) [54], Maximal-Causality (MC) [26]) become imprecise. For example, the classical HB algorithm [32] is precise, given that all critical events in the program execution are captured. However, this requirement can rarely be satisfied in practice, and HB-based tools [21, 52] tend to report many false alarms on real-world systems.

For our second contribution, we present a new dynamic race detection technique, called RDIT (**R**ace **D**etection from **I**ncomplete **T**races) that aims to fix this issue. RDIT is ***precise*** even when certain events in the program execution are not tracked or are missed. At the same time, RDIT is ***maximal*** such that it detects a maximal set of true races that can happen in all possible schedules inferred from the observed trace. RDIT is underpinned by a novel BarrierPair model of incomplete trace developed in this work. BarrierPair soundly abstracts the behavior of missing events through the invocation data of their enclosing methods. The BarrierPair model is *safe* since it conservatively 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 previous work [26], BarrierPair allows RDIT to formulate *maximal thread causality* as a series of quantifier-free first-order logic formulas. By solving the formulas together with data race constraints using an off-the-shelf SMT solver, RDIT is able to detect races precisely with maximal capability. In contrast to previous work [26], RDIT also allows arbitrary events to be missed in the trace without reporting any false alarm.

We anticipate that RDIT is 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 region (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 multi-threaded applications including *Eclipse IDE, Apache Derby Database* and *Floodlight SDN controller.* RDIT detected a total of 85 true races in these systems but zero false alarms. In contrast, existing precise algorithms (HB, CP, and MC) report hundreds of false alarms (149, 149, and 213, respectively) due to missing events in the trace. Moreover, RDIT improves the overall program performance significantly when used for capturing the incomplete trace in practice – capturing the BarrierPairs incurs only 4%-13% runtime overhead when a practically sound optimization is applied to compute reachable runtime data of missing methods compared to 65%-168% runtime overhead without the optimization.

In summary, our work in this thesis makes the following contributions:

- We provide a generalized model of a program trace and model the various types of redundant events that may be present in it.

- We develop a algorithm to remove these redundant events from the trace. TREE, which is an implementation of this algorithm, filters these redundant events before passing them on to the underlying race detector.

- We compare the performance of FastTrack + TREE to the performance of vanilla FastTrack on a variety of benchmarks. We show that for even the worst cases, TREE exhibits reasonable performance.

- Next, 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.

- RDIT is built on 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 RDIT on a range of real-world concurrent systems and demonstrate the race detection effectiveness and runtime performance.

The rest of the thesis is organized as follows – Section 2 discusses related work in this domain and how our techniques differ from existing literature. Section 3 introduces the concept of redundant events and shows how TREE is able to filter them. Section 4 builds on the need for improving the current trace model to account for missing events and how RDIT is able to precisely detect only true races. Next, Section 5 evaluates the performance of TREE and RDIT against a standard set of benchmarks and large real-world applications. Finally, Section 6 discusses plans for future work and our concluding thoughts.

2. RELATED WORK

Data race detection is a hot area of research at the moment. Researchers have proposed a large number of race detection techniques, both static [37, 57] and dynamic [3, 17, 21], targeting different types of software [16, 18, 47], memory models [8, 9, 20], application domains [2, 36], and at various stages of software development [8, 35].

TREE is targeted at addressing scalability problems faced by these tools when applied to real-world applications. Redundancy elimination by TREE is sound and generic in a way that makes it possible to apply it to any dynamic race detector in a plug-and-play approach to transparently improve their performance. In addition to performance gains through redundancy elimination, researchers have also looked at enhancements to the underlying race detection algorithms. Improved detection algorithms such as FastTrack [21], Eraser [50] have both greatly improved runtime performance from prior work. Building on top of these tools, several hybrid race detection tools [11, 40, 43] combine the performance of LockSet based techniques with the precision of HappensBefore based techniques. However, it is still challenging to develop new algorithms that can scale well. An easier, more practical approach is through redundancy elimination, which would work across all these tools. There are three families of techniques that aim to detect these redundancies, detailed below.

1. **Static Analysis based tools:** These tools [11, 17, 22, 45] reason about which memory accesses are redundant and mark such accesses statically. They eliminate those accesses that guaranteed to be race-free or would not result in generation of any new races. These tools however, struggle to properly analyze external library features and program constructs such as reflections, making

their analysis *unsound*. For example, RedCard [22] checks for races only on the first access to each shared address. While this approach does help in pruning the number of instrumented events, it also results in the loss of several real races. IFRit [17] on the other hand, identifies interference free regions of the program and reduces instrumentation in them.

2. **Online tools:** To improve runtime performance, several online sampling techniques [7, 34, 60] have been proposed to scale dynamic race detection to long running programs. LiteRace [34], Pacer [7], and RaceTrack [60] all use sampling to reduce the tracing overhead and may achieve negligible runtime slowdown, at the cost of reduced race detection ratio. RoadRunner [23] has an inbuilt thread-local pass that is supposed to speed up dynamic analysis tools by filtering memory addresses that are solely accessed by a single thread. However, we found that the design of this filter is unsound and results in missing races.

3. **Post-processing on trace:** TraceFilter [28] takes the generated trace file as input and performs redundancy elimination for offline predictive race analysis. In comparison to TraceFilter, TREE is able to deal with both intra-thread redundancy and inter-thread redundancy seamlessly using the same data structure to represent concurrency context[1] across threads. Moreover, TREE results in much better overall performance since redundancy elimination occurs at runtime in-contrast to post-processing.

Next, we look at the problem of false positives. Real-world program traces unfortunately exhibit plenty of missing trace events. These many be due to inherent limitations of the framework, or enterprising users who seek to extract maximum performance by targeting race detection to a small region. Our BarrierPair model,

[1]Discussed in Section 3.3

implemented in RDIT, bridges the gap between existing precise race detection algorithms and the challenges in capturing a full execution trace, enabling dynamic race detectors to precisely detect races from incomplete traces with maximal detection capability. Precise race detection has received considerable attention in the past few years and several approaches have been pursued.

1. **Runtime Pruning False Alarms:** Researchers have proposed several runtime validation techniques [28, 51] to improve 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.

2. **Deterministic Execution:** Complementary to race detection, a promising direction is to make the execution deterministic. This is pioneered by techniques such as DMP [15], DThreads [33], and Parrot [13]. These techniques ensure race conditions either manifest themselves, or do not, on every execution.

3. **Symbolic Constraint Analysis:** Researchers have proposed numerous analyses [19, 24–27, 58] based on logical constraint solving to detect and diagnose concurrency bugs, including a few race detection techniques [26, 49]. Nevertheless, none of the previous analysis considered the practical problem of missing events.

4. **Race Detection for Relaxed Memory Models:** Races in systems with memory consistency models can be more difficult to understand. Several approaches [8, 9, 20] have been proposed to detect races under relaxed memory

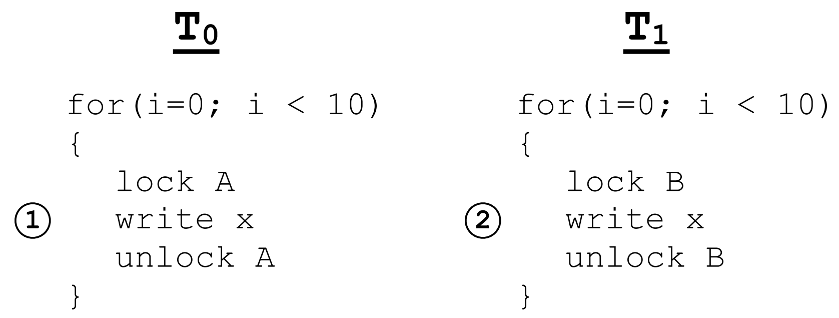
models, such as TSO, PSO, and Java memory models. In this work we have focused on sequential consistency only. Nevertheless, the RDIT technique can be extended to relaxed memory models, by relaxing the program order constraints.

5. **Harmful and Benign Races:** Not all true races may be considered harmful by developers. A few techniques [8, 30, 38] have been proposed to automatically classify benign and harmful races from true races through replay [38], symbolic analysis [30], or heuristics [8]. 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 [1], compiler transformations, or hardware optimizations [5, 6].

6. **Sampling-based Race Detection:** Although the sampling based tools mentioned previously [7, 34, 60] also allow missing events, their algorithms are not precise and do not guarantee the absence of false alarms.

3. A FAST RACE DETECTOR

Performance is a critical issue in any race detector. However, any performance improvements to the race detector tool must not come at the cost of precision or loss of race detection capability. To this end, we propose the concept of *trace redundancy* and demonstrate how eliminating it can significantly improve runtime performance of the race detection tool.



```
          T0                          T1

     for(i=0; i < 10)            for(i=0; i < 10)
     {                           {
       lock A                      lock B
 ①     write x               ②     write x
       unlock A                    unlock B
     }                           }
```

Figure 3.1: Program snippet exhibiting redundancy. There exists one real race between ① and ②.

In real-world program execution traces, we may observe several races between the same two lexical statements of a program. However, just a single pair is enough to alert the programmer to a race between these two statements - the other warnings are superfluous and can be ignored. More pressingly, these additional race checks negatively impact the runtime performance of the program since additional expensive vector clock operations must be performed without revealing any additional useful information to the programmer.

3.1 Examples

Consider the example in Figure 3.1. The two threads T_0 and T_1 both write to a shared address x. T_0 acquires lock A before writing to x, while T_1 acquires lock B. Because T_0 and T_1 do not share a common lock while writing to x, there exists a data-race between ① and ②. However, since the race exists inside a loop that runs 10 times, traditional HB-based detectors will check for races each time the event is generated. By filtering precisely those events that would not lead to any new *unique* race, the analysis can track lesser state and perform fewer race checks. This optimization is tremendous in modern day multi-threaded programs, as this type of redundancy is prevalent due to the single-process-multiple-data (SPMD) architectural design.

On the surface, this problem seems simple to solve by removing multiple events from the same program location. However, a treatment as such may remove important dependency information and produce incorrect results. Consider the following two approaches:

- *Approach 1:* Perform race checks for each lexical location just once.
- *Approach 2:* Filter events from the same lexical location if no synchronization event has been observed since the last time.

As we will elaborate below, *Approach 1* is unsound and *Approach 2* is overly limiting such that it cannot filter events across synchronization boundaries. We introduce a new concept – *concurrential equivalence* – that precisely and optimally captures redundant events. Specifically, two events are concurrentially equivalent if they have the same *concurrency context* – a history of *must happens-before* within the thread that performs the event. We show that for dynamic race detection, if there are two or more lexically-identical concurrentially equivalent events that access the same memory location, it is sufficient to keep only one of them for at most two threads, and safely drop all the others.

Prior work [28] observes a similar type of equivalency called *permutational redundancy* over events by the same thread, and develops an offline trace filtering technique to remove redundant events in the context of predictive concurrency analysis. However, this work makes two important advancements over prior work. First, the concept of concurrential equivalence is much more general than permutational redundancy – it characterizes both intra-thread and inter-thread event redundancy for race detection. Second, TREE is purely dynamic without any static analysis and is general enough to be applicable to HB-based, LockSet-based, or hybrid race detectors.

A redundant event in a trace is considered to be any event whose exclusion from the trace does not lead to any missed races or false alarms. We can categorize these redundant events into two types: intra-thread redundancy, and inter-thread redundancy.

3.1.1 Intra-thread Redundancy

This type of redundancy appears between events from the same program locations accessed by the same thread. Consider a program in Figure 3.2. Two methods *m_read*() and *m_write*() are called to read and write to the address x. The main loop runs 10 times, the first 5 times acquire lock A before writing to x and the next 5 acquire lock B before writing to x.

Assume the code above is executed by the main thread T_0. If there exists a second thread T_1 that also writes to x, there are two possible lexical locations where races might occur: the read at ⑥, and the write at ⑦. One simple strategy for removing these redundant events would be to check races at each lexical location just once.

```
    for(i=0; i<10; i++){
①      m_read()
②      m_read()
       if(i < 5) {
          lock(A)
③         m_write()
          unlock(A)
       } else {
          lock(B)
④         m_write()
          unlock(B)
       }
⑤      m_write()
    }
```

```
  m_read()
⑥ read x

  m_write()
⑦ write x
```

Figure 3.2: Intra-thread event redundancy on ⑥ and ⑦.

However, this would result in missed races. To see why, let us imagine that T_1 writes to x after acquiring lock A. It is easy to see that the first write to x from the call at ③ does not introduce a race due to the shared lock A. However, the second call to x from ⑤ does indeed race with the write from T_1. If we only check the first access and filter all subsequent events from the same lexical location, we would miss this race. We also note that in the second iteration of the loop, both the write events from ③ and ⑤ would be redundant. Similarly, the writes from ④ after the fifth iteration are also redundant.

3.1.2 Inter-thread Redundancy

We can generalize the redundancy to events across different threads. Figure 3.3 illustrates a simple example. The main thread T_0 runs a loop inside which it reads and writes to a common shared address x, and forks new threads with argument

T_0

```
   for(i=1; i<10; i++){
①    read x
②    fork(Ti, i)
     lock(A)
③    write x
     unlock(A)
   }
```

T_i

```
④  if(i<5) Lock(A)
⑤  write x
⑥  if(i<5) Unlock(A)
```

Figure 3.3: Inter-thread redundancy between threads.

being the iteration index. The shared lock A is used to protect the writes to x at ③. Thread T_i writes to this shared address protected conditionally through lock A for the first five threads. The remaining threads write to x without previously acquiring lock A.

Let us consider the first iteration of the loop. Thread T_0 reads x before forking thread T_1. Thus, this read does not result in a race. Then, T_0 and T_1 write to x ordered by a lock. Thus, the first iteration has no race. In the second iteration, the read of x, which previously did not result in a race, now races with the write by T_1. Subsequent iterations all serve to expose the same lexical pair as a race and can be ignored. However, not all of the threads are identical in their execution - threads T_5 to T_9 do not acquire the lock before writing to x. In these threads, ③ and ⑤ result in a race. Among these threads, T_6 to T_9 are completely redundant to T_5.

From these examples, we see that identical program location is just a necessary condition, but not the only condition to determine if the two events are redundant or not. A key contribution of TREE is a criterion (called concurrential redundancy) that captures redundant events without any loss of race detection ability, for both intra-thread and inter-thread redundancies. Before introducing our criterion, we first introduce necessary preliminaries.

3.2 The Trace Model

To formally define the event redundancies, we need a model of a general program execution trace. Similar to previous work [28, 51], we consider an event e in a program trace τ to be one of the following:

- **MEM**(t, a, v): A memory access event, where t refers to the thread performing the memory access, a can be one of $\mathbf{R}(\mathit{Read})/\mathbf{W}(\mathit{Write})$ event and v the memory address being accessed.
- **ACQ**(t, l): A lock acquire event, where t denotes the thread acquiring the lock and l is the address of the acquired lock.
- **REL**(t, l): A lock release event, where t denotes the thread releasing the lock and l is the address of the released lock.
- **SND**(t, g): A message sending event, where t denotes the thread sending message with unique ID g.
- **RCV**(t, g): A message receiving event, where t denotes the thread receiving message with unique ID g.

For Java programs, the events $\mathit{SND}(t, g)$ and $\mathit{RCV}(t, g)$ events can be one of the following:

- If Thread T_1 starts T_2, it corresponds to a $SND(T_1,\ g)$ and $RCV(T_2,\ g)$.
- If Thread T_1 calls T_2*.join()*, $SND(T_2,\ g)$ and $RCV(T_1,\ g)$ are generated once T_2 terminates.
- If Thread T_1 calls *o.notify()* signaling a *o.wait()* on Thread T_2, this corresponds to a $SND(T_1,\ g)$ and $RCV(T_2,\ g)$.

Every event is also associated with a *loc* attribute denoting the program location that generates the event.

3.3 Concurrential Equivalence

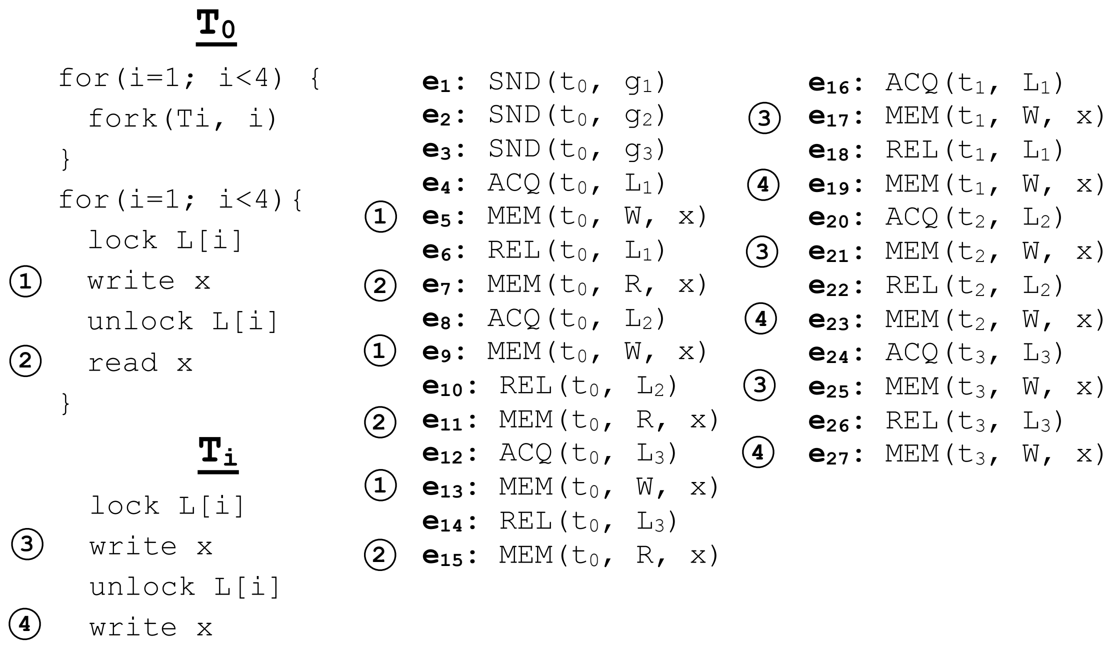
Having defined a standard model of a program trace, we now formally define a redundant event for data-race detection.

Definition 3.3.1. *An event e is said to be redundant iff $A(\alpha) = A(\alpha \backslash e)$, where A is the race detection algorithm and α is an input execution trace observed so far.*

In our case, A is a dynamic HB, LockSet, or a hybrid algorithm. In order to determine the conditions for an event to be redundant, we first look the attributes of an event that HB and LockSet track.

The Happens-Before relationship $\prec$ for a trace α is the smallest relation such that

- If a and b are events from the same thread and a occurs before b in the trace, then $a \prec b$.
- If a is a type of *SND* event and b is the corresponding RCV event, then $a \prec b$.
- $\prec$ is transitively closed.



```
          T0
   for(i=1; i<4) {
     fork(Ti, i)
   }
   for(i=1; i<4){
     lock L[i]
①    write x
     unlock L[i]
②    read x
   }
          Ti
     lock L[i]
③    write x
     unlock L[i]
④    write x
```

(a)

```
   e1: SND(t0, g1)              e16: ACQ(t1, L1)
   e2: SND(t0, g2)          ③ e17: MEM(t1, W, x)
   e3: SND(t0, g3)              e18: REL(t1, L1)
   e4: ACQ(t0, L1)          ④ e19: MEM(t1, W, x)
① e5: MEM(t0, W, x)            e20: ACQ(t2, L2)
   e6: REL(t0, L1)          ③ e21: MEM(t2, W, x)
② e7: MEM(t0, R, x)            e22: REL(t2, L2)
   e8: ACQ(t0, L2)          ④ e23: MEM(t2, W, x)
① e9: MEM(t0, W, x)            e24: ACQ(t3, L3)
   e10: REL(t0, L2)         ③ e25: MEM(t3, W, x)
② e11: MEM(t0, R, x)           e26: REL(t3, L3)
   e12: ACQ(t0, L3)         ④ e27: MEM(t3, W, x)
① e13: MEM(t0, W, x)
   e14: REL(t0, L3)
② e15: MEM(t0, R, x)
```

(b)

Figure 3.4: A program exhibiting both intra-thread and inter-thread event redundancies and a serialized execution trace.

The HB relationship is usually checked by the use of vector clocks [42]. If $a \prec b$, this implies $b \not\prec a$, *i.e., a must happen before b* in all executions of the same program. Two conflicting accesses (*i.e.*, *Read/Write* events, at least one is a *Write*, accessing the same memory address) are said to be in a race if they do not happen before each other: $\neg(a \prec b) \wedge \neg(b \prec a)$.

In LockSet-based race detectors, the contribution by locks is often ignored in the HB model. Instead, lock and unlock events are tracked separately using LockSet. The set of locks currently held by a given thread is referred to as its LockSet. The LockSet condition states that two conflicting accesses are in a race if the LockSets of the two threads do not intersect, *i.e.*, $L_i \cap L_j = \emptyset$, where L_i and L_j refer to the LockSet of T_i and T_j, respectively, at the time of event generation.

We lay the foundation of concurrential equivalence through the example in Figure 3.4a. This program exhibits both intra-thread and inter-thread redundancies. It contains two loops - in the first loop, we spawn three threads, $T_{1,2,3}$ and in the second loop, we perform a write at location ① guarded by a lock $L[i]$, and a read at ②, in each iteration. Threads $T_{1,2,3}$ are all identical except in the lock addresses each of them use to guard the write at ③. The write at ④ is unguarded. Figure 3.4b shows a serialized execution trace of the program that executes $T_0 \rightarrow T_1 \rightarrow T_2 \rightarrow T_3$. We note that there are 27 races in total in this trace: $(e_{(7,11,15)}, e_{(17,19,21,23,25,27)})$, $(e_{(5,9,13)}, e_{(19,23,27)})$, $(e_5,\ e_{(21,25)})$, $(e_9,\ e_{(17,25)})$, $(e_{13},\ e_{(17,21)})$, $(e_{17},\ e_{(21,25)})$, $(e_{21},\ e_{(17,25)})$, $(e_{25}, e_{(17,21)})$, $(e_{19},\ e_{(23,27)}4)$, $(e_{23},\ e_{(19,27)})$ and $(e_{27},\ e_{(19,23)})$. However, only six of them have unique lexical locations: (①,③), (①, ④), (②, ③), (②, ④), (③, ③) and (④, ④). The rest 21 races do not bring any additional information for programmers to aid in fixing the bug. We would like to identify those events that lead to these 21 redundant races.

The key observation behind concurrential equivalence is that, for two MEM events e_i and e_j, the combination of their LockSet and inter-thread Happens-Before relation can determine their equivalence. Regardless of which thread(s) they are from, e_i and e_j are ***concurrentially equivalent*** if they satisfy the following five conditions:

1. they share the same program lexical location;
2. they have the same access type (i.e., both are reads, or both are writes);
3. they access the same dynamic memory location;
4. they contain the same LockSet *i.e.,* $L_i = L_j$;
5. they have the same inter-thread HB relations with events from all other threads, *i.e.,* $\forall\ e_k,\ t_{e_k} \neq t_i \vee t_{e_k} \neq t_j,\ \neg(e_k \prec e_i) \wedge \neg(e_i \prec e_k) \Longleftrightarrow \neg(e_k \prec e_j) \wedge \neg(e_j$

$\prec e_k$).

For dynamic race detection, since a race involves only two events from two different threads, we can derive the following theorem:

Theorem 3.3.1. *An event e is redundant if there already exists one concurrential equivalent event from the same thread, or two from different threads.*

Proof. Let us assume two concurrentially equivalent events e_i and e_j, and consider an arbitrary event e_k. If e_i and e_j are from the same thread, and if e_k and e_i form a data-race, then e_k and e_j must be a race too. The reason is that e_i and e_j have the same LockSet and inter-thread HB relation, and e_k must be from a different thread. Hence, either e_i or e_j is redundant. On the other hand, if e_i and e_j are from different threads, and if e_k and e_i form a race, there are two possibilities. One is that e_k is from a third thread different from that of e_i and e_j. In that case, either e_i or e_j is redundant, because e_k would race with e_j too. The other case is that e_k is from the same thread as e_j. In that case, neither e_i nor e_j is redundant. However, for any other event e_w that is concurrentially equivalent to e_i and e_j, e_w must be redundant. The reason is that e_w would either form a race with e_k (if it is from a thread different from that of e_k), or is redundant with e_j (if it is from the same thread as e_k). $\square$

Meanwhile, since a race involves at least and at most two events, it is impossible to further remove any more event, otherwise a certain race can be missed. Therefore, the results obtained from Theorem 3.3.1 is also optimal. We can hence use Theorem 3.3.1 to precisely and optimally identify redundant events.

For dynamically generated event streams, checking the first three conditions of concurrential equivalence is easy – lexical equivalence can simply check the originating program location of the event, access types can be recognized easily during instrumentation, and dynamic memory location is available at runtime. Checking the last two conditions however, if done naively, would prove prohibitively expensive, especially when the algorithm needs to be run online during program execution. To efficiently check the last two conditions, we introduce a new concept – ***concurrency context***:

Definition 3.3.2. *The concurrency context of a thread t, Γ_t, encodes the LockSet and the history of SND, RCV events observed by t.*

Definition 3.3.3. *The concurrency context of an event e generated by thread t, is the value of Γ_t at the time e is observed.*

It is easy to see that two events with the same concurrency context must satisfy the last two conditions of concurrential equivalence, because both the LockSet and inter-thread Happen-Before of the event are encoded in the concurrency context.

Finally, we introduce the concept of ***concurrency history*** for a particular lexical location.

Definition 3.3.4. *The concurrency history at a location loc, Θ_{loc}, stores the union of Γ_t of all threads t that have accessed this location.*

Θ_{loc} is constructed as $\Theta_{loc} = \Gamma_{t_i} \vee \Gamma_{t_j} \vee \Gamma_{t_k} \ldots$, where Γ_{t_i}, Γ_{t_j}, Γ_{t_k} represent the concurrency contexts of different threads at the time the events (if there is any) were generated from *loc*. Since the concurrency contexts of different events from the same location exhibit strong temporal locality due to stack based computational model of programs, a prefix sharing data-structure such as *trie* is ideal for storing Θ_{loc}. As we shall see in the Section 3.4, this results in compact storage and fast retrieval.

Algorithm 1 TREE(e)

```
 1: $e \leftarrow$ input event
 2: $t = e.getThread$
 3: $loc = e.getLocation$
 4: $\Gamma_t$ // concurrency context of thread $t$
 5: $\Theta_{loc}$ // concurrency history at location $loc$
 6: switch $e$ do
 7:     case MEM:
 8:         if DetectRedundancy($t$, $\Theta_{loc}$, $\Gamma_t$) then
 9:             discard $e$
10:         else
11:             advance $e$
12:         end if
13:     case ACQ:
14:         $\Gamma_t.add(e.lock)$
15:         advance $e$
16:     case REL:
17:         $\Gamma_t.remove(e.lock)$
18:         advance $e$
19:     case SND $\vee$ RCV:
20:         $\Gamma_t.add(e.g)$//add the message $g$
21:         advance $e$
```

3.4 The TREE Technique

We design TREE as a filter pass over the event stream generated by the program execution. It is generic by design and can be applied to any Happens-Before or LockSet based detector. Algorithm 1 provides a high-level overview of how TREE applies the redundancy filter. It updates Γ_t as events stream by. The calls `discard` and `advance` indicate when TREE decides that the event is redundant and discard it or advance it to the race detector, respectively. Each type of event is handled separately:

1. **MEM**: Memory access events, both read and write are checked for redundancy through the **DetectRedundancy** call. If this call returns true, the event is

Algorithm 2 DetectRedundancy(t, Θ_{loc}, Γ_t)

```
1: $\Gamma_t$ // concurrency context of $e$
2: stack = $\Theta_{loc}$.contains($\Gamma_t$)
3: if stack.size = 0 then
4:     $\Theta_{loc}$.add($\Gamma_t$)
5:     return False
6: else if stack.contains($t$) then
7:     return True
8: else if stack.size = 1 then
9:     stack.add($t$)
10:    return False
11: else if stack.size = 2 then
12:    return True
13: end if
```

redundant and it is filtered.

2. **ACQ**: Lock acquire events add the lock address into Γ_t. If a lock previously acquired is acquired again, we ignore the event.

3. **REL**: Lock release events remove the lock address from Γ_t. In a well formed trace, the corresponding lock acquire event of this address must have already been observed before this event is seen.

4. **SND** and **RCV**: These events always append to Γ_t their unique ID g.

We check redundancy only for events of *MEM* type in the current design and all synchronization events are passed through. If it is a *MEM* event, we call the function **DetectRedundancy** to determine its redundancy by passing the corresponding concurrency history Θ_{loc} and concurrency context Γ_t of the thread.

Algorithm 2 then describes the algorithm to detect redundancy. It receives Γ_t as the current concurrency context of the event being checked. As we have seen previously in Theorem 3.3.1, an event is redundant if there already exists one concurrentially equivalent event from the same thread, or two from different threads.

To check this condition, each node in Θ_{loc} contains a bounded stack of size *two* that is used to keep track of the number of concurrential equivalent events seen so far. If the stack is full, new events having the same Γ_t are filtered out since they are redundant. The elements of the stack denote the threads that have contributed to the particular concurrency context. The first step is to check the stack corresponding to the current thread's concurrency context from Θ_{loc}. Based on the contents of this stack, there are four cases to consider:

1. **Stack is empty:** This implies that this particular concurrency context was not seen in any of the accesses so far, hence the event is not redundant. We proceed to add Γ_t into Θ_{loc} for future accesses.

2. **Stack contains t:** This case falls in the category of intra-thread redundancy and can be eliminated. t is the thread id of the current event e.

3. **Stack does not contain t and is of size 1:** Add t to the stack.

4. **Stack is full:** This event is redundant, so filter it out.

$$\text{①} \longrightarrow [g_1,\ g_2,\ g_3,\ L_i]$$

$$\text{②} \longrightarrow [g_1,\ g_2,\ g_3]$$

$$\text{③} \longrightarrow [L_i]$$

$$\text{④} \longrightarrow [\,]$$

Figure 3.5: Γ_t at the time when the locations ①-③ are accessed.

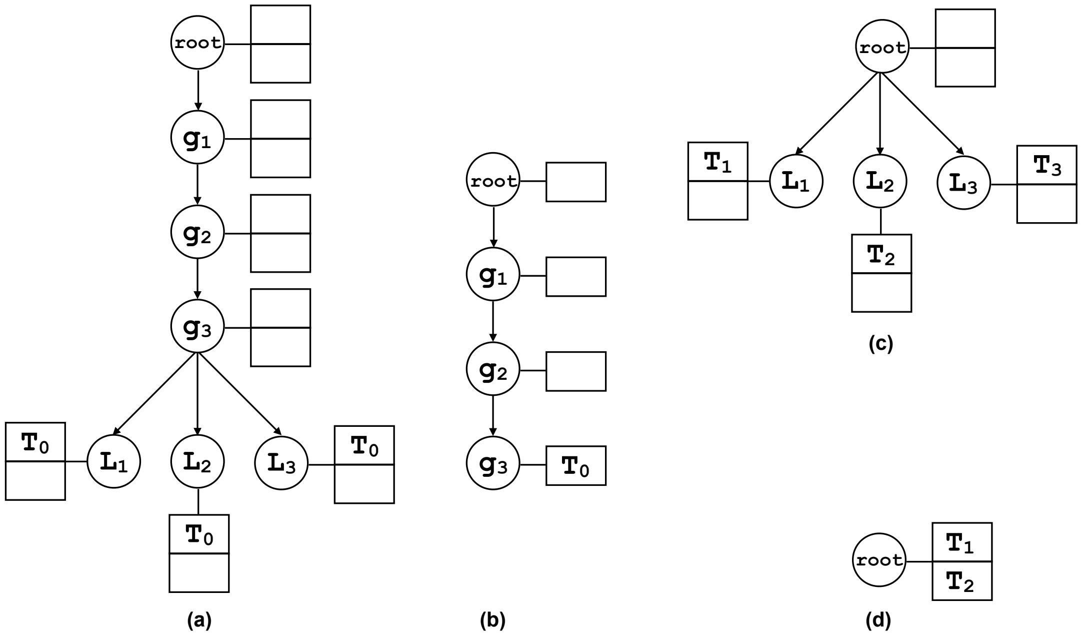


Figure 3.6: Concurrency history Θ_{loc} state of the four locations from Figure 3.4b: (a)→①, (b)→②, (c)→③, (d)→④.

3.5 A Case Study

Let us now see how TREE is able to prune these events for the example in Figure 3.4. There are four program locations of interests, marked ①-④. The Γ_t at these locations determines what gets put into the respective location Θ_{loc}. Γ_t can be thought of as a simple set constructed in program order. Figure 3.5 shows the state of Γ_t at these locations and Figure 3.6 shows the state of Θ_{loc} at the end of the three iterations.

- **Location ①:** Following Algorithm 1, the three events $e_{1,2,3}$ first add their unique message ids into the Γ_t. The lock acquire by e_4 is also appended into the Γ_t before the access at ①. At the end of three iterations, g_3 contains 3 branches formed by the three lock addresses. The stack at each of these locations contains the single thread T_0 and thus, none of the accesses are filtered.

- **Location ②:** The lock release events appearing before the access to this location remove the associated lock address from Γ_t. In the first iteration, the stack is empty and thus, the event is not filtered and T_0 is added into the stack. For every subsequent iteration, since T_0 is already present, it is redundant and hence filtered out.

- **Location ③:** Similar to how each T_0 acquires a lock before the write at ①, the write at ③ is guarded by a lock. Thus, the Θ at this location is similar to that at ①. The only difference is that each thread acquires a different lock, as we can see from the final state of the Θ.

- **Location ③:** Finally, this location is similar to the write by T_0 at ②. The first two threads T_1 and T_2 access this location and get added into the stack. The third thread T_3 is however filtered since the stack is already full, exhibiting inter-thread redundancy.

3.6 Optimization

We discuss an optimization to TREE, inspired by [22], that is specifically tied to race detectors such as FastTrack [21], which only guarantee to detect *the first race*, if one exists. Making use of this fact, we can define *span redundancy* to further improve runtime performance and reduce memory overhead.

Definition 3.6.1. *A **span** refers to the region between two outgoing HB edges of the same thread.*

Theorem 3.6.1. *If e_i and e_j refer to two events originating from the span accessing the same memory location, and e_i appears before e_j in the trace, then if e_j is in a* *race with some other conflicting event, e_i is also in a race with the same conflicting event.*

Proof. Let us assume e_i is not in a race with the other event. Then in that case, the vector clocks of e_i and e_j would be different (from the Happens-Before relation). This would imply that there is a HB edge in between them - but this is not possible since they both belong to the same span, from Definition 3.6.1. Thus, e_i must also be in a race. $\square$

Span redundancy appears within trace events from the same thread that are in the same *span*. From Theorem 3.6.1, we determine that if there is a race on an access to a shared address, a race would also exist with the first access to that shared address in the current span (and every other access to the shared address in that span). While we cannot say anything about future accesses if the first access results in a race, if we determine the first access in a particular span to be race-free, then we can safely ignore all other accesses in that span. In addition, since FastTrack guarantees to only detect the first race on a shared address, we may consider just the first access to each shared address.

Consider the example in Figure 3.7. The outer loop runs five times and the inner loop runs for each element of the array. Inside the inner loop, we perform a write on $Array[j]$ and j in each iteration. There exists no synchronization in the program trace, and hence the all accesses are within the same span. This implies that race checks on the first access to each unique memory location are sufficient. This reduces the events that need to be checked for races by four times $Array.length()$.

Span redundancy is explored via static analysis in RedCard [22]. Some of the benefits of applying span redundancy at runtime instead of statically are

1. Greatly simplified algorithm instead of special cases to deal with various types

T_0

```
① for(i = 0; i < 5)
  {
②   for(j = 0; j < Array.length)
    {
③     write Array[j]
④     j = j + 1
    }
⑤   i = i + 1
  }
```

Figure 3.7: Program snippet depicting an example of span redundancy in array accesses.

of operations.

2. Handling aliases automatically without any special instrumentation or analysis.

3. Handling array accesses in the same way as that for object field accesses.

We also observe that this type of redundancy is truly useful only on the first access of each lexical location, since subsequent accesses will be covered by the concurrential redundancy. This redundancy is also much less flexible than the techniques implemented in TREE since it would not work across span boundaries. As such, we have not evaluated span redundancy since it cannot be generalized for all dynamic race detectors and still ensure soundness.

4. A PRECISE RACE DETECTOR

As noted previously in Section 1.2, several practical issues result in less than ideal situations where the trace information gathered is incomplete. This is particularly harmful when the missing region contains synchronization events. We start by illustrating the problem of incomplete trace 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.

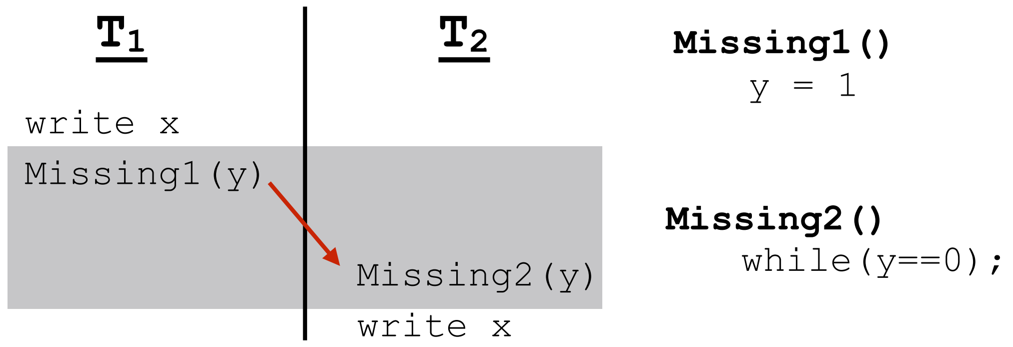


Figure 4.1: Ad hoc synchronization in the missing methods results in false alarms reported by Happens-Before.

4.1 Examples

Consider the trace in Figure 4.1. We have two threads T_1 and T_2 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 [26, 32, 54] will all report a race between the two accesses. However, this is a false alarm when the two missing methods in the region introduce an ad hoc synchronization on a shared address y (set to 0 initially). Thread T_1, after performing the *Write* to x, sets y to 1, while Thread T_2 waits until y is set before it can perform the *Read* on x. The shared address y is used as a barrier in Thread T_2 to induce a desired ordering.

A simple approach to avoiding false alarms in the presence of these missing events would be to consider each, or a sequence of continuous missing events as a barrier, and add HB edges between barriers in the observed order (**Caveat 0**). This approach would guarantee to detect no false alarm, because it strictly serializes the missing events. However, it is also overly conservative that it would miss many true data races. For instance, if the two missing methods in Figure 4.1 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.

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 Figure 4.1 again. Both of the two missing methods in threads T_1 and T_2 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 intersected addresses to synchronize. This observation leads to our introduction of the BarrierPair, explained next.

4.2 The BarrierPair Model

Building on from [46], we introduce the concept of a BarrierPair. 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.

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 4.4) 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.

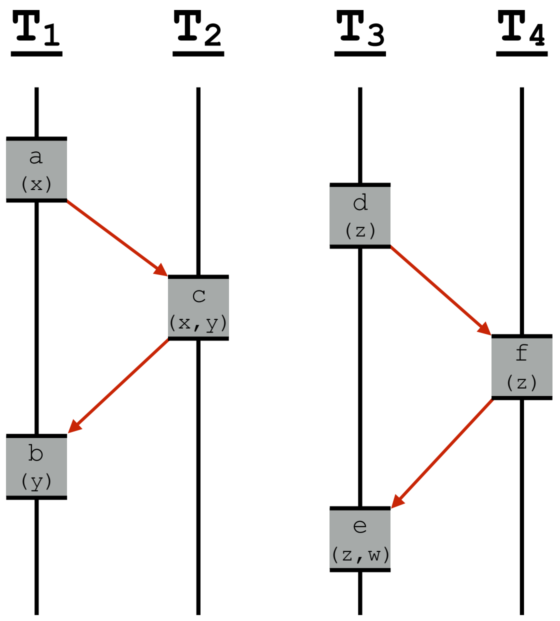


Figure 4.2: A program trace consisting of four threads and six BarrierPairs $(a - f)$, each denoting a missing method call with its reachable memory addresses. For example, the BarrierPair $a(x)$ denotes that the corresponding missing method a may access address x. Four HB edges $(a \rightarrow c, c \rightarrow b, d \rightarrow f, f \rightarrow e)$ are added between those BarrierPairs with overlapping reachable addresses.

The attributes of BarrierPair can be recorded and computed at runtime without knowing the computation in the missing methods. This information can be used to determine 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 4.2 illustrates six BarrierPairs in a trace and four added HB edges between them. With this enhancement, the same HB algorithm [20, 32] or other precise algorithms [26, 54] can be directly applied to detect races without any change.

Moreover, the BarrierPair model matches with 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 [10, 26, 56]. It reduces both the trace size and runtime overhead, and also avoids the problem of tracing native code used in those excluded methods. Furthermore, BarrierPair can be used to approximate the computation inside the missing method. For example, if the method is deterministic, given the same invocation data, it will always produce the same return data.

4.3 Technical Challenges

The BarrierPair model provides a 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 sufficient 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 five related caveats in the rest of this section through Figures 4.3, 4.4 and 4.5. The threads and BarrierPairs in these examples correspond to that in Figure 4.2 with minor modification on the reachable addresses (explained below). A false race on the two observed accesses to x would be reported if any of the HB edges (denoted by the red arrows) are missed. We then present our race detection technique in detail in Section 4.4 to address all these challenges.

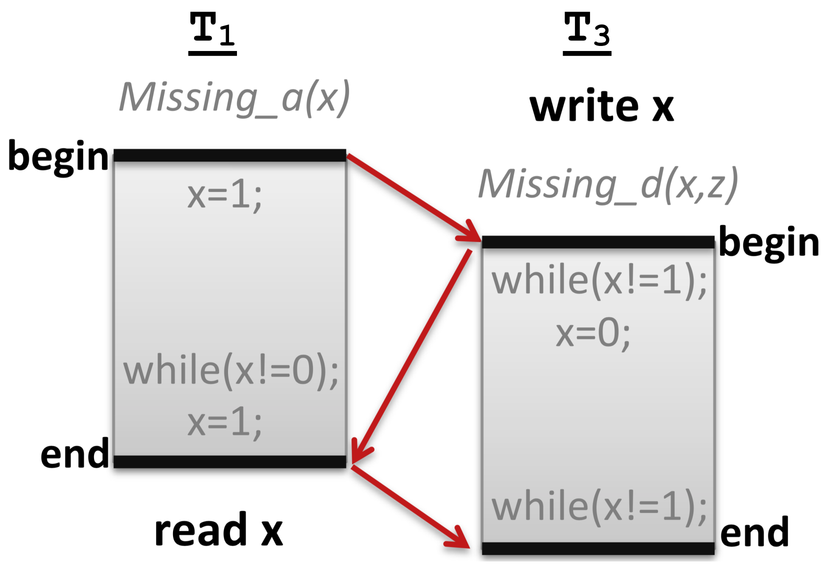


Figure 4.3: Overlapping BarrierPairs can incur multiple HB edges.

- **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 of the trace. For example, in Figure 4.2, 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, x, and a occurred before c. However, this naive method does not work when two BarrierPairs overlap in time. For instance, suppose the BarrierPair d in Figure 4.2 overlaps with a *i.e.,* d also accesses address x. We cannot simply add one HB edge from a to d or from d to a. The reason is that the overlapping region may incur multiple HB edges between events in the missing methods. Consider an example in Figure 4.3. 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 x. For instance, the HB edge $d.begin \rightarrow a.end$ must be added, because the *MethodEnd* event of BarrierPair a cannot happen until x is set to 0 by Thread T_3, which is after the *MethodBegin* of BarrierPair d. Otherwise, a false alarm would be reported between the *Read* to x in Thread T_1 and *Write* to x in Thread T_3.

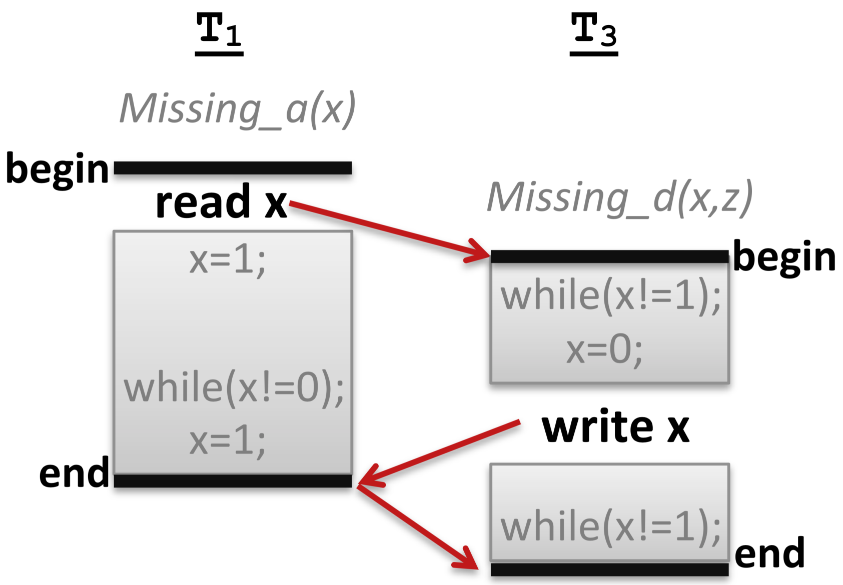


Figure 4.4: Events in between BarrierPairs may be observed and can introduce HB edges.

- **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 4.4 (slightly modified from Figure 4.3). 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 4.3. However, this is a false alarm because the *Write* cannot happen until x is set to 1 by Thread T_1, 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 Figure 4.4 again – suppose the method d in Thread T_3 is not missing, the events at '`while x!=1`' are ordinary events. We must add a HB edge from the event *Read(x)* in Thread T_1 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 pair-wisely. For example, consider the three BarrierPairs c, e, and f shown in Figure 4.5, 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,

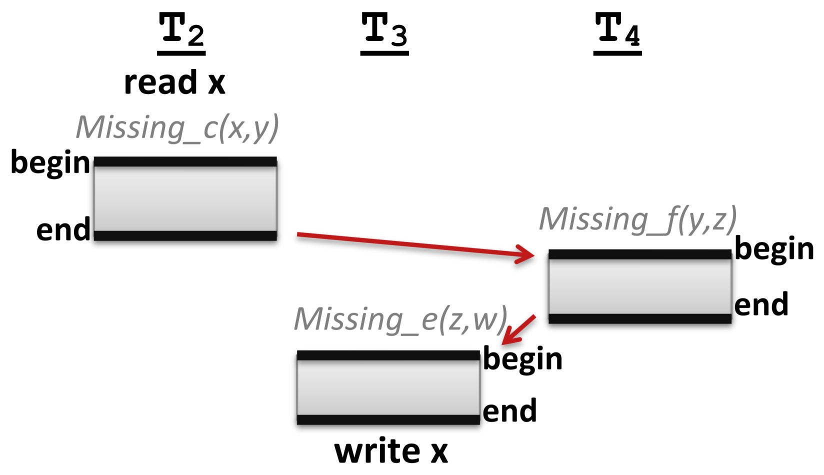


Figure 4.5: Multiple BarrierPairs can introduce HB edges transitively.

the BarrierPair c must happen before e, though they do not have any common reachable address. Hence, the two accesses to x by threads T_2 and T_3 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 synchronization 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 in missing methods, false alarms may be introduced[1].

One way to address this issue is to use the simple barrier approach described in **Caveat 0**, which is ineffective. Instead, we propose a language extension that allows the users of RDIT to annotate direct global variable accesses at the call sites of

[1]Note that in the BarrierPair model, global variables are allowed to be accessed in missing methods as long as their accesses are visible (for example, through callbacks). No false alarm will be introduced in such cases.

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 may be directly accessed in a missing method. At runtime, x is added to the set of reachable memory addresses of the BarrierPair. This method guarantees soundness, though reduces automation. However, note that directly accessing global variables in external methods is considered bad programming practice and is rarely seen in real-world production systems. In all 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, annotations are almost never needed in practice to use RDIT.

4.4 The RDIT Technique

We first present in Section 4.4.1 a formal model of maximal thread causality with missing events, following the approach introduced in [26] (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 4.4.2, 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 [26] to guarantee soundness and maximality. In addition, we must also consider the additional constraints introduced by the BarrierPairs.

4.4.1 Maximal Causality Model with Missing Events

Consider an arbitrary multi-threaded 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 [20, 26, 54]:

- *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

In this model, 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 always paired for the same thread t and method m. Each pair of *MethodBegin* and *MethodEnd* events form 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 closeness* 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*, $MCM(\tau)$, 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 $MCM(\tau)$ by checking the two axioms, such that (1) any program which can generate τ can also generate all traces in $MCM(\tau)$, and (2) for any trace τ' not in $MCM(\tau)$ there exists a program generating τ which cannot generate τ'. Note that $MCM(\tau)$ here is different from that in prior work [26], because τ is incomplete and contains BarrierPairs that abstract missing events.

4.4.2 Data Race Detection Algorithm

To perform precise and maximal race detection, intuitively, we can generate $MCM(\tau)$ and detect races in every trace in the set. However, generating $MCM(\tau)$ is challenging. Exhaustively enumerating all re-orderings 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 4.3). In RDIT, following [26], we encode $MCM(\tau)$ as a series of quantifier free first-order logic formulas, Φ_{mcm}, such that any solution to Φ_{mcm} represents a trace in $MCM(\tau)$. 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 $MCM(\tau)$ with two conflicting events a and b from different threads, such that a and b are next to each other in τ'. Algorithm 3 outlines our race detection algorithm. A key step is to introduce an order variable O_e for each event e in τ, denoting the order of e in τ', and use these order variables to encode Φ_{mcm}. We first construct the formula Φ_{mcm} from τ, which involves getting the set of all BarrierPairs from

Algorithm 3 The RDIT Algorithm

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

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 $MCM(\tau)$ in which the two events a and b are unordered, and hence (a, b) is a true race.

Algorithm 4 ConstructMCMFormula(τ)

```
1: $\tau$ ←input trace
2: $\Phi_{mcm} = true$
3: $\Phi_{mcm}$ ∧= **ConstructBarrierPairConstrains($\tau$)**
4: $\Phi_{mcm}$ ∧= **ConstructProgramOrderConstrains($\tau$)**
5: $\Phi_{mcm}$ ∧= **ConstructForkJoinConstrains($\tau$)**
6: $\Phi_{mcm}$ ∧= **ConstructLockingConstrains($\tau$)**
7: $\Phi_{mcm}$ ∧= **ConstructReadConsistencyConstrains($\tau$)**
8: **return** $\Phi_{mcm}$
```

4.4.3 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 object-oriented programs such as Java, 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). 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 large runtime overhead and produce huge logs when calls to missing methods are frequent and the object tree is large.

We develop an efficient method that does not compute a complete set of reachable addresses for every object upon every missing method call, but only *once* for each object for all missing method calls. 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 not in D, o_2 should not be used for synchronization. In fact, this condition is never violated in our study of real-world applications (see Section 5.2). 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.

Algorithm 5 ConstructBarrierPairConstrains(τ)

```
1: $\tau \leftarrow$input trace
2: $\Phi_{mcm} = true$ // initialized to true
3: $BP$ = **ComputeBarrierPairs**($\tau$)
4: **for** $bp_1, bp_2 \in BP$ **do**
5:     **if** $bp_1.D \wedge bp_2.D \neq \emptyset$ **then**
6:         $S \leftarrow$ **UnionEvents**($bp_1, bp_2$)
7:         $\Phi_{mcm} \wedge =$ **GetLinearizationConstrains(S)**
8:     **end if**
9: **end for**
10: **for** $bp \in BP$ **do**
11:     **for** $x \in bp.D$ **do**
12:         **for** $e \in$ **GetAllReadWritesOnAddress**($\tau, x$) **do**
13:             $S \leftarrow$ **UnionEvents**($bp, e$)
14:             $\Phi_{mcm} \wedge =$ **GetLinearizationConstrains(S)**
15:         **end for**
16:     **end for**
17: **end for**
18: **return** $\Phi_{mcm}$
```

4.4.4 Constraint Encoding of MCM(τ)

Algorithm 4 shows our constraint encoding algorithm for $MCM(\tau)$. 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} conjuncts on the following five types of constraints:

1. **BarrierPair Constraints - Algorithm 5:** This type of constraint addresses 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 intersect, 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 syn-

Algorithm 6 GetLinearizationConstrains(S)

```
1: $S \leftarrow$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** $\Phi$
```

chronization with any other event (either observed or not) accessing the intersected address. Specifically, the function **UnionEvents** first unions all these events into a set S. Then we call the **GetLineatizationConstrains** (Algorithm 6) function on this set. It 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 intersected address, we construct constraints to enforce their HB orderings.

Algorithm 7 ConstructProgramOrderConstrains(τ)

```
1: $\tau \leftarrow$input trace
2: $\Phi_{mcm} = true$ // initialized to true
3: $T$ = **GetAllThreads($\tau$)**
4: **for** $t \in T$ **do**
5:     $\tau_t$ = **GetThreadEvents($\tau$,$t$)** // events by Thread $t$
6:     **for** $i = 1 : |\tau_t| - 1$ **do**
7:         // $O_{t,i}$: order variable of the $i$th event in $\tau_t$
8:         $\Phi_{mcm} \wedge = O_{t,i} < O_{t,i+1}$
9:     **end for**
10: **end for**
11: **return** $\Phi_{mcm}$
```

2. **Program Order Constraints - Algorithm 7:** This type of constraint ensures sequential consistency, such that events from the same thread cannot be reordered. Specifically, we construct 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 [55]. Nevertheless, we focus on sequential consistency in this work.

Algorithm 8 ConstructForkJoinConstrains(τ)

```
1: $\tau \leftarrow$input trace
2: $\Phi_{mcm} = true$ // initialized to true
3: for $e \in$ GetThreadForkJoinEvents($\tau$) do
4:     if $e = ThreadFork(t, t')$ then
5:         $\Phi_{mcm} \wedge = O_e < O_{t',begin}$
6:     else if $e = ThreadJoin(t, t')$ then
7:         $\Phi_{mcm} \wedge = O_{t',end} < O_e$
8:     end if
9: end for
10: return $\Phi_{mcm}$
```

3. **Fork Join Constraints - Algorithm 8:** 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 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 $ThreadJoin(t', t)$.

Algorithm 9 ConstructLockingConstrains(τ)

```
1: $\tau \leftarrow$input trace
2: $\Phi_{mcm} = true$ // initialized to true
3: $L$ = **GetAllLocks($\tau$)**
4: **for** $l \in L$ **do**
5:     // pairs of lock/unlock events on $l$
6:     $LP_l$ = **GetLockPairs($\tau, l$)**
7:     **for** ($e_a$,$e_b$), ($e_c$,$e_d$) $\in LP_l$ **do**
8:         $\Phi_{mcm} \wedge = O_{e_b} < O_{e_c} \vee O_{e_d} < O_{e_a}$
9:     **end for**
10: **end for**
11: **return** $\Phi_{mcm}$
```

4. **Locking Constraints - Algorithm 9:** 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_l, e_u), (e'_l, e'_u), we construct the constraint ($O_{e_u} < O_{e'_l} \vee O_{e'_u} < O_{e_l}$) and conjunct them.

5. **Read Consistency Constraints - Algorithm 10:** This type of constraint ensures that the two basic axioms (recall Section 4.4.1) are satisfied by requiring that every event in the inferred trace τ' is feasible.

Due to prefix closeness, 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 constraints 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 interfering *Write* to x with

Algorithm 10 ConstructReadConsistencyConstrains(τ)

```
1: $\tau \leftarrow$input trace
2: $\Phi_{mcm} = true$ // initialized to true
3: for $e = Read(t, x, v) \in \tau$ do
4:     $W^x \rightarrow$ GetAllWritesOnAddress($\tau, x$)
5:     $W_v^x \rightarrow$ GetAllWritesOnAddressValue($\tau, x, v$)
6:     $\Phi_{mcm} \wedge = \bigvee_{w \in W_v^x} (O_w < O_e \bigwedge_{w \neq w' \in W^x} (O_{w'} < O_w \vee O_e < O_{w'}))$
7: end for
8: return $\Phi_{mcm}$
```

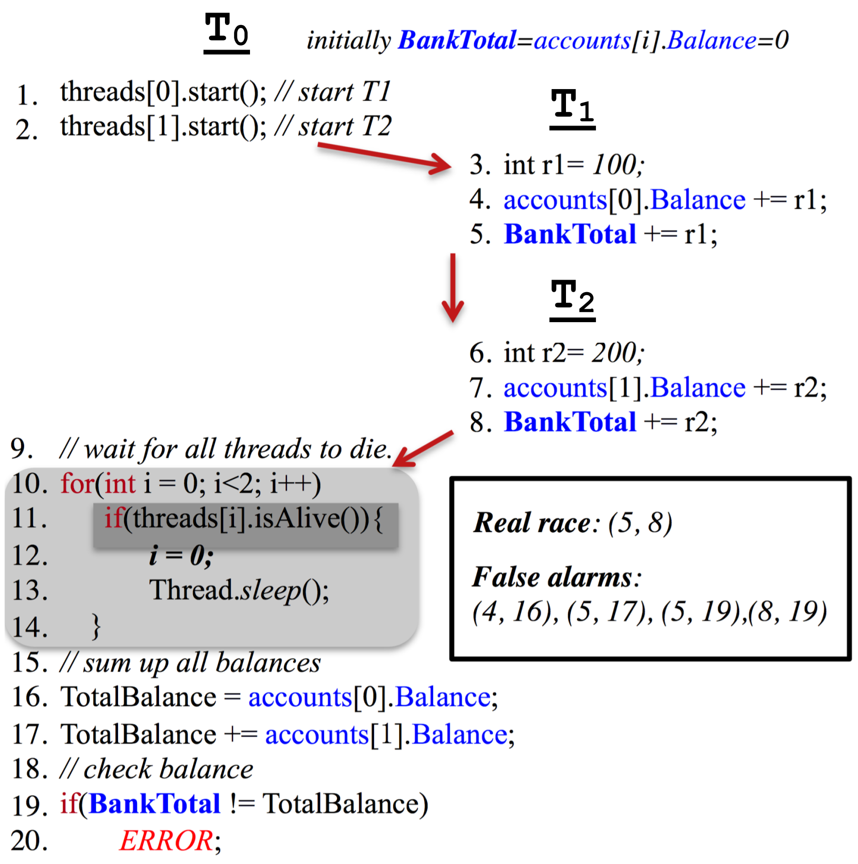
a different value. The size of read consistency constraints is cubic in the number of *Read/Write* events, and may dominate the size of Φ_{mcm}.

It is worth noting that the constructed formula Φ_{mcm} encodes all the feasible traces in $MCM(\tau)$. Each solution of the order variables to Φ_{mcm} corresponds to a valid reordering of events in τ. The size of $MCM(\tau)$ 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 $MCM(\tau)$. Instead, it suffices to just find one trace that satisfies the race condition.

4.5 A Case Study

In this section, we present a case study of race detection in a popular multi-threaded benchmark - *Account* (Figure 4.6). We show that all existing precise algorithms [20, 26, 54] report several false alarms in this benchmark due to missing events in the naive library and illustrate how RDIT avoids them.

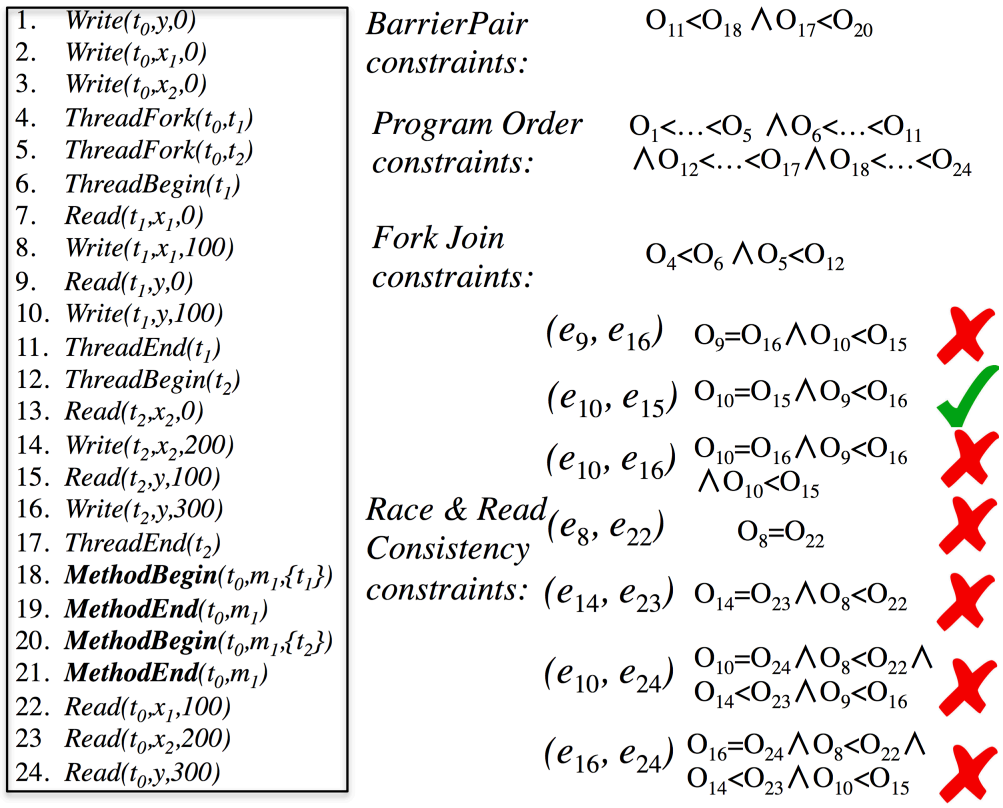
We first describe the false alarms present in the Account benchmark. This benchmark has been used frequently in previous race detection studies [20, 26, 31, 51, 54]. In this program, a number of bank accounts are simulated by concurrent threads to handle deposits. The sum of deposited amounts by all threads is tracked dynamically.



```
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;
```

Figure 4.6: The *Account* benchmark. Existing precise dynamic algorithms such as Happens-Before all report four false alarms due to missing events caused by the native method call *Thread.isAlive*() at line 11.

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 4.6 shows code snippets of the main thread (T_0) and two account threads (T_1 and T_2). The loop at lines 10-14 in T_0 is important to note here. It behaves as a join for T_1 and T_2, though it contains no *Thread.join*() statement. Specifically, Line 11 calls *Thread.isAlive*() to check if T_1 and T_2 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 naive method implemented through JNI, it is difficult



1. $Write(t_0,y,0)$
2. $Write(t_0,x_1,0)$
3. $Write(t_0,x_2,0)$
4. $ThreadFork(t_0,t_1)$
5. $ThreadFork(t_0,t_2)$
6. $ThreadBegin(t_1)$
7. $Read(t_1,x_1,0)$
8. $Write(t_1,x_1,100)$
9. $Read(t_1,y,0)$
10. $Write(t_1,y,100)$
11. $ThreadEnd(t_1)$
12. $ThreadBegin(t_2)$
13. $Read(t_2,x_2,0)$
14. $Write(t_2,x_2,200)$
15. $Read(t_2,y,100)$
16. $Write(t_2,y,300)$
17. $ThreadEnd(t_2)$
18. **MethodBegin**$(t_0,m_1,\{t_1\})$
19. **MethodEnd**(t_0,m_1)
20. **MethodBegin**$(t_0,m_1,\{t_2\})$
21. **MethodEnd**(t_0,m_1)
22. $Read(t_0,x_1,100)$
23 $Read(t_0,x_2,200)$
24. $Read(t_0,y,300)$

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}) $O_{16}=O_{24} \wedge O_8<O_{22} \wedge O_{14}<O_{23} \wedge O_{10}<O_{15}$ ✗

Figure 4.7: 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).

to trace the computations inside the method. As a result, existing dynamic race detectors [23, 26, 52] all report false alarms at lines (4,16), (5,17), (5,19), (8,19) due to missing events in this method, even though the race detection algorithms [20, 26, 54] they use are precise. In fact, the only true race in this benchmark is between lines (5,8) (because T_1 and T_2 can execute concurrently and there is no lock protecting these two statements), and this race may cause the error at line 20 to occur.

Next, we illustrate how RDIT detects the only true race present. Suppose we observe an execution of the program following an order denoted by the line numbers. The corresponding trace is shown in Figure 4.7. 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 4.4.1, 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*($\mathrm{t_i}$)/$e_{11,17}$: *ThreadEnd*($\mathrm{t_i}$) 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 from 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 4.7. 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. 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.

Conjuncting all these constraints, we invoke an SMT solver (Z3 [14] 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 six conflicting event pairs at lines (4,16), (5,17), (5,19), (8,19), the solver reports that no solution exists. Therefore, all of them are false alarms.

5. RESULTS AND DISCUSSIONS

5.1 TREE

We have implemented TREE in the RoadRunner framework [23], which also implements the FastTrack algorithm. Our implementation is open source and publicly available at `https://github.com/parasol-aser/TREE`.

Our evaluation focuses on answering the following two research questions:

1. **Effectiveness:** How effective is TREE in removing redundant events? How much percentage of redundant events are there in real-world applications?

2. **Efficiency:** Can TREE improve runtime performance of dynamic race detectors? How much speedup or slowdown can TREE bring?

5.1.1 Evaluation Methodology

We use TREE as a pre-processing step in the FastTrack tool chain, and compare the results and performance between vanilla FastTrack and FastTrack+TREE. TREE intercepts the event steam generated by RoadRunner and passes them along to FastTrack when it determines that the specific event is not redundant. We have evaluated TREE on a set of standard Java benchmarks as well as custom micro-benchmarks that we design for quantifying the performance characteristics of TREE. We have also run TREE on two large real-world applications – Jigsaw and Derby. Table 5.1 summarizes these applications. Each of these applications were tested running on 4 threads. The hardware used to run these experiments was a Apple MacBook Pro machine with 2.6 GHz Intel Core i5 processor, 8 GB DDR3 memory with Java JDK 1.7 installed.

Program	**%Skipped**	**ΔMemory**	**%SpeedUp**
atomicity	25.67	0	2.80
chess	0	0	0.63
moldyn	51.85	16	-0.95
montecarlo	58.67	388	27.14
jgfUtilAll	50.60	250	26.46
raytracer	50.74	68	12.73
philo	47.20	90	13.92
tsp	19.93	36	24.52
boundedbuffer	6.97	0	2.79
nestedMonitor	11.11	1	2.65
pipeline	15.84	0	1.81
sor	3.26	0	0.24
stringBuffer	21.87	0	16.56
jigsaw	35.78	35	13.20
derby	69.99	973	15.57

Table 5.1: Experimental results of running FastTrack and FastTrack + TREE on a bunch of benchmarks and a couple of real-world programs. All the benchmarks were run on 4 threads. We captured the memory usage of the entire benchmark and the complete execution time of the program. We also captured the number of events that are skipped by using TREE versus the total events generated by RoadRunner. The columns indicate the percentage of skipped events, the delta memory increase because of TREE and the percentage improvement in runtime respectively.

5.1.2 Standard Benchmarks

Table 5.1 reports our experimental results on the standard Java benchmarks. We make several observations from these results below.

- **Runtime:** Overall, TREE improved the runtime speed of FastTrack by 10-25% on most benchmarks (as large as 27% on *MonteCarlo*). For some small benchmarks, TREE did not result in noticeable improvements (less than 3%), because they do not generate a large number of events. However, we do note that even in these cases, TREE does not add any noticeable performance deterioration to the program execution, in both program runtime and memory

```
         T0
for(i=1; i< num_threads){
   fork(Ti)
}

         Ti
for(j = 1; j < num_iters){
  for(k = 1; k < num_locks) lock L[k]
  x = x + 1
  for(k = 1; k < num_locks)unlock L[k]
}
```

Figure 5.1: Sample program snippet that targets redundant event elimination.

overhead. For the larger programs (*i.e., Jigsaw* and *Derby*), the runtime improvements were more modest (13%-15%) since, as the data-structures holding the meta concurrency context information become larger, their accesses time increases due to hardware cache misses. We can optimize this further through a user-configurable limit on the size of the concurrency context, discussed below.

- **Redundant events:** The percentage of redundant events that TREE is able to safely skip is pretty large, ranging from 35-70% of the total number of events for reasonably sized programs. This suggests that TREE is effective in practice as a pre-processing step in removing redundant events for dynamic race detectors.

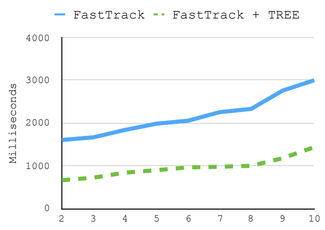
- **Memory overhead:** We notice that for most benchmarks, there was only a modest increase in memory overhead from the use of TREE. Memory usage was measured across the entire program run. The largest increase came from running Derby (with 973MB memory increase), but we also saw that this application resulted in the maximum number of skipped events with good improvement in runtime. Currently, TREE stores the concurrency context for all program locations encountered till that point. Nevertheless, for many applications in practice, keeping track of just the ***n*** most recent locations could be effective enough. An example of such an application are programs where the loops are small, say around 100 program locations. In this case, keeping track of k*100 (where k can be configured by the user) locations is sufficient to get a good performance-memory balance. Additionally, we find that for most of the benchmarks we have studied, the loop sizes are typically small, making this a useful user configurable parameter.

- **Warnings verbosity:** One additional benefit we observed, that wasn't originally planned, was that error output verbosity tended to be greatly reduced. Sometimes, we observed that FastTrack reported races on a particular race pair several hundreds or thousands of times, even though a single instance is sufficient to alert the programmer to the concurrency bug. TREE filters all these other races before sending them to FastTrack, saving the user valuable time in parsing the tool output. This proved really useful in the evaluation stage to compare the output with and without our filter.

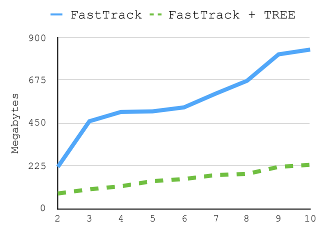
Finally, we also empirically validated that the number of unique races detected by FastTrack matches the number of races detected upon using TREE. This confirms that TREE is both theoretically sound and practically useful.

5.1.3 Micro-benchmarks

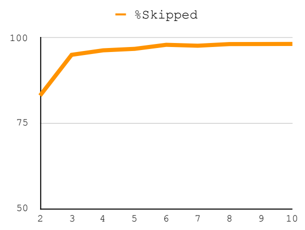
In addition to the standard benchmarks, we specifically sought to target both the strengths and the weaknesses of our algorithm to establish an upper-bound and



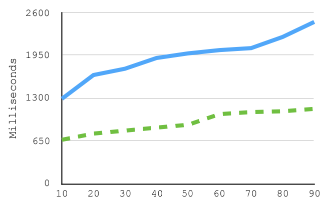
(a) 100 iters; 50 locks



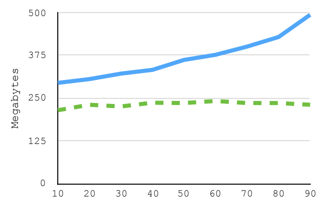
(b) 100 iters; 50 locks



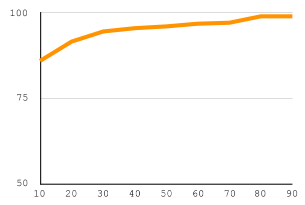
(c) 100 iters; 50 locks



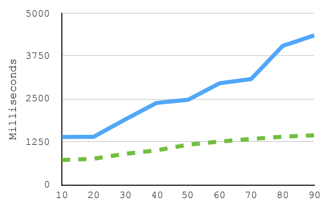
(d) 10 threads; 50 locks



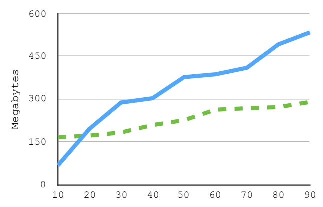
(e) 10 threads; 50 locks



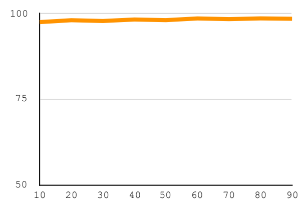
(f) 10 threads; 50 locks



(g) 10 threads; 100 iters



(h) 10 threads; 100 iters



(i) 10 threads; 100 iters

Figure 5.2: The number of threads, number of iterations and the number of locks are the parameters on which the graphs are generated for a Java program similar to the sample in Figure 5.1. These graphs depict the execution time, memory overhead and percentage of skipped events respectively. Figure (a) - (c) plot the changes in these values as number of threads is varied, Figure (d) - (f) plots the changes as the number of iteration is varied, and finally Figure (g) - (i) plots the changes as the number of locks are varied.

lower bound on its performance. Figure 5.1 shows an example that targets at showing TREE in the best light. The program forks a number of threads running in a loop acquiring/releasing locks and writing to a shared address x. It is evident that there exists no race. Without TREE, FastTrack has to instrument and check every single event and track all the lock operations. Figure 5.2 shows the performance comparing FastTrack and FastTrack+TREE with the three parameters, *num_threads*, *num_locks* and *num_iters* varied. We observe a few interesting results. First, all the programs exhibited significant number of redundant events, which increased and became close to 100% as the value of the parameter increased. Second, FastTrack+TREE showed significant reduction in memory overhead as much lesser state needs to be tracked. Lastly, we observed that both the runtime and memory overhead improvements from running TREE were super-linear in many cases, indicating that TREE scales well.

```
          T_0
for(i=1; i< num_threads){
   x = x + 1
   fork(T_i)
}
          T_i
for(j = 1; j < num_iters){
  for(k = 1; k < num_locks) lock L_i[k]
  x = x + 1
  for(k = 1; k < num_locks)unlock L_i[k]
}
```

Figure 5.3: Sample program snippet that targets TREE's weakness.

Figure 5.3 shows an example that targets the weakness of TREE. There exists a single race in this program between the write of x in T_0 and any T_i. There are two differences between this program and the one in Figure 5.1: 1) the writes happen inside the loop that forks the new threads; 2) each thread acquires and releases a private lock array. This program particularly stresses the data-structures used in TREE since none of the prefixes matches in the Trie, and we have to create a new branch for each concurrency-context. However, we noticed that the program still showed reasonable performance, leading to a sub-linear memory increase up to 100MB when there were 100 threads and 100 private locks present. The runtime overhead was also reasonable ranging from 0 to 500ms. Surprisingly, we noticed that as the iterations increased, the number of redundant events increased, and very soon the benefits of filtering them out-weighted the negative effects, leading to overall reduction in runtime.

5.2 RDIT

We have implemented the RDIT algorithm in RVPredict [26], a recent race detector for multi-threaded Java programs based on ASM [12] and Z3 [14]. RVPredict allows us to perform a direct comparison between RDIT and three existing precise algorithms *Happens-Before* (HB) [32], *Causally-Precedes* (CP) [54], *Maximal-Causality* (MC) [26]), all of which have been implemented in RVPredict. RDIT aims to be useful for dynamic race detection in real-world programs where missing events are common due to instrumentation challenges and performance consideration. 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?

App	LoC	#Thrd	#Evnt	#RW	#Sync	#BP
ftpserver	32K	12	48K	34K	3K	5K
floodlight	68K	9	58K	33K	3K	11K
jigsaw	101K	9	15.6M	11M	0.6K	2.3M
subflow	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 5.2: Benchmarks and traces. The total size of all benchmarks is over 1.3M LoC. #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.

5.2.1 Evaluation Methodology

We compare RDIT with HB, CP, and MC on seven real-world large multi-threaded applications, including *Eclipse, Apache Derby, Jigsaw, Sunflow, Xalan, and Floodlight.* Table 5.2 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 [26] to cut the traces into smaller chunks (each with 10K events, default configuration in RVPredict), so that all techniques can finish within a reasonable time (1h). 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 4.

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 methods in each trace that contain synchronization events (*i.e., Lock/Unlock* and *ThreadFork/ThreadJoin*) and consider those method calls as BarrierPairs. 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 evaluate the runtime performance of RDIT with the *Xalan* benchmark. We choose *Xalan* because it is CPU intensive. 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 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 machine with 8GB memory and JDK 1.8 8GB heap space. All data were averaged over three runs.

5.2.2 Race Detection Results

Table 5.3 summarizes the results of race detection. For all the seven benchmarks, RDIT detected a total number of 85 races, all of which are true races. Comparatively, the other three techniques (HB, CP, and MC) reported 149, 149, and 213 false alarms, respectively. HB and CP reported 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 detected by each technique, HB

App	**#True**	**HB**	**CP**	**MC**	**RDIT**
ftpserver	24	37(**17**)	37(**17**)	56(**32**)	15(**0**)
floodlight	5	16 (**16**)	16 (**16**)	24 (**19**)	4 (**0**)
jigsaw	8	41 (**36**)	41 (**36**)	55 (**47**)	2 (**0**)
sunflow	1	6 (**5**)	6 (**5**)	6 (**5**)	1 (**0**)
xalan	56	22 (**2**)	22 (**2**)	57 (**1**)	52 (**0**)
derby	9	13 (**5**)	13 (**5**)	37 (**28**)	7 (**0**)
eclipse	9	72 (**68**)	72 (**68**)	90 (**81**)	4 (**0**)
Total:	112	207(**149**)	207(**149**)	325(**213**)	85(**0**)

Table 5.3: For each benchmark, the same incomplete trace after excluding all the synchronization events is used in all the four techniques: Happens-Before (HB), Causal-Precede (CP), Maximal-Causality (MC), and RDIT. For RDIT, the missing methods are set to those containing the excluded synchronization events. Column 2 reports the number of true races (those reported by MC based on the full trace). Columns 3-6 report the total number of races and false alarms reported by each technique on the incomplete trace. For all benchmarks, RDIT detected a total of 85 data races all of which are true races, while the other three techniques reported hundreds of false alarms.

and CP detected a total number of 58, and MC detected 112. Surprisingly, even with missing methods, RDIT detected 27 more true races than HB and CP, due to the power of the maximal causality model. For MC, although it detected more true races than RDIT (112 vs 85), it also reported an excessive number (213) of false alarms. Moreover, the results are consistent with and without using the reachable address computing optimization described in Section 4.4.3, because the optimization condition always holds in these benchmarks. We next discuss the results of several interesting benchmarks.

- **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 nine threads and 58K events. There are five true races identified by MC and RDIT without excluding any synchronization. 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. By contrast, RDIT reports 4 races all of which are true races. RDIT misses only one true race due to the BarrierPair model (because the two race events are both in a BarrierPair method).

- **Jigsaw:** This benchmark is a web server application that has been studied frequently in previous work [24, 26, 54]. There are eight 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 there are synchronizations contained in them).

- **Xalan:** This benchmark (collected from Dacapo [4]) 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 HB and CP (22), though HB and CP report only two false alarms. For MC on the other hand, it detects all the true races but also reports one false alarm. The number of false alarms is small because the majority of synchronizations in this benchmark are not protecting the race events.

- **Eclipse:** This benchmark contains JDT tests for the Eclipse IDE, also from Dacapo. There are nine true races detected on the full trace with ten threads and 16.6M events. All the three techniques (HB, CP, and MC) report a large number of false alarms (68, 68, and 81, respectively). RDIT still reports no false alarm but four out of the nine true races. The reason for the large number of false alarms, in contrast to that in *Xalan*, is that most conflicting events in

Excluded	Log Size			Time		
	Orig	**BP**	**BP+Opt**	**Orig**	**BP**	**BP+Opt**
a	1.3G	3.2G	1.4G	16.5s	44s(**+168%**)	18.6s(**+13%**)
a+b	1.3G	3.1G	1.4G	15.8s	39s(**+147%**)	16.6s(**+5%**)
a+b+c	1.1G	2.5G	1.2G	13.6s	31s(**+134%**)	15.3s(**+12%**)
a+b+d	652M	1.3G	711M	6.4s	11.3s(**+77%**)	6.7s(**+4%**)
a+b+c+d	548M	990M	598M	4.8s	8.6s(**+79%**)	5.2s(**+8%**)
a+b+c+d+e	489M	823M	537M	4.3s	7.1s(**+65%**)	4.5s(**+7%**)
(a) JDK libraries (`java.*,javax.*,com.*,sun.*`) **(b)** `org.dacapo.harness.*` **(c)** `org.apache.xpath.*` **(d)** `org.apache.xml.*` **(e)** `org.apache.xalan.*`						

Table 5.4: 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 naive execution of *Xalan* takes 0.36s.

Eclipse are properly protected by synchronizations.

5.2.3 Runtime Performance

Table 5.4 reports the runtime performance results. Overall, the runtime overhead of RDIT for capturing BarrierPairs in the *Xalan* benchmark ranges from 4%-13% with the reachable address optimization and 65%-168% without, and the space overhead for trace storage is even 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 additionally 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 44s 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 results strongly support the application of RDIT 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. CONCLUSIONS AND FUTURE WORK

We have developed enhancements to two crucial attributes of race detection tools:

1. Performance
2. Precision

The first observations we make it that there exists several redundancies in real-world program traces. Our online tool, TREE, precisely identifies these redundancies and filters them. The use of our tool does not result in the loss of any race, unlike previous approaches. We have also benchmarked TREE and found that in all cases, the increase in memory overheads are reasonable, making the use of our tool practical. The design of TREE is not limited just to HB based tools; it can readily be used with any dynamic analysis tool based on LockSet or even hybrid approaches. We plan release it into the RoadRunner tool chain. In the future, in addition to the memory access events, we may also look at removing redundant synchronization operations, although currently we observe that they make up a very small portion of the generated program trace.

The second observation we make is that race detection tools that claim to be precise, are less so in practice. RDIT 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 such 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 multi-threaded applications with zero false alarm, 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.

REFERENCES

[1] Sarita V. Adve and Hans-J. Boehm. “Memory Models: A Case for Rethinking Parallel Languages and Hardware”. In: *Commun. ACM* 53.8 (Aug. 2010), pp. 90–101. ISSN: 0001-0782.

[2] Pavol Bielik, Veselin Raychev, and Martin Vechev. “Scalable race detection for Android applications”. In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM. 2015, pp. 332–348.

[3] Swarnendu Biswas et al. “Valor: efficient, software-only region conflict exceptions”. In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM. 2015, pp. 241–259.

[4] Stephen M Blackburn et al. “The DaCapo benchmarks: Java benchmarking development and analysis”. In: *ACM Sigplan Notices*. Vol. 41. 10. ACM. 2006, pp. 169–190.

[5] Hans-J. Boehm. “How to Miscompile Programs with ”Benign” Data Races”. In: *Proceedings of the 3rd USENIX Conference on Hot Topic in Parallelism*. HotPar’11. Berkeley, CA, USA: USENIX Association, 2011, pp. 3–3.

[6] Hans-J Boehm. “Position paper: nondeterminism is unavoidable, but data races are pure evil”. In: *Proceedings of the 2012 ACM workshop on Relaxing synchronization for multicore and manycore scalability*. ACM. 2012, pp. 9–14.

[7] Michael D. Bond, Katherine E. Coons, and Kathryn S. McKinley. "PACER: Proportional Detection of Data Races". In: *SIGPLAN Not.* 45.6 (June 2010), pp. 255–268. ISSN: 0362-1340.

[8] Sebastian Burckhardt and Madanlal Musuvathi. "Effective program verification for relaxed memory models". In: *Computer Aided Verification*. Springer. 2008, pp. 107–120.

[9] Jacob Burnim, Koushik Sen, and Christos Stergiou. "Testing concurrent programs on relaxed memory models". In: *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ACM. 2011, pp. 122–132.

[10] IBM T.J. Watson Research Center. *T. J. Watson Libraries for Analysis(WALA)*. URL: `http://wala.sourceforge.net/wiki/index.php/Main_Page` (visited on 03/28/2016).

[11] Jong-Deok Choi et al. "Efficient and Precise Datarace Detection for Multithreaded Object-oriented Programs". In: *SIGPLAN Not.* 37.5 (May 2002), pp. 258–269. ISSN: 0362-1340.

[12] OW2 Consortium. *ASM for Java*. URL: `http://asm.ow2.org/` (visited on 03/28/2016).

[13] Heming Cui et al. "Parrot: A practical runtime for deterministic, stable, and reliable threads". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM. 2013, pp. 388–405.

[14] Leonardo De Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems.*

TACAS'08/ETAPS'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 337–340. ISBN: 3-540-78799-2, 978-3-540-78799-0.

[15] Joseph Devietti et al. "DMP: deterministic shared memory multiprocessing". In: *ACM SIGARCH Computer Architecture News*. Vol. 37. 1. ACM. 2009, pp. 85–96.

[16] Dimitar Dimitrov et al. "Commutativity Race Detection". In: *SIGPLAN Not.* 49.6 (June 2014), pp. 305–315. ISSN: 0362-1340.

[17] Laura Effinger-Dean et al. "IFRit: Interference-free Regions for Dynamic Data-race Detection". In: *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA '12. ACM, 2012, pp. 467–484. ISBN: 978-1-4503-1561-6.

[18] T Elmas, S Qadeer, and S Tasiran. "Goldilocks: A Race and Transaction-aware Java Runtime". In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '07. ACM, 2007, pp. 245–255. ISBN: 978-1-59593-633-2.

[19] Azadeh Farzan et al. "Predicting null-pointer dereferences in concurrent programs". In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM. 2012, p. 47.

[20] Cormac Flanagan and Stephen N Freund. "Adversarial memory for detecting destructive races". In: *ACM Sigplan Notices*. Vol. 45. 6. ACM. 2010, pp. 244–254.

[21] Cormac Flanagan and Stephen N. Freund. "FastTrack: Efficient and Precise Dynamic Race Detection". In: *SIGPLAN Not.* 44.6 (June 2009), pp. 121–133. ISSN: 0362-1340.

[22] Cormac Flanagan and Stephen N Freund. “Redcard: Redundant check elimination for dynamic race detectors”. In: *ECOOP 2013–Object-Oriented Programming*. Springer, 2013, pp. 255–280.

[23] Cormac Flanagan and Stephen 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*. ACM. 2010, pp. 1–8.

[24] Jeff Huang. “Stateless model checking concurrent programs with maximal causality reduction”. In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM. 2015, pp. 165–174.

[25] Jeff Huang, Qingzhou Luo, and Grigore Rosu. “Gpredict: Generic predictive concurrency analysis”. In: *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press. 2015, pp. 847–857.

[26] Jeff Huang, Patrick O’Neil Meredith, and Grigore Rosu. “Maximal sound predictive race detection with control flow abstraction”. In: *ACM SIGPLAN Notices* 49.6 (2014), pp. 337–348.

[27] Jeff Huang, Charles Zhang, and Julian Dolby. “CLAP: recording local executions to reproduce concurrency failures”. In: *ACM SIGPLAN Notices*. Vol. 48. 6. ACM. 2013, pp. 141–152.

[28] Jeff Huang, Jinguo Zhou, and Charles Zhang. “Scaling predictive analysis of concurrent programs by removing trace redundancy”. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 22.1 (2013), p. 8.

[29] *Java native interface specification.* `http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html/`. 2015.

[30] Baris Kasikci, Cristian Zamfir, and George Candea. "Data races vs. data race bugs: telling the difference with portend". In: *ACM SIGPLAN Notices* 47.4 (2012), pp. 185–198.

[31] Zhifeng Lai, S. C. Cheung, and W. K. Chan. "Detecting Atomic-set Serializability Violations in Multithreaded Programs Through Active Randomized Testing". In: *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1.* ICSE '10. New York, NY, USA: ACM, 2010, pp. 235–244. ISBN: 978-1-60558-719-6.

[32] Leslie Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System". In: *Commun. ACM* 21.7 (July 1978), pp. 558–565. ISSN: 0001-0782.

[33] Tongping Liu, Charlie Curtsinger, and Emery D Berger. "Dthreads: efficient deterministic multithreading". In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles.* ACM. 2011, pp. 327–336.

[34] Daniel Marino, M Musuvathi, and S Narayanasamy. "LiteRace: Effective Sampling for Lightweight Data-race Detection". In: *SIGPLAN Not.* 44.6 (June 2009), pp. 134–143. ISSN: 0362-1340.

[35] Nicholas D Matsakis and Thomas R Gross. "A time-aware type system for data-race protection and guaranteed initialization". In: *ACM Sigplan Notices.* Vol. 45. 10. ACM. 2010, pp. 634–651.

[36] Jeremie Miserez et al. "Sdnracer: Detecting concurrency violations in software-defined networks". In: *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research.* ACM. 2015, p. 22.

[37] Mayur Naik, Alex Aiken, and John Whaley. "Effective Static Race Detection for Java". In: *SIGPLAN Not.* 41.6 (June 2006), pp. 308–319. ISSN: 0362-1340.

[38] Satish Narayanasamy et al. "Automatically classifying benign and harmful data races using replay analysis". In: *ACM SIGPLAN Notices.* Vol. 42. 6. ACM. 2007, pp. 22–31.

[39] Robert H. B. Netzer and Barton P. Miller. "What Are Race Conditions?: Some Issues and Formalizations". In: *ACM Lett. Program. Lang. Syst.* 1.1 (Mar. 1992), pp. 74–88. ISSN: 1057-4514.

[40] Robert O'Callahan and Jong-Deok Choi. "Hybrid Dynamic Data Race Detection". In: *SIGPLAN Not.* 38.10 (June 2003), pp. 167–178. ISSN: 0362-1340.

[41] Kevin Poulsen. *Software bug contributed to blackout.* URL: `http://www.securityfocus.com/news/8016` (visited on 03/28/2016).

[42] Eli Pozniansky and Assaf Schuster. "Efficient on-the-fly data race detection in multithreaded C++ programs". In: *Parallel and Distributed Processing Symposium, 2003. Proceedings. International.* Apr. 2003, 8 pp. DOI: `10.1109/IPDPS.2003.1213513`.

[43] Eli Pozniansky and Assaf Schuster. "MultiRace: Efficient On-the-fly Data Race Detection in Multithreaded C++ Programs: Research Articles". In: *Concurr. Comput. : Pract. Exper.* 19.3 (Mar. 2007), pp. 327–340. ISSN: 1532-0626.

[44] Christoph von Praun and Thomas R. Gross. "Object Race Detection". In: *SIGPLAN Not.* 36.11 (Oct. 2001), pp. 70–82. ISSN: 0362-1340.

[45] Christoph von Praun and Thomas R Gross. "Static conflict analysis for multithreaded object-oriented programs". In: *ACM Sigplan Notices.* Vol. 38. 5. ACM. 2003, pp. 115–128.

[46] Arun K. Rajagopalan and Jeff Huang. "RDIT: Race Detection from Incomplete Traces". In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 914–917. ISBN: 978-1-4503-3675-8.

[47] Veselin Raychev, Martin Vechev, and Manu Sridharan. "Effective race detection for event-driven programs". In: *ACM SIGPLAN Notices*. Vol. 48. 10. ACM. 2013, pp. 151–166.

[48] Paul Rubens. *Software bug contributed Facebook IPO glitch*. URL: `http://www.cio.com/article/2378046/net/why-software-testing-can-t-save-you-from-it-disasters.html` (visited on 03/28/2016).

[49] Mahmoud Said et al. "Generating data race witnesses by an SMT-based analysis". In: *NASA Formal Methods*. Springer, 2011, pp. 313–327.

[50] Stefan Savage et al. "Eraser: A Dynamic Data Race Detector for Multithreaded Programs". In: *ACM Trans. Comput. Syst.* 15.4 (Nov. 1997), pp. 391–411. ISSN: 0734-2071.

[51] Koushik Sen. "Race Directed Random Testing of Concurrent Programs". In: *SIGPLAN Not.* 43.6 (June 2008), pp. 11–21. ISSN: 0362-1340.

[52] Konstantin Serebryany and Timur Iskhodzhanov. "ThreadSanitizer: data race detection in practice". In: *Proceedings of the Workshop on Binary Instrumentation and Applications*. ACM. 2009, pp. 62–71.

[53] Konstantin Serebryany et al. "Dynamic race detection with llvm compiler". In: *Runtime Verification*. Springer. 2012, pp. 110–114.

[54] Yannis Smaragdakis et al. "Sound predictive race detection in polynomial time". In: *ACM SIGPLAN Notices* 47.1 (2012), pp. 387–400.

[55] CORPORATE SPARC International Inc. *The SPARC Architecture Manual: Version 8*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1992. ISBN: 0-13-825001-4.

[56] Raja Vallée-Rai et al. "Soot-a Java bytecode optimization framework". In: *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press. 1999, p. 13.

[57] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. "RELAY: static race detection on millions of lines of code". In: *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM. 2007, pp. 205–214.

[58] Chao Wang et al. "Symbolic predictive analysis for concurrent programs". In: *FM 2009: Formal Methods*. Springer, 2009, pp. 256–272.

[59] Wikipedia. *An engineering disaster.* URL: `http://en.wikipedia.org/wiki/Therac-25` (visited on 03/28/2016).

[60] Yuan Yu, Tom Rodeheffer, and Wei Chen. "Racetrack: efficient detection of data race conditions via adaptive tracking". In: *ACM SIGOPS Operating Systems Review*. Vol. 39. 5. ACM. 2005, pp. 221–234.