RDIT: Race Detection from Incomplete Traces

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

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

ABSTRACT

We present RDIT, a novel dynamic algorithm to precisely detect data races in multi-threaded programs with incomplete trace information – the presence of missing events. RDIT enhances the classical Happens-Before algorithm by relaxing the need to collect the full execution trace, while still guaranteeing full precision. The key idea behind RDIT is to abstract away the missing events by capturing the invocation data of the missing methods. This provides valuable information to approximate the possible synchronization behavior introduced by the missing events. By making the least conservative approximation that two missing methods introduce synchronization only when they access common data, RDIT guarantees to detect a maximal set of true races from the information available. We have conducted a preliminary study of RDIT on a real system and our results show that RDIT is promising; it detects no false positive when events are missed, whereas Happens-Before reports many.

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

General Terms: Algorithms, Design, Theory

Keywords: Missing Trace, Data Race, Happens-Before, Precise, Reachable Addresses

1. INTRODUCTION

One of the most serious problem in today's concurrent software systems is probably data races. They manifest nondeterministically, often appearing only on very rare executions and have caused many real world problems. These include the Therac-25 medical accidents [7] and the 2003 blackout in the USA and Canada [12]. To detect data races, researchers have proposed a wide range of techniques, both static and dynamic, targeting different types of software at various stages of the software development process. Most detectors are based on one of three techniques: LockSet [9], Happens-Before [6], or a combination of the two.

ESEC/FSE'15, August 30 – September 4, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3675-8/15/08...$15.00
http://dx.doi.org/10.1145/2786805.2803209

A crucial issue in these tools are false alarms. Because data races are difficult to diagnose and validate, any false alarms could significantly decrease programmer productivity and make the tool less useful. However, it remains highly challenging to develop a false-alarm-free race detection technique. The general problem of precisely identifying all data races is NP-hard [8], and the LockSet algorithm is known to be incomplete. The difficulty comes not only from the algorithmic complexity, but also from various practical issues. Although Happens-Before (HB) is precise theoretically, in practice, HB-based techniques tend to report many false positives.

1.1 Missing Events

A primary practical factor causing false positives are missing events. Happens-Before is often applied on a dynamic execution trace of events performed by the threads. To guarantee preciseness, Happens-Before requires that the trace information be complete, that is, all critical events are captured. This is however, a strong requirement for large scale programs and is not easily achievable. Sometimes, we may even desire to miss certain events for performance reasons. Some common situations where we miss events are:

- **External libraries** These may be loaded in a different language or even on the fly over the network where we do not have access to add necessary instrumentation to capture the events.
- **Performance sensitive applications** Such applications may hand-off execution of critical sections to an optimized external program usually written in a lower level language.
- **System calls** Calls into the host operating system's libraries. Some examples include sending and receiving data over the network bus in MPI programming. These calls are inherently synchronized.
- **Localized debugging** Programmers may be interested in debugging a specific section of the application for data races and can choose to skip instrumenting large sections of their program to speed up run-time.

In all these situations, we may end up missing vital trace information. If the missing section contains synchronization primitives that would have led to a HB edge, Happens-Before based tools would generate false positives.

Example. Consider the trace in Figure 1. We have two threads T1 and T2 performing a Write (event e1) and Read (event e2) on a common address X. The grayed out region in-between the two events $e1$ and $e2$ is the region of interest where we would like to check for any synchronization. The

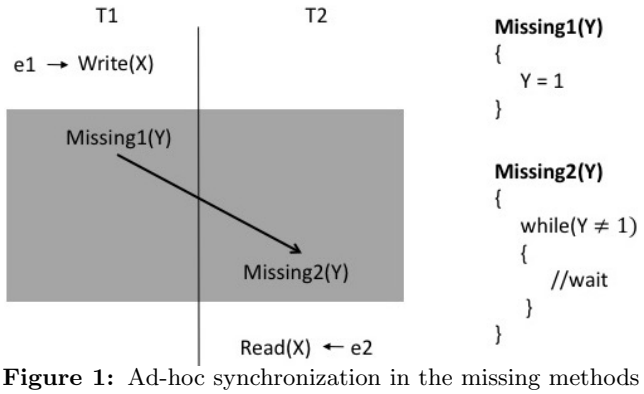

Figure 1: Ad-hoc synchronization in the missing methods results in false positives reported by Happens-Before.

synchronization can either be in the form of a HB edge inducing event such as LOCK/UNLOCK, FORK/JOIN, or an ad-hoc synchronization which causes an ordering in the program execution. In the absence of any such synchronization, we will flag $e1$ and $e2$ as a race. When all computations in this region are missed, $e1$-$e2$ will be reported as a race.

However, this is a false alarm when the two missing methods introduce an ad-hoc synchronization on a shared address Y (set to 0 initially). Thread T1, after performing the write to X, sets $Y = 1$. Before thread T2 can perform the Read of X, it waits while the value of $Y \neq 1$. Thus, Read(X) from thread T2 can occur only after the Write(X) from thread T1. The shared address Y is used as a barrier in thread T2 to induce a desired ordering.

1.2 Our New Idea

In this paper, we explore an enhancement to the Happens-Before algorithm that relaxes the need to collect complete trace information, that is, to precisely detect data races in the presence of missing events. A naive technique to avoiding false positives would involve adding HB edges between all missing events that occur on different threads. Although this approach is guaranteed to detect no false positives, it is overly conservative and misses a lot of true data races, as we shall see below.

Our new algorithm, RDIT, uses program analysis to reason about the missing events and provides the guarantee that all detected races are real races, at the cost of fewer detected races. However, as compared to the naive approach, RDIT detects a maximal set of true data races from the available program trace. Our key observation is that although the computations inside the missing methods cannot be instrumented, we can usually capture the invocation of those missing methods. The runtime data at the invocation sites actually provides valuable information to approximate the behavior of the missing methods.

Consider our example in Figure 1. Both missing methods 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 in threads T1 and T2 reach addresses A and B, respectively, and if $A \neq B$, then we can safely conclude that no ordering can be induced in T1 and T2 through this pair of missing methods.

This observation leads to our first contribution - gathering a set of *reachable addresses* at the invocation of each missing method. We abstract each external library or missing method call as two events: $MethodBegin$ and $MethodEnd$. Only if the reachable addresses of two missing methods overlap, we add the necessary HB edges on the abstracted events. With this enhancement, the same Happens-Before algorithm can be applied to detect races without any change.

We have conducted a preliminary study on a popular database system – Apache Derby, and our results show that our algorithm is promising: it detects **zero** false positives, whereas the original Happens-Before algorithm reports many false alarms.

We next introduce necessary background on race detection with Happens-Before, then describe our RDIT algorithm in more detail.

2. DATA RACES AND HAPPENS-BEFORE

A data race occurs when there are unordered conflicting accesses in the program without proper synchronization. Happens-Before analysis has been implemented in many state of the art dynamic race detection tools such as ThreadSanitizer [11], FastTrack [3], and RVPredict [4]. These tools usually have two phases - an instrumentation phase that generates an execution trace, and an analysis phase that takes the trace as an input and detects races, either online or offline. All necessary information needed for race detection is collected during the instrumentation phase. This includes several program events such as READ-/WRITE, LOCK/UNLOCK, WAIT/NOTIFY, etc. In addition, various run-time parameters such as thread ID, memory address, etc., are also collected to aid the analysis. The amount of information gathered in this phase is limited by the run-time slowdown we are able to tolerate. Thus, the goal during instrumentation is to minimize the run-time overhead while collecting all the information needed for the analysis.

The gathered execution trace is then fed to the analysis engine. Happens-Before analysis proceeds by building a Directed Acyclic Graph (DAG) representing the trace. The vertices of this graph correspond to program events such as READ/WRITE while the edges correspond to HB edges. A HB edge is added between two events if they are performed by the same thread or there is some synchronization (such as FORK/JOIN, LOCK/UNLOCK, WAIT/NOTIFY) between them. If two conflicting events (a READ-WRITE or a WRITE-WRITE event pair) do not have a path between them in the DAG, then the two events are reported as a race.

In practice, Happens-Before is typically implemented using vector clocks, or its variants [3], to realize the principle of Lamport clocks [6]. In addition to the program order, synchronization events result in HB edges and cause the vector clock to update. Every conflicting access to shared memory leads to a check against the vector clocks of the threads involved. If the vector clocks of two conflicting accesses are not comparable, meaning that the two accesses are not ordered, then a data race is detected.

3. ALGORITHM

In our new algorithm, every missing method comprises of two events - a $MethodBegin$ and a $MethodEnd$. The instrumentation phase is tasked with gathering this information. We refer to this pair of events as a BarrierPair.

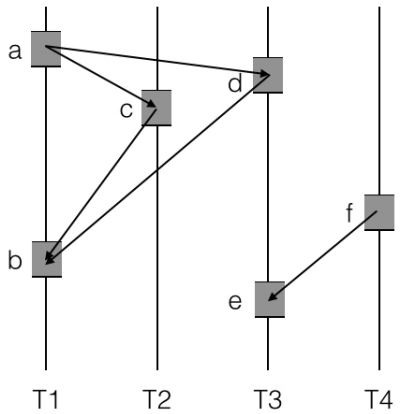

Figure 2: A program trace consisting of 4 threads and 6 BarrierPairs. HB edges are added between the BarrierPairs if they have overlapping reachable addresses.

For example, Figure 2 illustrates six BarrierPairs in a trace - a, b, c, d, e and f. The shaded region indicates the region of program execution where missing events are encountered. A BarrierPair contains the following attributes:

- A thread-ID denoting the thread that called the missing method.
- A $MethodBegin$ event corresponding to the invocation of the missing method.
- A $MethodEnd$ event corresponding to the return of the missing method.
- A set of addresses that can be reached by this BarrierPair.
- A (possibly empty) set of recorded events that occur in-between the begin and end events of the BarrierPair for the particular thread.

RDIT, an extension of the Happens-Before algorithm, is described in Algorithm 1. If the trace is missing events (synchronization events in particular), then the DAG constructed by Happens-Before will have missing edges between its vertices, giving us false positives. Since we aim to achieve no false positive, we conservatively approximate the synchronization behavior of the missing methods, and add a HB edge between missing methods that share at least one common address in its reachable address set. While this approach is conservative, it is the least conservative one, because as long as the reachable address sets of two missing methods intersect, they may use the intersected address to synchronize. In other words, our algorithm guarantees that we detect a maximal set of true races from the information available.

Our algorithm proceeds in two phases - the first phase gathers all the BarrierPairs from the program trace and the second phase adds HB edges between BarrierPairs from different threads that have at least one common reachable address. Associated with each thread is a *BeginStack*. This stack stores the $MethodBegin$ events encountered by this particular thread. For every $MethodBegin$ event, we push onto the stack, and for every $MethodEnd$ event encountered, we pop from the top of stack. This ensures that a correct BarrierPair is constructed. For all other events, we leave the original Happens-Before algorithm unmodified. At the end of the first phase, we would have constructed a DAG from the execution trace containing HB edges between synchronization events that were recorded.

Algorithm 1 The RDIT Algorithm

```
1: input ← τ // input trace
2: bpArray ← ∅ // Initialization
3: for all threads t in τ do
4:     t.BeginStack ← ∅
5: end for
6:
7: // Process trace and gather BarrierPairs
8: for all events e in τ do
9:     t ← e.tid // thread ID
10:    if e = MethodBegin then
11:        t.BeginStack.push(e)
12:    else if e = MethodEnd then
13:        mBegin ← t.BeginStack.pop()
14:        bp ← new BarrierPair(mBegin, e)
15:        bpArray.add(bp)
16:    else
17:        // Process normal HB events
18:    end if
19: end for
20:
21: // Add HB edges between BarrierPairs
22: for all (bp1, bp2) in bpArray do
23:    if bp1.tid ≠ bp2.tid then
24:        if bp1.addr ∩ bp2.addr ≠ ∅ then
25:            addHBEdge(bp1.events, bp2.events)
26:        end if
27:    end if
28: end for
29:
30: // Call the original HappensBefore algorithm
31: for all (e1, e2) in τ do
32:    if (e1, e2) conflict then
33:        HappensBefore(e1, e2)
34:    end if
35: end for
```

In order to account for the missing events, the second phase then takes this DAG as input and iterates through all pairs of BarrierPairs. We add HB edges between the first BarrierPair from each of the different threads that satisfies the criteria of at least one common reachable address. Since HB is transitive, it suffices to add HB edges on the begin/end events. The added HB edge is always in the direction of increasing trace location, i.e., a HB edge between two events $e1$ and $e2$ goes from $e1 \rightarrow e2$ if $e1$ occurs before $e2$ in the trace. Figure 2 illustrates a trace with HB edges added between BarrierPairs that intersect.

Finally, we perform race detection using the original HB algorithm. A pair of nodes are said to be conflicting if they are either a READ-WRITE or a WRITE-WRITE pair. We look at all such pairs of conflicting events $(e1, e2)$ in the trace and check if there exists a path from $e1 \rightarrow e2$ in the DAG. If such a path does not exist, then we report a race.

In the preceding algorithm analysis, we have made the assumption that the addresses that are used to perform synchronization are local in scope i.e., they are passed in as a parameter 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 as their inclusion would reduce RDIT to the naive approach. Although global addresses could then potentially result in false positives reported by our algorithm, this is a minor concern since such programming practices are discouraged and are rarely seen in large production grade programs.

4. PRELIMINARY RESULTS

Implementation We have implemented RDIT in RVPredict [4], a dynamic race detection tool for Java programs. Instrumentation is performed using ASM [1]. During instrumentation, we insert logger methods into source byte-code that records all the necessary events. The set of 'reachable addresses' of a particular method is computed as the union of the set of reachable addresses of each of its parameters. To calculate the 'reachable address' set of a particular object, we perform a breadth-first search through its declared fields and inheritance stack. Each unique address is pushed onto a queue. We iterate this process until the queue is empty. The set of addresses we gather in this way is the complete set of addresses reachable by that particular method.

Results We evaluated our tool against Apache Derby, a widely used open source Java database management system. Since the application is large, we can easily simulate missing events. To simulate the condition of missing events, we randomly exclude certain classes and packages from being instrumented. Table 1 shows the reported number of races by Happens-Before and our RDIT algorithm with various number of randomly missed classes. All experiments were conducted on an 8-core Linux machine with OpenJDK 1.7.0 and 32GB heap space.

Table 1: Results on Apache Derby Database

Missing Classes	HB	RDIT	**RDIT False Positives**
0	4	4	**0**
22	67	3	**0**
23	63	4	**0**
27	69	4	**0**
29	64	4	**0**

Our results show that when no class is excluded, both Happens-Before and RDIT report 4 races. However, when 22-29 random classes were excluded, a large number (63-69) of races are reported by Happens-Before, whereas RDIT consistently reported 3-4 races only. We also manually inspected these races and found that all races reported by RDIT were real, that is, *RDIT reported no false positive.*

On the contrary, we found many false alarms among those 63-69 races reported by Happens-Before. Moreover, for some races, it was very difficult and time-consuming to determine their validity, and as such we were not able to make a conclusive judgment.

5. RELATED WORK

Data race detection has been widely discussed in the literature. Our technique is distinguished in that it is the first to address the practical problem of missing events. Although Happens-Before is precise, its guarantee of no-false-positive is only true when the entire execution trace is available.

Several techniques [10, 5, 2] exist to improve accuracy of the detected races through runtime validation. These techniques take a set of potential races as input and execute the program again trying to simulate the inter-leavings necessary to induce the race. If the conditions to reproduce the race are not met, the race is marked as $NoRace$ and not reported. While these techniques can prune false positives, they require multiple runs of the program, and thus suffer from livelocks and missed true races.

6. ACKNOWLEDGMENT

We would like to thank the anonymous reviewers for their constructive comments. This research is supported by faculty start-up funds from Texas A&M University and a Google Faculty Research Award to Jeff Huang.

7. CONCLUSION AND FUTURE WORK

We have presented a novel enhancement to the classical Happens-Before algorithm that ensures preciseness in situations when the trace information is incomplete – missing events. Our algorithm requires only a single run of the program and guarantees no false positive. We have implemented RDIT in Java and conducted a preliminary evaluation on a large database application. Our results show that our algorithm detects only real races even when arbitrary sections of code are missed.

Our study opens several interesting directions for future work. First, we plan to formalize the BarrierPair model of our algorithm and establish it's theoretical soundness. Second, we plan to fully realize our algorithm and evaluate its performance on several larger real world multi-threaded systems. Third, we plan to optimize runtime performance and integrate our algorithm in popular race detection tools with large user base such as ThreadSanitizer.

8. REFERENCES

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

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

[3] C. Flanagan and S. N. Freund. Fasttrack: efficient and precise dynamic race detection. In *PLDI*, 2009.

[4] J. Huang, P. O. Meredith, and G. Rosu. Maximal sound predictive race detection with control flow abstraction. In *PLDI*, 2014.

[5] J. Huang and C. Zhang. PECAN: Persuasive Prediction of Concurrency Access Anomalies. In *ISSTA*, 2011.

[6] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *CACM*, 1978.

[7] J. Lim. An engineering disaster: Therac-25. *http://en.wikipedia.org/wiki/Therac-25*, 1998.

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

[9] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *TOCS*, 1997.

[10] K. Sen. Race directed random testing of concurrent programs. In *PLDI*, 2008.

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

[12] S. B. C. to Blackout. Securityfocus. *http://www.securityfocus.com/news/8016*, 2004.