Effective and Precise Dynamic Detection of Hidden Races for Java Programs

Yan Cai†
State Key Laboratory of Computer Science
Institute of Software
Chinese Academy of Sciences, Beijing, China
ycai.mail@gmail.com

Lingwei Cao
State Key Laboratory of Computer Science
Institute of Software
Chinese Academy of Sciences, Beijing, China
lingweicao@gmail.com

ABSTRACT
Happens-before relation is widely used to detect data races dynamically. However, it could easily hide many data races as it is interleaving sensitive. Existing techniques based on randomized scheduling are ineffective on detecting these hidden races. In this paper, we propose DrFinder, an effective and precise dynamic technique to detect hidden races. Given an execution, DrFinder firstly analyzes the lock acquisitions in it and collects a set of "may-trigger" relations. Each may-trigger relation consists of a method and a type of a Java object. It indicates that, during execution, the method may directly or indirectly acquire a lock of the type. In the subsequent executions of the same program, DrFinder actively schedules the execution according to the set of collected may-trigger relations. It aims to reverse the set of happens-before relation that may exist in the previous executions so as to expose those hidden races. To effectively detect hidden races in each execution, DrFinder also collects a new set of may-trigger relation during its scheduling, which is used in its next scheduling. Our experiment on a suite of real-world Java multithreaded programs shows that DrFinder is effective to detect 89 new data races in 10 runs. Many of these races could not be detected by existing techniques (i.e., FastTrack, ConTest, and PCT) even in 100 runs.

Categories and Subject Descriptors
D.2.4 [Software Engineering]: Program Verification; D.2.5 [Software Engineering]: Testing and Debugging, testing tools; D.4.1 [Operating Systems]: Processing Management – synchronizations, threads.

General Terms
Reliability, verification.

Keywords
Data race, thread scheduling, hidden race, synchronization order

1. INTRODUCTION
A data race (or race for short) [17] occurs when two or more threads access a same memory location concurrently, and at least one of these accesses is a write [17]. Data race occurrences often indicate other concurrency bugs in the same program [34]. Many dynamic data race detectors are based on the locking discipline [40].

† Corresponding author.

This is the author’s version of the work. It is posted here for your personal use. Not for redistribution. The definitive version was published in the following publication:
ESEC/FSE’15, August 30 – September 4, 2015, Bergamo, Italy
ACM, 978-1-4503-3675-8/15/08...
http://dx.doi.org/10.1145/2786805.2786839

or the happens-before relations (HBR for short) [25]. The locking discipline requires every two concurrent accesses (one of them is a write) to a shared memory location to be protected by a common set of locks. But, such lockset-based detectors are imprecise [18]. HBR-based detectors (or HB detectors for short) precisely [17] report a data race only if they observe the two accesses involving in a race not ordered by any HBR in an execution/trace. (In this paper, we use the two terms execution and trace interchangeably.)

Some races in a program can be easily exposed in many traces and HB detectors [17][35][37] can effectively detect them. There are other races that are difficult to be detected due to reasons like conditional variables and ad-hoc synchronizations. They can be detected by two most recent techniques RVPredict [20] through data flow analysis offline and Racageddon [16] through generating specific test inputs for each predicted race.

HB detectors are interleaving-sensitive [49]. They may miss to detect a race if the two accesses of this race are ordered by HBRs in an execution; but the same race can be detected in another execution with a different thread interleaving such that no HBR orders the two accesses. That is, such a race is hidden by HB edges in some executions [43]. Some of these races, even on repeated executions, can still be hard to detect [43]. For ease of reference, we refer to such a race as a hidden race (also known as a "hard" race [43]). This paper focuses on the detection of hidden races.

Existing online techniques (e.g., [8][15][41][50]) are ineffective to detect hidden races. Randomized scheduling techniques (e.g., PCT [8] and ConTest [15]) only randomly identify changed points [8] or insert random time delays [15] to modify thread priorities. They are ineffective in exposing races whose accesses are separated by consecutive sequences of locking orders [25] among threads. Active testing (e.g., [41][50], Racageddon and RVPredict) techniques are built on top of random (or native) scheduling and/or concurrency bug patterns to produce a predictive trace for potential race analysis. It fails to be successfully applied if no hidden race can be predicted in the predictive run. Besides, they need many runs to deem a potential race as a false positive with confidence. Coverage-driven testing techniques [50] or using adequacy criteria [53] demand either patterns of problematic memory accesses as well as synchronization operations or applicable adequacy criteria as inputs. To the best of our knowledge, there is no effective pattern or adequacy criterion for hidden races discovered yet.

As such, a key challenge for dynamic race detectors is to generate executions that effectively expose hidden races.

Offline techniques [20][43] to may infer hidden races. The Causally-Precedes (CP) detector [43] can interestingly predict hidden races in a given trace under limited scenarios. CP is however inapplicable if a hidden race is separated by HBR (i.e., HB edges) having conflicting data accesses [43]. Also, analyzing large traces (e.g.,
running Eclipse for 1 hour of code development) by CP is still impractical; besides, no online CP detector has been invented yet [43]. RVPredict [20] confirms each predicted data race via constraint solving. Like other predictive techniques [41], it needs to solve the scheduling constraints for each predicted data race, which may fail.

Our work exploits two observations: (1) many races hidden in one execution can be detected by reversing the direction of one or more consecutive HBRs [25] in another execution, and (2) in real-world programs, only a small proportion of methods generates lock acquisition events, and these methods usually generate events on designated (instead of arbitrary) lock objects.

In this paper, we propose DrFinder (Data races Finder), a dynamic technique to detect hidden races by reversing possible HBRs. DrFinder is based on may-trigger relation. This relation relates a method to a type of lock object in Java programs. It represents that the method may directly or indirectly (by calling several other methods) trigger a lock acquisition on a lock object of that type.

DrFinder consists of two phases. Figure 1 shows an overview of DrFinder. In Phase I, DrFinder analyzes each lock acquisition event in a given trace to relate each selected method to the type of the lock object of the event and takes them as a set of may-trigger relations (i.e., MTR in Figure 1). In Phase II, it generates a trace for hidden race detection based on the collected set of may-trigger relation. Specifically, if a thread generates a lock acquisition event $e_1$ on a lock of type $c$ and some other thread may-trigger a lock acquisition event $e_2$ on a lock of the same type $c$, DrFinder postpones the execution of $e_1$ until an expected $e_2$ occurs. As such, the locking order on these two events that they may form in the trace analyzed in Phase I is reversed. In this way, the races originally hidden by such HBRs are exposed in the later trace.

DrFinder also collects a set of new may-trigger relations in each scheduled execution which is used in the next scheduled execution. This feedback mechanism makes DrFinder effective to detect new hidden races in each its scheduled execution.

We have implemented DrFinder in the Jikes RVM [3], and evaluated it on the Dacapo benchmarks [6]. The experimental result shows that DrFinder finds races that cannot be effectively detected by both native runs, ConTest, and PCT configured with FastTrack. In total, DrFinder detects 89 new data races on 5 Dacapo benchmarks within 10 runs each. Besides, many of these new races could not be detected by existing techniques in 100 runs.

In summary, the main contributions of this paper are:

- This paper proposes a may-trigger relation and a novel hidden data race detector DrFinder. DrFinder predicts locking orders, and makes decisions on the reversals of locking orders at the may-trigger relation level. It profiles no memory accesses in Phase I, and only carries forward the may-trigger relations between phases.
- It reports the feasibility of DrFinder by implementing it as a prototype tool in the Jikes RVM. It presents an experiment to evaluate DrFinder. In the experiment, DrFinder detects 89 new races and is promising in exposing hidden races. DrFinder is scalable to large-scale Java programs (e.g., Eclipse).

In the rest of this paper, Section 2 reviews preliminaries followed by a motivating example in Section 3. Sections 4 and 5 present the design rationales and the details of DrFinder, respectively. Section 6 reports the evaluation of DrFinder. In Sections 7 and 8, we discuss the related work and conclude this paper, respectively.

### 2. PRELIMINARIES

A multithreaded Java program $p$ contains a set of classes denoted as $C$. Each class $c$ in $C$ consists of a set of fields and a set of methods. We denote the set of all methods of $p$ as $M$. An object $o \in O$ is an instance of a class $c$, and the type of $o$, denoted as $type(o)$, is $c$. Each lock is an instance of a class.

Each thread $t$ in program $p$ executes a nested sequence of methods. Each method may execute a set of operations $Op = \{rd, wr, acq, rel, enter, return\}$, where $rd$ and $wr$ mean read and write to a field of a class instance, respectively; $acq$ and $rel$ mean acquisition and release of a lock, respectively; and $enter$ and $return$ mean a call and a return to and from a method, respectively.

An event $e = \langle t, op, o \rangle$ means that a thread $t$ performs an operation $op \in Op$ on an object $o \in O \cup M$. We denote $o \in e$ by $object(e)$. A trace $\tau$ is a sequence of events.

The Happens-before relation (→, HBR) [25] in a trace is defined by three rules: (1) if two events $\alpha$ and $\beta$ are performed by the same thread, and $\alpha$ appeared before $\beta$, then $\alpha \rightarrow \beta$. (2) if two events $\alpha = \langle t, \alpha, \tau, m \rangle$ and $\beta = \langle t, \beta, \alpha, m \rangle$ are performed by different threads, and $\alpha$ appeared before $\beta$, then $\alpha \rightarrow \beta$. (3) if $\alpha \rightarrow \gamma$ and $\beta \rightarrow \gamma$, then $\alpha \rightarrow \beta$.

Two memory events $e_1 = (t_1, op_1, v_1)$ and $e_2 = (t_2, op_2, v_2)$ form a race on $v_i$ if ($1$) $t_1 \neq t_2 \wedge v_1 = v_2$, ($2$) $wr \cap (op_1, op_2) \neq \emptyset$, and ($3$) neither $e_1 \rightarrow e_2$ nor $e_2 \rightarrow e_1$.

The relation $e_1 \rightarrow e_2$ is called an HB edge if $e_1 = (t_1, rel, l), e_2 = (t_2, acq, l)$, and $t_1 \neq t_2$ [43]. For instance, the two arrows in Figure 2 depict two HB edges $e_1 \rightarrow e_6$ and $e_2 \rightarrow e_6$, where "sync(o) […]" denotes a pair of events "acq(o) ... rel(o) "; and we will use this short form in Section 3. Given two traces $\tau$ and $\tau'$ and a pair of events $e_1$ and $e_2$, if $e_1$ and $e_2$ form a race in $\tau$ but does not form any race in $\tau'$, then the race is called a Hidden Race in $\tau'$.

In a Java program, each thread starts its execution from its method run().

### 3. MOTIVATING EXAMPLE

Figure 2 shows our motivating example, where, each thread (i.e., $t_1$, $t_2$, or $t_3$) executes a sequence of events from top to bottom. These events are memory accesses (i.e., $m_1$ to $m_4$) to the locations $x$ and $y$, and lock acquisition/release events (i.e., $e_1$ to $e_9$) on seven lock objects $k$, $n$, and $o_1$ to $o_2$. The two HB edges $e_1 \rightarrow e_6$ and $e_3 \rightarrow e_9$ on the two lock objects $n$ and $k$, respectively, are denoted as arrows. We denote the trace as $\tau_1 = \langle \ldots, e_1, \ldots, e_9, \ldots, e_3 \ldots, e_6 \ldots \rangle$. The two pairs of accesses to locations $x$ and $y$ (i.e., $m_1$}
and \(m_4\), \(m_2\) and \(m_3\) are ordered by the HB edges \(e_1 \rightarrow e_9\) and \(e_3 \rightarrow e_6\), respectively. Hence, no HB detector can detect any race in the trace \(\sigma_1\).

Next, we reverse the direction of each HB edge shown in Figure 2 to sketch Figure 3, which represents the trace \(\sigma_2 = (\ldots, e_9, \ldots, e_1, \ldots, e_6, \ldots, e_3, \ldots)\) of the same program. An HB detector can now report the two (hidden) races on \(x\) and \(y\) in \(\sigma_2\).

If a native schedule exhibits the trace \(\sigma_1\) as shown in Figure 2, the probability to observe the trace \(\sigma_2\) shown in Figure 3 is very low [43]. Hence, the two races may not be easily detected.

**Naive strategy:** A naive strategy is to suspend every lock acquisition event observed in a trace before executing the event. For trace \(\sigma_2\), it suspends the three threads from executing \(e_1\), \(e_4\), and \(e_7\), respectively, producing an occurrence of the suspension of all the threads of the trace (known as thrashing [41]). Thrashing is typically resolved by randomly resuming one of the suspended threads. There is no thoughtful design to ensure \(e_9 \rightarrow e_1\) to expose the hidden race on \(x\) effectively.

**Offline techniques:** CP [43] can infer these two races from \(\sigma_1\), providing that no conflicting memory accesses exist inside the synchronization bodies of \(e_1\) and \(e_9\). That is, if \(z\) is a new location, where \(e_1\) protects a write access to \(z\) (i.e., \(e_1 = \text{sync}(k)\{\ldots, \text{wr}(z)\ \ldots \})\) and \(e_9\) protects a read access to \(z\) (i.e., \(e_9 = \text{sync}(k)\{\ldots, \text{rd}(z)\ \ldots \})\). Then, the race on \(x\) cannot be detected by CP (and the case on \(y\) is similar). This is restrictive. In their experiment [43], on only 2 out of 11 programs, can CP detect 2 and 7 more races than FastTrack [17] (an online HB detector).

**Online randomized schedulers:** ConTest [15] inserts a small amount of random time delays on some lock acquisitions. Suppose that \(e_4\) is generated but not executed by \(t_4\) yet. A small time delay in between the generation and execution of \(e_4\) may not be long enough for \(t_4\) to have generated and executed \(e_9\), which depends on both the underlying (native or randomized) scheduler and the sequence of operations performed by \(t_4\) in between the current execution point and \(e_9\). The design of ConTest is insensitive to both factors. PCT [8] provides a theoretical guarantee to find a concurrency bug, but this guarantee is very low even for the illustrating example. Its guaranteed probability (i.e., \(1 + (n \times k)^{\text{BugDepth}-1}\))

**Figure 2.** A trace \(\sigma_1\) hiding two races on \(x\) and \(y\) as two HB edges \(e_1 \rightarrow e_9\) and \(e_3 \rightarrow e_6\), order the two accesses of each race, respectively.

**Figure 3.** A trace \(\sigma_2\) generated by DrFinder, exposed two races on \(x\) and \(y\).

Table 1. The May-Trigger relation for methods and lock object types of program shown in Figure 2.

<table>
<thead>
<tr>
<th>May-Trigger Relation</th>
<th>May-Trigger Relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{mt}(f_1, c_o)), (\text{mt}(f_2, c_1)), (\text{mt}(f_1, c_6))</td>
<td>(\text{mt}(f_5, c_o))</td>
</tr>
<tr>
<td>(\text{mt}(f_2, c_1)), (\text{mt}(f_3, c_3)), (\text{mt}(f_5, c_6))</td>
<td>(\text{mt}(f_2, c_o), \text{mt}(f_3, c_3), \text{mt}(f_5, c_6))</td>
</tr>
<tr>
<td>(\text{mt}(f_5, c_2)), (\text{mt}(f_4, c_3)), (\text{mt}(f_3, c_4))</td>
<td>(\text{mt}(f_2, c_1), \text{mt}(f_3, c_4), \text{mt}(f_5, c_6))</td>
</tr>
<tr>
<td>(\text{mt}(f_4, c_2)), (\text{mt}(f_4, c_3)), (\text{mt}(f_4, c_4))</td>
<td>(\text{mt}(f_5, c_2))</td>
</tr>
</tbody>
</table>

Next, it is feasible for \(t_4\) to execute \(e_9\) or for \(t_2\) to execute \(e_4\). Because neither \(\text{mt}(f_4, \text{type}(c_2))\) nor \(\text{mt}(f_2, \text{type}(c_3))\) matches any may-trigger relation, DrFinder resolves the tie randomly: (1) Suppose that \(t_3\) is selected. When \(t_3\) generates \(e_9\), DrFinder finds that \(\text{mt}(f_4, \text{type}(c_2))\) is a may-trigger relation. Thus, it suspends \(t_1\), and escorts \(t_2\) to execute until \(t_3\) has executed \(e_6\). After that, \(t_4\) executes \(e_2\), and the HB edge \(e_9 \rightarrow e_2\) is formed. When \(t_2\) executes \(m_3\), the race on \(y\) is detected. (2) Suppose that \(t_2\) executes \(e_4\) first.
When \( t_2 \) further generates \( e_5 \), \textit{DrFinder} finds that \( m(t_2, \text{type}(e_5)) \) does not match any \textit{may}-\textit{trigger} relation. Thus, both threads \( t_1 \) and \( t_2 \) may proceed further. So, \textit{DrFinder} resolves the tie randomly. If \( t_1 \) is selected to execute first, a race on \( y \) is detected. Otherwise, no race is reported because the HB edge between \( e_5 \) and \( e_6 \) is still \( e_3 \leftrightarrow e_6 \).

### 4. Design Rationales

In this section, we present the design rationales of \textit{DrFinder}, with the help of two traces \( \sigma_1 \) and \( \sigma_2 \), and their corresponding set of HB edges are \( \Delta_1 \) and \( \Delta_2 \), respectively. Besides, there are two threads \( t_1 \) and \( t_2 \) in two traces and they produce two events \( e_1 \) and \( e_2 \), respectively, in trace \( \sigma_1 \); however, the two events may not be produced by two threads in trace \( \sigma_2 \).

#### 4.1 Basic Requirements

We recall that a hidden race is difficult to expose in a trace generated by a native scheduler or a pure randomized scheduler. That is, although the two accesses involving in a hidden race may appear in a trace, yet the pair of accesses may be separated by non-trivial numbers of HB edges (e.g., the two accesses \( m_1 \) and \( m_4 \) in Figure 2). For ease of reference, we refer to such a native or pure randomized scheduler as a default scheduler.

A strategy modeled after the above intuition is as follows: In Phase I, a technique observes the set of HB edges \( \Delta_1 \) in the trace \( \sigma_1 \) produced by a default scheduler. Then, in Phase II, it aims to reverse the directions of some HB edges in \( \Delta_1 \) on generating the trace \( \sigma_2 \). That is, if the two events \( e_1 \) \( \Rightarrow \) \( e_2 \) \( \in \Delta_1 \), it aims to produce \( e_2 \) \( \Rightarrow \) \( e_1 \) \( \in \Delta_2 \) if possible as shown in Figure 4 (a) and (b), respectively.

As such, a dynamic hidden race detector should aim to:

- **Phase I** keep a (sub)set of HB edges (i.e., \( \Delta_1 \)) in trace \( \sigma_1 \).
- **Phase II** and schedule a subsequent execution (generating trace \( \sigma_2 \)) to reverse HB edges in \( \Delta_1 \) to expose races hidden in trace \( \sigma_1 \).

However, \textit{DrFinder} does not target to keep any HB edges as which usually incurs high runtime overhead [12][21][41]. It tries to predict HB edges dynamically via the type of locks. In the next two subsections, we present how \textit{DrFinder} achieves this aim to reverse HB edges.

#### 4.2 Reversing Happens-before Edges

Suppose that \( e_1 \) \( \Rightarrow \) \( e_2 \) is an HB edge in \( \Delta_1 \) as depicted in Figure 4(a), there is a good chance that, using the default scheduler, \( e_1 \) \( \Rightarrow \) \( e_2 \) may also exist in \( \Delta_2 \). That is, most of HB edges in \( \Delta_1 \) cannot be easily reversed in \( \Delta_2 \). Therefore, our target is to actively produce \( e_2 \) \( \Rightarrow \) \( e_1 \) as depicted in Figure 4(b).

Suppose that during the execution (to generate the trace \( \sigma_2 \)), both events \( e_1 \) and \( e_2 \) exist. In theory, a precise but hypothetical strategy can be formulated as follows:

| To reverse an HB edge from \( e_1 \) \( \Rightarrow \) \( e_2 \) to \( e_2 \) \( \Rightarrow \) \( e_1 \), the thread \( t_1 \) should be suspended when it generates but does not execute the event \( e_1 \) until the thread \( t_2 \) has executed the event \( e_2 \). |

However, implementing such a strategy is challenging: if there is no such an event \( e_2 \) in trace \( \sigma_2 \), then the HB edge \( e_2 \) \( \Rightarrow \) \( e_1 \) will not exist, and no HB edge needs to be reversed. Hence, above strategy will suspend the thread \( t_1 \) until the thread \( t_2 \) terminates. If all events (or at least lock acquisition events) in trace \( \sigma_2 \) are logged to check the existence of an event \( e_2 \), it is necessary to compute an object abstraction [13][21][41] (e.g., a unique id) for each event.

However, before the occurrence of \( e_2 \), there is no way to compute an object abstraction for \( e_2 \).

Therefore, an effective technique must address a problem: Given an event \( e_1 \) to be executed by a thread \( t_1 \), how to determine whether some other thread \( t_2 \) will execute an event \( e_2 \) such that \( \text{object}(e_2) = \text{object}(e_1) \) without computing an object abstraction for each event?

#### 4.3 Stack and Type Based Events Predictions

Let us refine the problem further as the two events \( e_1 \) and \( e_2 \) should be causally related; otherwise, there is no need to consider them to form an HB edge. Suppose that when \( t_1 \) is about to execute \( e_1 \), thread \( t_2 \) is executing an event \( e^* \) within the body of a method \( f_k \) as shown in Figure 5(a). To ease our explanation, we refer to the current call stack of thread \( t_2 \) as stack \( x \). If the event \( e_2 \) will occur in future in the execution of \( t_2 \), there will be another call stack fragment: \( s_1 \) in the below Backward Case or \( s_2 \) in the below Forward Case:

- **Backward Case** shown in Figure 5(b): after thread \( t_2 \) returns from method \( f_k \) recursively to a method \( f_1 \), and then calls some other methods, an event \( e_2 \) from a method \( f_{k'} \) is executed. We refer to the two call stack fragments \( \langle f_1 \ldots f_k \rangle \) and \( \langle f_{k'} \ldots f_d \rangle \) as \( s' \) and \( s_1 \), respectively, as depicted.

- **Forward Case** shown in Figure 5(c): before thread \( t_2 \) returns from its execution in method \( f_k \), it further calls some methods and then an event \( e_2 \) from a method \( f_d'' \) is executed. We refer to the call stack fragments \( \langle f_k \ldots f_d'' \rangle \) as \( s_2 \), as depicted.

Therefore, to predict the existence of event \( e_2 \), thread \( t_2 \) should be aware of the method \( f_{k'} \) (in Backward Case) or the method \( f_d'' \) (in Forward Case) that executes an event \( e_2 \). To do so, it is necessary to record the events that a given method will execute directly or indirectly (i.e., via calling other methods). With such information, given an event \( e^* \) from a method \( f_k \), it becomes easy to know whether there will be an event \( e_2 \).

However, directly implementing above idea to detect hidden races is ineffective or even does not work. It is because each method, once called, directly (for event within this method) or indirectly (for events out of this method) executes all later events. For example, the method \textit{run()} executes all events. Therefore, to make the prediction of an event effective for detection of hidden races via reversing HB edges, the used stack should be limited. In other words, only some methods in a stack should be used to do prediction, but not all.

Let’s further review the two cases. For Backward Case, we can observe from Figure 5(b) that the effective call stack to predict the event \( e_2 \) (when thread \( t_2 \) is executing an event \( e^* \) in method \( f_k \)) is only the stack fragments \( s' \) and \( s_1 \). Similarly, for Forward Case, the effective stack is the stack fragment \( s_2 \).

Therefore, we only use the stack fragments \( s' \) and \( s_1 \) or \( s_2 \) to predict events. In theory, the size of the stack \( s' \) can range from 1 to infinite. In this paper, we aim to present the basic model of
DrFinder. Hence, we choose the size of $s'$ to be 1, which is a minimal setting. In this case, we have $f_k = f_1$. Therefore, the two cases (i.e., Backward Case and Forward Case) are actually the same one. And the prediction of event $e_2$ is based on one stack fragment (i.e., $s_2$ or $s_2$) to be formed. Thus, we propose our stack based events prediction model $M$ to predict the event $e_2$ as follows:

During the generation of a trace $σ_2$, if a thread $t_2$ is executing an event $e'$ from a method $f_k$, and there exists a method $f_d'$, such that the method $f_d'$:
1. contains an event $e_2$ and
2. is reachable by thread $t_2$ via a sequence methods $s_t$ after executing the event $e'$,
then thread $t_2$ will execute the event $e_2$.

We refer to the size of the above stack $s_t$ plus the method $f_d'$ as the depth $d$ of model $M$. Model $M$ looks forward to see whether there will be a sequence of at most $d$ methods (i.e., $s_t$) with the last method containing an above discussed event $e_2$.

However, for above prediction model, we still need to address a new problem: given an event $e_1$ by $t_2$ and a stack $s$ of $t_2$, does there exist a method $f_d'$ in the stack contains an event $e_2$?

We propose the May-Trigger Relation to further predict whether a given method will trigger a certain event. Suppose that there is an event $e$ from a method $f$ such that $e = e_2$. Then, we must have object($e$) = object($e_2$), and type(object($e$)) = type(object($e_2$)). Our insight is that in real-world Java programs, most methods only acquire specific (instead of arbitrary) lock objects, and their method instances often follow the same locking patterns. We propose to use the type of a lock object in a lock acquisition event to predict the possible occurrence of the event $e$: to achieve the reversal of the HB edges from $e_1 \Rightarrow e_2$ in $Δ_1$ to $e_2 \Rightarrow e_1$ in $Δ_2$.

Formally, May-Trigger Relation is defined as follows: Given a method $f$, a type $c$, and an execution trace $σ$. If a method $f'$ is reachable from $f$ during the generation of a trace $σ$ by a sequence of at most $d$ methods, and $f'$ produces an event $e = (t, acq, o)$ such that type$(o) = c$, then we say $f$ and $c$ forms a May-Trigger Relation (MTR for short), denoted as $mtf(c)$. DrFinder is developed on top of $M$ using MTR to predict occurrences of events like $e_2$ to schedule a program to detect hidden races. It uses two pieces of information for its prediction: (1) an event $e_1$ from a thread $t_2$, and (2) a method $f_d$ from a second thread $t_2$ and a depth $d$. It interestingly predicts the presence of an event $e_2$ by checking whether $mtf$ is a MTR identified from $σ_1$.

### 4.4 Effective Scheduling via Feedbacks

From above discussion, DrFinder executes a program once to collect a set of MTR and then schedules the program execution based on the relation set. However, races in a program cannot be dynamically detected in merely one run. Therefore, it is necessary for DrFinder to execute a program multiple times to detect more hidden races.

On the other hand, if a same set of MTR relation is used in each execution by DrFinder, the increment of new races detected is marginal. Actually, after the first scheduling execution by DrFinder, the probability to detect new hidden races for DrFinder at its other subsequent executions is the same as the existing dynamic techniques (e.g., FastTrack) and may be even lower than existing active schedulers (e.g., PCT). It is because, the subsequent executions are similar to the first scheduled execution as they are scheduled by DrFinder based on a same set of MTR.

Therefore, we design DrFinder, at each of its executions, to both schedule the execution and collect a new set of MTR from the execution being scheduled. The newly collected set of MTR is regarded as a feedback to be used in the next execution by DrFinder. As such, DrFinder is able to effectively schedule each execution based on a set of MTR exactly from the previous execution, to detect new hidden races. This feedback mechanism is also depicted in Figure 1.

### 5. DRFinder in Details

#### 5.1 Phase I: May-Trigger Relation Collector

The MTCollector algorithm is responsible to collect may-trigger relations, shown in Algorithm 1. Given a program $p$ and a depth $d$, MTCollector executes the program, and collects a set of may-trigger relations (i.e., MTR in Algorithm 1) from the observed trace.

Algorithm 1 first assigns the set of all threads in $p$ to the set Enabled, null to MTR, and assigns Stack$(t)$ for each thread $t$ in Enabled to empty at lines 3–4. It uses the data structure Stack$(t)$ to keep track of the call stack fragment. It then uses randomized scheduling to execute the program $p$ by selecting the next event $e$.
Algorithm 2: DrFinder.Agent

1. Input: p – a given program.
2. Input: MTR – may-trigger relations returned by MTCollector
3. Input: RD – an HB race detector (e.g., FastTrack)
4. Output: MTR’ – a set of new may-trigger relations /feedbacks
5. Enabled = all threads in p, MTR’ = ∅
6. fork(scheduler()) /start scheduler (in Algorithm 3)
7. while Enabled ≠ ∅ do
8. let t be a random thread from Enabled.
9. let e = (t, op, o) be the next event of t.
10. if op = acq then
11. allow := DrFinder.Scheduler.requestALock(t, o)
12. if allow = false then
13. continue /while loop
14. end if
15. end if
16. execute(e)
17. RD.onEvent(e) //for detection of data races
18. if op = acq then
19. DrFinder.Scheduler.lockAcquired(t, o)
20. end if
21. update MTR’ according to Algorithm 1
22. end while

(i.e., (t, op, o)) from a random thread (lines 6–7) and checks the operation op of the event e:

- If op is either enter or return, Algorithm 1 pushes the method o into the Stack(t) or pops out the topmost method from the Stack(t), respectively (lines 8–11).
- If op is acq, Algorithm 1 updates MTR := MTR ∪ {m(f, type(o))}, for each method f in the top d methods in the Stack(t) (lines 13–16) to maintain may-trigger relations.

Then, the algorithm executes the event at line 18.

5.2 Part A of Phase II: DrFinder Agent

The Agent (Algorithm 2) is responsible to execute each event. For the acq events, their executions depend on Scheduler.

Algorithm 2 accepts the given program p, the set MTR (i.e., may-trigger relations) from Phase I, and an HB race detector RD. It returns a set of new may-trigger relations (i.e., MTR’). Algorithm 2 firstly assigns the set of all threads of p to the set Enabled, which is shared by both Agent and Scheduler, and set MTR’ to be empty (line 5). It then starts Scheduler (i.e., Algorithm 3 to be presented in Section 5.3) through a fork() call (line 6). Next, it takes a next event e = (t, op, o) from a random thread t (lines 8–9), and checks the operation op of the event e. If op is acq, Agent asks Scheduler whether t is allowed to execute e through a function call to requestALock() of Scheduler (line 11). If the function returns a false value, Agent simply keeps e from execution. (Note: the thread t has been removed from Enabled by Scheduler at line 10 in Algorithm 3). If requestALock() does not return a false value (line 12), Agent will both execute and pass e to the race detector RD (lines 16–17). Finally, if e is an acq event, Agent also informs Scheduler that the thread t has acquired the lock object specified in e via function lockAcquired() (lines 18–20). It also collects a new set of MTR’ at line 21 (which is based on Algorithm 1) as the input MTR of the next scheduling execution.

5.3 Part B of Phase II: DrFinder Scheduler

Scheduler (Algorithm 3) maintains four data structures: ATHs, RTHs, allowedTH, and allowedLK (lines 1–4), to make scheduling decision:

- ATHs is a set of all the threads in the program p.
- RTHs is a set of pairs of a thread t and a lock object o, each of which representing that t is requesting to acquire the object o, but Scheduler suspends this acquisition. Thus, all the threads in this set are waiting to be scheduled by Scheduler. For ease of our presentation, we use RTHs.get(t) to denote the lock o paired with the thread t.
  - allowedTH keeps a particular thread t that both (i) is "escorted" by DrFinder with a top priority to execute its lock acquisition events and (ii) is the thread expected by DrFinder to acquire a lock object defined by allowedLK.
  - allowedLK keeps a lock object o. DrFinder expects the thread defined by allowedTH to acquire a lock object having the same type as this lock object.

Scheduler consists of four functions: requestALock(), lockAcquired(), mayTrigger(), and schedule(). In Section 5.2, we have presented that Agent (Algorithm 2) invokes the first two functions. We firstly present them followed by presenting schedule() which is the core part of DrFinder.

The function requestALock() is called by Agent on determining whether to execute the event e (i.e., the lock acquisition on o by t). It checks whether the given thread t is a chosen thread to execute any event (i.e., allowedTH) at line 6. (As such, a targeted HB edge may be formed as soon as it can.) If a true value is returned, it indicates that the event e is allowed to execute (line 7); otherwise, the thread t is added to the set RTHs and is also removed from the set Enabled (lines 9–10) so that Agent will not pick any event of it for execution (line 8 in Algorithm 2). Next, the function notifies Scheduler (that there is a thread to be scheduled, see line 30, to be explained below). On the other hand, if it returns a false value at line 12, it indicates that the event e is not allowed to be executed.

The function lockAcquired() is called by Agent right after a lock acquisition event e is executed (lines 17–19 in Algorithm 2). It checks whether the executed lock acquisition event e is an event expected by Scheduler (line 16). An event e = (t, op, o) is an expected event if (1) t is the thread defined by allowedTH (i.e., t = allowedTH) and (2) e operates on a lock object defined by allowedLK (i.e., type(o) = type(allowedLK)) at line 16. If so, an expected event occurs and Scheduler resets both allowedTH and allowedLK to null (line 17). It then notifies function schedule() that there is no thread marked as "allowedTH" (line 18).

We are going to explain the function schedule(). Scheduler is started by Agent (at line 5 in Algorithm 2). If no thread is requesting any lock object (i.e., RTHs.size() = 0) or there is no event expected by Scheduler (i.e., allowedTH ≠ ∅) at line 30, the function schedule() just waits for notifyScheduler() to notify it at either line 11 or line 18.

Once schedule() is notified, the same function selects a random thread t from the set ATHs. It then checks whether there is any thread whose currently executing method f and the type of the given object o match any may-trigger relation kept in MTR via mayTrigger() (lines 35–39). All the threads that match this condition are collected as the set CT (line 37), in which a thread t is randomly selected and escorted by DrFinder to execute all its events before t is allowed to continue its execution (line 41). The function then assigns the thread t and the lock object o to allowedTH and allowedLK, respectively (lines 42–43). If the thread t is also in the set RTHs, t’ will be removed from RTHs and added to Enabled (lines 44–46). Scheduler then waits until the thread t’ being escorted by DrFinder has acquired a lock object having the type
Algorithm 3: DrFinder.Scheduler

1. AThs := all threads in p
2. RThs := ∅ /a set of pairs of threads and locks
3. allowedTH := ∅ /a thread that is expected to acquire a lock
4. allowedLK := ∅ /the lock expected to be acquired
5. Function requestLock(t, o)
   6. if allowedTH=t then
      7. return true
   8. else
      9. RThs.add(t, o)
     10. Enabled := Enabled ∪ {t}
     11. notifyScheduler()
     12. return false
   13. end if
   14. end Function
7. Function lockAcquired(t, o)
   8. if allowedTH=t and type(allowedLK) = type(o) then
      9. allowedTH := ∅, allowedLK := ∅
   10. end if
   11. end Function
9. Function mayTrigger(t’)
   12. let o = RThs.get(t), f := getCurrentMethod(t’)
   13. if ∃ not(f, c) ∈ MTR such that c = type(o) then
      14. return true
   15. end if
   16. end Function
17. Function schedule()
   18. while Agent does not exit do
      19. while RThs.size() = 0 or allowedTH ≠ ∅ do
         20. wait()
      21. end while
   22. t := a random thread from AThs
   23. CandidateThread CT := ∅
   24. for each thread t’ ∈ AThs do
      25. if t ≠ t’ and mayTrigger(t, t’) then
         26. CT := CT ∪ {t’}
      27. end if
   28. end for
   29. if CT ≠ ∅ then
      30. t’ := a random thread from CT
   31. allowedTH := t’,
   32. allowedLK := RThs.get(t’)
   33. if t ∈ RThs then
      34. Enabled := Enabled ∪ {t’}
   35. RThs.remove(t’)
   36. end if
   37. if while allowedTH ≠ ∅ do
      38. wait()
   39. end while
   40. if Enabled := Enabled ∪ {t}
   41. RThs.remove(t)
   42. allowedTH := t [let t to acquire the lock RThs.get(t)
   43. allowedLK := RThs.get(t)
   44. end while
   45. end Function

A thread t is allowed to execute (lines 52–55).

The function mayTrigger() accepts two threads t and t’ as its parameters. This function firstly gets the lock object o being requested by t (i.e., paired with t in RThs) and the current executing method f of t’ (line 22). It then checks whether the tuple (f, type(o)) is a valid May-trigger relation (i.e., in the set MTR) and returns a true-or-false result accordingly (lines 23–26).

5.4 Discussion

DrFinder is an active scheduler. It suffers from thrashing [21][41], and may lead the execution to form deadlocks [12][41]. Similar to existing techniques [9][11][12][21][41], when a thrashing occurs, DrFinder randomly selects a suspended thread to execute and when a deadlock occurs, the whole execution is restarted (see Section 6.1).

DrFinder actively schedules an execution to produce HB edges that cannot be easily formed in normal executions to expose races. It drives a happens-before based detector (e.g., FastTrack) to detect races precisely and report all races. Therefore, other HB detectors, DrFinder is also precise.

6. EXPERIMENT

This section presents our evaluation on DrFinder and its comparison with the state-of-the-art HB race detector FastTrack, a random delay scheduler ConTest, and a state-of-the-art randomized scheduler PCT. All these techniques are reviewed in Sections 1 and 3.

6.1 Implementation and Benchmarks

Implementation. We implemented DrFinder, FastTrack, ConTest, and PCT in Jikes JVM [2][3]. Jikes JVM is a Java virtual machine, developed almost in Java language, and could be run on Linux and Mac OS X systems. These tools report a race at the Java class field level [17]. Our tool uses the shadow mechanism [18] to track the state of an execution and adds a shadow lock to each object instance to keep the vector clock data and type information. For each memory location (i.e., an instance of a field of a Java class), it allocates a shadow memory to track the reads and writes to this memory location. For each thread, it adds a member in the JVMThread class [2] to keep the Java thread data.

To generate memory and synchronization events in runtime, our tool instruments each class when it is loaded, except those Jikes JVM classes and Java standard library classes. It uses a static escape analysis [5] to identify accesses to provably thread local memory location. It also fully tracks happens-before relations on other program semantics (e.g., accesses to volatile fields [17]).

Our implementation periodically monitors the states of all threads by tracking various synchronizations events and scheduling of DrFinder as well as other functions calls (e.g., sleep()). Such monitoring is helpful to identify deadlock occurrences and thrashing occurrences.

Benchmarks. We used the Dacapo benchmark suite [6] to evaluate DrFinder. We selected two multithreaded programs from Dacapo 2006-10-M1 (xalan and eclipse) and five multithreaded programs from Dacapo 2009 (xalan, pmd, sunflow, luindex, and lusearch). Dacapo 2009 includes other multithreaded benchmarks; however, they cannot be run on the latest Jikes JVM 3.1.3 even without our tool. In total, we selected 7 multithreaded benchmarks, including a large-scale real-word program Eclipse (eclipse).

Table 2 shows the descriptive statistics of the benchmarks. The first two columns show the benchmark name and size. The third column
shows the numbers of locks and threads in the benchmarks. The last two columns show the numbers of methods (all and those containing lock acquisitions) and the mean number of HB edges in each benchmark over 10 runs. All dynamic data are collected under native scheduling.

### 6.2 Experimental Setup

Our experiment was performed on an Apple Mac Pro with 2.6GHz Intel Core i5 and 8GB memory running OS X 10.9.2. We compiled Jikes RVM with GNU Class-path 0.98 [1]. We configured FastTrack with the native (OS) scheduler, with ConTest, with PCT, and with DrFinder, which are referred to as FT, CT, PCT, and DR, respectively. We followed the previous experiments [43] and ran each technique on each subject for 10 times.

### 6.3 Experimental Results

Table 3 summarizes the experimental results. The first column shows the benchmark name. The second major column shows the number of distinct data races reported by each technique in 10 runs. The third major column shows the number of new data races detected by DR but not detected by FT, CT, and PCT. The fourth major column shows that the total number of distinct races detected by all four techniques. The fifth major column shows the number of races not detected by each technique compared to the total number of distinct races (i.e., the data in the fourth major column). The sixth major column shows the mean time in seconds for each technique to run each benchmark. It also shows the time of native run (i.e., without any testing tool) and the time needed by both phases of DR (as P-I and P-II, respectively). The overhead of each technique is also calculated. The last column shows the thrashing rate (“Thrash. rate”) of DR. The last row shows either the total (“Sum”) or the mean value (“Avg”) of each column.

### 6.3.1 Summary of Results

**Effectiveness.** From Table 3 (the second and third major columns), DR detects more races in 5 out of 7 benchmarks by 11.1% to 240.0%. On the remaining two benchmarks, all four techniques detect the same set of data races. In total, DR detects 89 more new races from all benchmarks. We find that FT, CT, and PCT detect almost the same set of data races (where the difference is at most 10). This is consistent with an intuition that the random sleep strategy used by CT is not quite effective and randomized scheduling strategy used by PCT is also not quite effective without a larger number of runs.

**Performance.** From the column on time, we observe that FT has the best performance, which is expected. The overhead of CT on top of FT ranges from 0.11x to 5.28x except on eclipse. For PCT, it incurs about 2.2x higher overhead than DR on average.\(^1\)

The overhead of the Phase I of DR is only 0.39x. The overhead of DR (Phase II) on top of FT ranges from 1.1x to 4.0x. On average, DR only incurs 2.65x on top of FT.

On eclipse, CT incurs a heavy overhead, which is 19.5x on top of FT; but, DR only incurs 1.1x on top of FT. Although eclipse included 26 threads, in most of the execution time, there are only two active threads. We find that DR is able to suspend these two threads according to their locking orders most of the time. But, CT has to delay each lock acquisition by a random period. As a result, the total time overhead of CT is much heavier than that of DR on this benchmark.

**Thrashing Rate.** From the last column of Table 3, DR is able to make successful thread suspension decisions in nearly 60% of all cases. In the remaining cases (40.7%), thrashing occurred. We have inspected these thrashing occurrences and found that about half of them were caused due to the type of the unique and global instance of Class Loader class used by the Dacapo test harness to load each class instead of the program under test. (This harness strictly speaking is not a part of each benchmark.) In our experiment, we have not seen any deadlock occurrence.

### 6.3.2 Comparison on Not Detected Races

Table 3 also shows the total number of distinct races detected by all four techniques in the fourth major column. It also shows the number of races that are not detected by each technique but detected by other three techniques in the fifth major column. Overall speaking, among all 474 detected races detected by all four techniques on all the benchmarks, DR only misses 26 races; however, FT, CT, and PCT misses 96, 102, and 89 races, respectively.

From above analyses, we find that DR is effective in exposing hidden races; but it may be unable to expose some races that can be detected by HB detectors with randomized or native scheduling. We argue that this is not a major issue. It is because, in practice, one may run a program with random or native scheduling to detect these races (e.g., configured in Phase I of DR) followed by detecting the hidden races in Phase II of DR. We have checked the races not detected by DR and found that almost all 26 races have been detected by FT in each of 10 runs, and the remaining ones can be detected by FT in at least one run.

### 6.3.3 Comparison on Races Detected in 10 Runs

Figure 6 shows the cumulative number of races detected in the experiment by the four techniques on each benchmark except on luindex (on which, all three techniques detected exactly one race in each run). The x-axis shows these 10 runs and y-axis shows the cumulative number of races detected.

\(^1\) Note that, there is a parallel version of PCT known as PP/PCT [33] that has the same effectiveness but runs faster than PCT. Hence, the time overhead of PCT in Table 3 is for reference only and we do not discuss the overhead of PCT in the next paragraph. We believe our DrFinder could also be implemented in parallel and we leave it as a future work.
Figure 6 shows that FT, CT, and PCT almost always detect the same numbers of races except on eclipse06. This indicates that they have similar race detection ability among all 10 runs (where the detected races are almost the same ones). Whereas, DR has an increasing trend on the number of detected races. Apparently, DR may detect fewer races in some runs (e.g., the first five runs on xalan06). But, we have shown in Table 3 that DR actually exposes more races.

To measure the ability of DR on the detection of new races with increasing number of runs, we further analyze the cumulative number of 89 new data races detected by DR in the first "i" (where i is from 1 to 10) runs. We normalize this cumulative number by the total number of new races detected by DR on the corresponding benchmark. The result is shown in Figure 7. Note that we do not show the result on luindex09 and lusearch09 in Figure 7 as no new races are detected on them.

Figure 7 shows that on each benchmark except eclipse06, all the new races detected by DR were detected in the first 7 runs. On eclipse06, almost on each run, more new races were detected by DR; and moreover, more than 71.43% new races were detected in the first 4 runs. Therefore, we tend to believe that DR is able to effectively detect hidden races, even on large-scale multithreaded programs (e.g., eclipse06), which cannot be detected by FT, CT, and PCT in 10 runs (or even up to 100 runs, see Section 6.3.5).

6.3.4 DrFinder with Different Depths
In our main experiment, we have set the depth to 12. To evaluate the ability of DR on its detection of hidden races with different depths, we repeated the main experiment for DR but set the depth from 2 to 20 with step 2 in turn, where each configuration was also conducted for 10 runs. The results are summarized in Table 4. In each data cell, the format is "s (y)" where the "s" is the total number of races detected by DR with corresponding depth and the "y" is the number of new races that cannot be detected by FT, CT, and PCT in their 10 runs. On each benchmark, if DR is able to detect most new races among all its depths, we mark the corresponding cell with gray color. Similarly, we do not mark cells corresponding to luindex09 and lusearch09 (as the data in the either entire row shows that there is no new race detected).

From Table 4, we observe that with different depths, DR is generally able to detect more races than that detected by FT, CT, and PCT. DR also detects new races in almost all these depths, where the exception is on sunflow09.

Another observation from Table 4 is that when depth values are within 4 and 12, DR is likely to detect a significant amount of new races on top of FT, CT, and PCT. And these depths values also lead to a larger amounts of total races. We also highlight these cells in green color in the last row of Table 4.

In future, we will study both how the depth values affect the ability of DR and non-parametric strategies to determine the depth.

6.3.5 Further Evaluation on New Races
In the 10 runs by all techniques, DR detects 89 new races. We further repeatedly ran other three techniques more times until either

Table 4. The total number of data races detected by DR with depths from 2 to 20 (with Step 2) in 10 runs.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
<th>10</th>
<th>12</th>
<th>14</th>
<th>16</th>
<th>18</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>xalan06</td>
<td>28 (16)</td>
<td>31 (15)</td>
<td>30 (15)</td>
<td>27 (15)</td>
<td>27 (15)</td>
<td>26 (13)</td>
<td>31 (15)</td>
<td>26 (15)</td>
<td>27 (15)</td>
<td>27 (15)</td>
</tr>
<tr>
<td>eclipse06</td>
<td>306 (13)</td>
<td>337 (43)</td>
<td>336 (42)</td>
<td>343 (46)</td>
<td>336 (41)</td>
<td>351 (54)</td>
<td>316 (21)</td>
<td>301 (18)</td>
<td>284 (15)</td>
<td>312 (15)</td>
</tr>
<tr>
<td>xalan09</td>
<td>18 (7)</td>
<td>18 (7)</td>
<td>18 (7)</td>
<td>18 (6)</td>
<td>18 (7)</td>
<td>20 (8)</td>
<td>17 (6)</td>
<td>12 (6)</td>
<td>11 (5)</td>
<td>13 (1)</td>
</tr>
<tr>
<td>pm09</td>
<td>20 (2)</td>
<td>18 (1)</td>
<td>19 (2)</td>
<td>20 (2)</td>
<td>20 (2)</td>
<td>20 (2)</td>
<td>20 (2)</td>
<td>20 (2)</td>
<td>20 (2)</td>
<td>20 (2)</td>
</tr>
<tr>
<td>sunflow09</td>
<td>5 (0)</td>
<td>5 (0)</td>
<td>6 (0)</td>
<td>5 (0)</td>
<td>14 (9)</td>
<td>17 (12)</td>
<td>6 (1)</td>
<td>6 (0)</td>
<td>5 (0)</td>
<td>5 (0)</td>
</tr>
<tr>
<td>luindex09</td>
<td>13 (0)</td>
<td>13 (0)</td>
<td>13 (0)</td>
<td>13 (0)</td>
<td>13 (0)</td>
<td>13 (0)</td>
<td>13 (0)</td>
<td>13 (0)</td>
<td>13 (0)</td>
<td>13 (0)</td>
</tr>
</tbody>
</table>

Sum: 391 (38) 423 (66) 423 (66) 427 (69) 428 (74) 448 (89) 404 (45) 379 (41) 361 (37) 391 (33)
We found that on eclipse06, xalan04, xalan09, and pmd05, other three techniques could not detect as many races even exhausting all 100 runs as what DR detects in the 10 runs. On sunflow06, all three techniques need more than 25 runs to catch up with DR.

Because eclipse06 is the largest one in our benchmarks, we further analyzed the ability of FT, CT, and PCT on detecting the 56 new races from eclipse06 that are only detected by DR but missed by all the other three in their 10 runs in the main experiment. The result is shown in Figure 8, where we also list the 56 races for comparison. It shows that FT, CT, and PCT were able to detect only 35, 13, and 19 races out of the 56 races in all 100 runs. (Note that, in a run, FT detects 17 races. Excluding these 17 races, it detects less races than that by PCT in all 100 runs.) This experiment, once more, illustrates that DR is effective on detecting hidden races.

7. RELATED WORK

Many techniques on data race detection have been proposed. They mainly fall into two groups: static techniques [22][32][36][46] and dynamic techniques [17][35][40][43][49]. Static techniques like RELAY [46] and LockSmith [36] rely on statically but imprecisely identifying memory-accessing statements that may concurrently visit same memory locations without the same locksets. Chord [32] reduces the number of false warnings by using several stages of refinement on the entire data race warning set. But, it loses the soundness guarantee of reporting all data races in a program.

Many dynamic detectors use the locking discipline [40][42] to predict races. However, this discipline is not necessarily to be obeyed even for data-race-free programs so that many false positives may be generated using such a strategy. HB based dynamic ones [17][35] can precisely report data races. However, they are sensitive to particular thread interleaving (even with improvement [45][47]) which provides less coverage than those using the lockset strategy.

RaceMob [23] statically detects data race warnings and distributes them to a large number of user processes to validate real races. However RaceMob only works on limited scenarios where a distributed user site computation is available. Active testing techniques [41][37] need runs for confirmation after an imprecise race detection phase. In such a run, the schedules are guided by the set of data race warnings to trigger real data races. This kind of approach is able to confirm real races but cannot eliminate false positives.

DrFinder takes another approach by using a precise data race detector, i.e., FastTrack [17], in the first place. With the inherent limitation of the sensitivity on thread interleaving, an effective thread scheduling technique such as DrFinder is a desirable complement with HB detectors (if used in our Phase I) to provide precise data race reports with high coverage.

Thread Scheduling techniques are more promising to detect races than pure stress testing. Systematic scheduling techniques such as model checking [48][30], are in theory able to exhaustively execute every schedule. However, due to the state explosion problem, enumerating each schedule is not practical for real-world programs. Chess [30] sets a heuristic bound on the number of pre-emptions to explore the schedules. Also, although systematic approaches avoid executing previously explored schedules and are more scalable than pure model checking techniques [14], they usually incur large overheads and fail to scale up to handle long running programs. Although improvement for Chess exists [4], finding the positions for such bounded exhaustive exploration from a large trace to effectively expose hidden races is challenging [24].

Another type of scheduling technique is based on some coverage criteria of concurrent programs [7][19]. For example, Maple [50] relies on patterns (i.e., lRoots [50]) to mine certain coverage to expose concurrency bugs. However, Maple is insensitive to detect races requiring reversing more than one HB edge. Existing experiments (e.g., [10]) have shown that on a large benchmark like Chromium, there may be 16 million or more HB edges in a trace. It is challenging to select an effective subset of all such HB edges to confirm given patterns as Maple is designed to confirm one pattern per confirmation run. Besides, the relation between the coverage of a specific metric and targeted concurrency bugs cannot be verified in theory. A previous empirical study [26] has shown that different criteria have different effectiveness on different testing techniques. This increases the difficulty of choosing a suitable criterion.

DrFinder is specially designed to detect hidden races based on our may-trigger relation. Unlike above reviewed systematic scheduling techniques or coverage based techniques that have to restrict their scheduling bounds [30][50], DrFinder is able to scale up to large-scale programs (i.e., Eclipse in our benchmark) and does not require any bug patterns.

8. CONCLUSION

This paper presents a dynamic technique DrFinder to detect hidden races in multithreaded Java programs. It tries to reverse possible HB edges based on a type based May-trigger Relation. The experiment shows that DrFinder is promising in detecting hidden races and detected 89 races that were missed by existing techniques FastTrack, ConTest, and PCT. Many new races detected by DrFinder in 10 runs cannot be detected by other techniques even in 100 runs. DrFinder is also efficient as it incurs less overhead than other active scheduling techniques CT and PCT. In future, we will extend our basic model of DrFinder proposed in this paper to further validate its ability on detection of hidden data races.

9. ACKNOWLEDGEMENTS

We thank anonymous reviewers for their invaluable comments and suggestions on improving this work. We also thank Dr. W.K. Chan and Mr. Chunbai Yang at City University of Hong Kong for their help on the preliminary version of this work. This work is partially supported by the National Natural Science Foundation of China (NSFC) under grant No. 91418206.

Figure 8. Cumulative effectiveness of FT, CT, and PCT in 100 runs on the 56 races that can only be detected by DR in 10 runs.
10. REFERENCES


