Race Directed Scheduling of Concurrent Programs

Mahdi Eslamimehr
UCLA, University of California, Los Angeles
mahdi@cs.ucla.edu

Jens Palsberg
UCLA, University of California, Los Angeles
palsberg@ucla.edu

Abstract

Detection of data races in Java programs remains a difficult problem. The best static techniques produce many false positives, and also the best dynamic techniques leave room for improvement. We present a new technique called race directed scheduling that for a given race candidate searches for an input and a schedule that lead to the race. The search iterates a combination of concolic execution and schedule improvement, and turns out to find useful inputs and schedules efficiently. We use an existing technique to produce a manageable number of race candidates. Our experiments on 23 Java programs found 72 real races that were missed by the best existing dynamic techniques. Among those 72 races, 31 races were found with schedules that have between 1 million and 108 million events, which suggests that they are rare and hard-to-find races.

Categories and Subject Descriptors D.2.5 Software Engineering [*Testing and Debugging*]

Keywords concurrency; race detection

1. Introduction

Concurrent programming with shared memory offers both the benefit of efficient execution and the pitfall of data races. Efficiency can be achieved when we let multiple processors run in parallel and exchange data via the shared memory. A data race arises when two processes simultaneously access a shared memory location and at least one of the two accesses is a write operation. Data races often result in hard-to-detects bugs and usually the programmers of concurrent software should try to avoid data races.

One reason for why data races are problematic can be found in a seminal paper by Adve, Hill, Miller, and Netzer [10]. Their observation is that on suitable hardware, every execution of a data-race-free program is *sequentially consistent*. Sequential consistency was introduced by Lamport in 1979 and means that "the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program" [31]. Sequential consistency provides a useful memory model that simplifies the task of producing correct concurrent programs. If programmers can avoid data races, they can use sequential consistency as their memory model.

Researchers have developed many techniques to help programmers detect data races. Some of those techniques require program annotations that typically must be supplied by a programmer; examples include [9, 15]. Other techniques work with unannotated programs and thus they are easier to use. In this paper we focus on techniques that work with unannotated Java programs. We use 23 open-source benchmarks that have a total of more than 4.5 million lines of code, which we use "straight of the box" without annotations.

We can divide race-detection techniques into three categories: static, dynamic, and hybrid. A static technique examines the text of a program without running it; a dynamic technique runs a program, possibly multiple times, and gathers information during those executions; and a hybrid technique does both.

The advantage of a static technique is that if it is sound, then it will report every possible race, though it may also report false positives. We will show via experiments that the best existing static technique reports a large number of false positives that would be daunting to examine by hand. For our benchmarks, the Chord tool [34] reports a total 127,136 data races. So, current static techniques are of little use to working programmers. Potentially, a sound static technique can be valuable, particularly because if it reports zero races for a benchmark then indeed that benchmark has no races.

The advantage of a dynamic technique is that it reports only real races. For example, for our benchmarks, the FastTrack, Goldilocks, RaceFuzzer, and Pacer tools together report a total 304 data races. So, current dynamic techniques give programmers valuable help, yet our experiments show that they leave many races to be discovered.

The advantage of a hybrid technique is that it may be able to combine the best of both worlds, static and dynamic. The

PPoPP '14, February 15–19, 2014, Orlando, Florida, USA.

http://dx.doi.org/10.1145/2555243.2555263

best existing hybrid technique appears to be a technique by O'Callahan and Choi [35] that we call Hybrid, which for our benchmarks report a total 405 data races. This technique may produce both false positives and false negatives, yet the tool provides programmers with output of a fairly manageable size.

In this paper we focus on dynamic techniques. We will present a dynamic technique that reports significantly more real races than the previous techniques.

The main shortcoming of the existing dynamic techniques is that when they search for an execution that leads to a real race, they often come up empty handed. We present a novel approach to execution search that gives much better results. The central concept in our approach is the standard notion of *schedule*, which is a sequence of events that must be executed in order.

The challenge. Find an execution that leads to a real race.

Our result. We present *race directed scheduling* that for given a race candidate searches for an input and a schedule that lead to the race. The search iterates a combination of concolic execution and schedule improvement.

We have implemented race directed scheduling in a tool called Racageddon that does race detection for Java programs. We use an existing hybrid technique to produce a manageable number of race candidates.

For our benchmarks, our tool found 72 real races that were missed by the best existing dynamic techniques. Among the 304 real races found by the existing dynamic techniques, our technique found 272 of them. Our tool is fully automatic and its user needs no expertise on data races. Once our tool reports a race, our tool can replay the execution that leads to the race.

In summary, the two main contributions of this paper are:

- an effective and useful dynamic race detector and
- an experimental comparison of seven race detectors.

The rest of the paper. In the following section we discuss two techniques from previous work that we use as "black-box" components of Racageddon. In Section 3 we present our new approach, in Section 4 we present our experimental results, and in Section 5 we discuss related work.

2. Two Techniques from Previous Work

Racageddon uses two techniques from previous work [35, 41]. In both cases, Racageddon uses those techniques as "black boxes", that is, as unmodified components for which we rely only on their input-output behavior. We implemented both techniques ourselves after a careful study of the seminal papers [35, 41].

Generation of race candidates. We use a hybrid race detector by O'Callahan and Choi [35] that we call Hybrid. Hybrid combines lockset-based detection and happens-before-based detection into a single efficient technique that can produce both false positives and false negatives. We view the output of Hybrid as *race candidates* that deserve further attention. Hybrid provides a rather small number of race candidates, namely a total of 405 for our benchmarks of more than 4.5 million lines of code. Those 405 race candidates are an excellent starting point for our search for real races.

Schedule improvement. We use an approach to schedule improvement by Said, Wang, Yang, and Sakallah [41]. Their method maps a schedule to a permutation of the schedule. The idea is that a user supplies both a schedule that represents a trace of a program execution and also a race candidate, and then in return gets a schedule that has a better chance to lead to the race. The method has "memory": it takes advantage of the schedules that have been submitted in all previous calls. Together, all those schedules provide a wealth of information about happens-before relationships in a specific program. The method uses an SMT-solver and is highly efficient, even for the schedules of lengths beyond length 10^8 that we encountered in our experiments.

3. Race Directed Scheduling

We now present our approach to data race detection. We will use pseudo-code to describe both our approach and the data types that we use.

3.1 Data Types

Here are six data types that we use in Racageddon.

$$\begin{array}{rcl} \mathsf{Program} & = & \text{a Java 6 program} \\ \mathsf{Input} & = & \text{input to a Java 6 program} \\ \mathsf{Event} & = & \mathsf{threadId} \times \mathsf{statementLabel} \\ \mathsf{EventPair} & = & \mathsf{Event} \times \mathsf{Event} \\ \mathsf{Schedule} & = & \text{Event sequence} \\ \mathsf{Race} & = & \mathsf{EventPair} \times \mathsf{Input} \times \mathsf{Schedule} \end{array}$$

Racageddon works for Java 6 programs, which have the type Program. The input to such programs is a vector of values; we use Input to denote the type of input vectors.

When a program execution executes a particular statement in a particular thread, we refer to that as an *event* that has type Event. In the context of race detection, the key data type is EventPair that we use to describe two events that may form a race.

The standard notion of *schedule* is here the data type Schedule, which is a sequence of events.

A Race is the type of information that we need to replay an execution that leads to a race. A Race has three components, namely the EventPair that is the race, the Input that we should supply at the beginning of the execution, and the Schedule that the execution should follow to reach the race.

3.2 Two Tools

Let us describe the interfaces to the two off-the-shelf tools from Section 2 in terms of the data types listed above.

$$\begin{array}{l} \mathsf{hybrid} : \mathsf{Program} \to (\mathsf{EventPair\ set}) \\ \mathsf{improve} : (\mathsf{Schedule} \times \mathsf{EventPair}) \to (\mathsf{Schedule} \oplus \{\mathsf{none}\}) \end{array}$$

Here hybrid stands for O'Callahan and Choi's technique, while improve stands for Said, Wang, Yang, and Sakallah's technique. Notice that hybrid maps a Java program to a set of event pairs, that is, a set of race candidates. Notice also that improve maps a schedule to a better schedule or else to none if no better schedule was found. We leave implicit that improve has "memory" and takes advantage of the schedules that have been submitted in all previous calls. Notice finally that improve is idempotent in the sense that if $\mathsf{improve}(trace, c) \neq \mathsf{none}$, then

$$\mathsf{improve}(\mathsf{improve}(trace, c), c) = \mathsf{improve}(trace, c)$$

3.3 Concolic Execution

Racageddon uses concolic execution as one of its components. We will summarize the idea of concolic execution and we will introduce a slight generalization of the approach that we use in Racageddon.

Concolic execution [17, 18, 27, 32, 45–48], executes code with concrete and symbolic values simultaneously and uses the result to generate inputs for another execution. The term "concolic" combines the words "concrete" and "symbolic". Each execution collects constraints from the symbolic values and the conditions in the control-flow. Those constraints represent the executed control-flow path and they have *the concrete input to the run* as solution.

Suppose we want to execute a particular event, that is, a particular statement in a particular thread. We can execute a sequence of concolic runs that successively get closer and closer to execute the desired event. The idea is to do a minor modification of the constraints collected from conditions of branches. Imagine that a prefix of the concolic run made progress towards the desired event but at a particular branch B went off in a direction that appears to lead away from the desired event. We take the constraints from the prefix plus the *negation* of B. The solution to those constraints is an input that will steer the next concolic execution a little closer to the desired event by going off in the other direction at branch B.

Experience shows that concolic execution achieves better branch coverage with fewer test cases than testing with random inputs. In the first round of concolic execution, the input is chosen randomly.

We can generalize the standard approach to pursue execution of an entire schedule, that is, an event sequence. For example, suppose we want execution of the schedule (e_1, e_2, e_3). Some rounds of concolic execution may lead to execution of e_1. We can refer to those rounds together as a super-round. Now we can use the constraints that lead to execution of e_1 and continue with a second super-round that leads to execution of first e_1 and later e_2. Finally, we can do a third super-round and achieve execution of the entire schedule.

The above method generalizes easily to schedules of any length.

If we manage to execute an entire given schedule, we continue to explore additional schedules that have the given schedule as prefix.

We describe our interface to concolic execution in the following way.

$$\begin{aligned}\mathsf{concolic}: \quad & (\mathsf{Program} \times \mathsf{Schedule}) \to \\ & ((\mathsf{Race\ set}) \times \mathsf{Schedule})\end{aligned}$$

The input to concolic is a program and a schedule, and concolic will execute one super-round per element in the schedule. A run of concolic has two outputs. The first output is a set of all races that were found by any of the individual concolic executions. The second output is a schedule that represents the trace of final concolic execution, irrespectively of whether the given schedule was executed. We emphasize that each call to concolic may do many concolic executions, hence have many opportunities to collect races.

3.4 Helper functions

We use three helper functions:

$$\begin{aligned}\mathsf{present}: \quad & (\mathsf{EventPair} \times \mathsf{Schedule}) \to \mathsf{boolean} \\ \mathsf{swap}: \quad & (\mathsf{Race\ set}) \to (\mathsf{Race\ set}) \\ \uplus: \quad & ((\mathsf{Race\ set}) \times (\mathsf{Race\ set})) \to (\mathsf{Race\ set})\end{aligned}$$

Informally, present checks that the two elements of an event pair occur consecutively in a schedule. Additionally, swap makes a change to each element $((e', e''), v, s)$ of a race set, namely to swap e' and e'' both in the first component of the triple and also where they first occur consecutively in s. Finally, $\uplus$ does something akin to a union of two race sets, namely to do the union based only on the event pair of each race. We will maintain the invariant that for a given $c \in \mathsf{EventPair}$, a race set contains at most one race of the form (c, v, s). The idea of $X \uplus Y$ is that if X contains a race of the form (c, v', s'), and Y contains a race of the form (c, v'', s''), then $X \uplus Y$ will, somewhat arbitrarily, contain the first race (c, v', s') (and leave out (c, v'', s'')). Formally,

$$\mathsf{present}((e', e''), (e_1, \ldots, e_n)) = \begin{cases} \mathsf{true} & \text{if } \exists i : e' = e_i \ \wedge \ e'' = e_{i+1} \\ \mathsf{false} & \text{otherwise} \end{cases}$$

$$\begin{aligned}&\mathsf{swap}(X) = \\ &\quad \{\ ((e'', e'), v, (e_1, \ldots, e_{i-1}, e_{i+1}, e_i, e_{i+2}, \ldots, e_n)) \mid \\ &\qquad ((e', e''), v, (e_1, \ldots, e_n)) \in X \wedge \\ &\qquad i \in 1..(n-1) \text{ is the smallest index such that:} \\ &\qquad\quad e' = e_i \ \wedge \ e'' = e_{i+1}\ \}\end{aligned}$$

For every $X \in (\mathsf{Race\ set})$, we assume that if $(c', v', s') \in X$ and $(c'', v'', s'') \in X$ and $c' = c''$, then $v' = v''$ and $s' = s''$. The following definition of $\uplus$ maintains this property.

$$\begin{aligned}X \uplus Y = X \ \cup \{\ &((e', e''), v, s) \in Y \mid \\ &\forall (e_x, v_x, s_x) \in X : e_x \neq (e', e'')\ \}\end{aligned}$$

3.5 Racageddon Overview

Racageddon iterates a combination of concolic execution and schedule improvement. We begin with a run of hybrid

to produce candidate races and then we do two phases of search for races. In the Phase 1 we do a separate search for each of the candidate races. In the Phase 2 we do a search based on the races found in Phase 2. For our benchmarks, our experiments with Racageddon found 291 real races in Phase 1 and 53 additional real races in Phase 2.

In Phase 1 we interleave calls to concolic and improve. The idea is to turn the search for a race into a search for a schedule that leads to the race. Each call to concolic will produce a schedule that gets closer to execute the race, after which a call to improve will further improve that schedule. In more detail, each call to concolic will both try to execute the given schedule *and* continue execution beyond that schedule, typically until termination of the program. Part of the continued execution may make progress towards the desired race. The call to improve will permute some events in the schedule to make the next concolic run have a better chance to succeed. Notice that because improve is idempotent, we apply improve just once.

In Phase 2 we consider each race found in Phase 1 and do a swap of the two racing events in the schedule that lead to the race. The "swapped" schedule leads to a race of the same two events, which in itself provides nothing new. The interesting aspect of the "swapped" schedule is that a concolic execution will continue after the race and may proceed in a different way than the execution in Phase 1. Our experience is that those continued executions may find races that Phase 1 missed. Once Phase 2 finds a new race, we also do a swap of the schedule that led to that new race.

The alternation of improve and concolic steps is considerably more powerful than either one alone. For our benchmarks, our technique finds 344 races, while improve alone (applied to the trace of an initial run) finds only 151 races, and concolic alone finds only 47 races.

3.6 Racageddon Pseudo-code

Figure 1 shows pseudo-code for Racageddon. We will now go over the pseudo-code in detail. We hope our pseudo-code and explanation will enable a better understanding of the approach and enable practitioners to implement Racageddon easily.

The input to the Racageddon procedure is a program while the output is a set of races. The first four lines of Racageddon declares these four variables: (1) a set of race candidates, called $candidates$, that we initialize by a call to hybrid, (2) a set of races, called $races$, that initially is the empty set and that we eventually return as the result of the procedure, (3) a set of races, called r, that we use to hold intermediate results, and (4) a schedule, called $trace$, that holds each trace produced by concolic.

Phase 1 consists of a for-each-loop that tries each of the event pairs in the set of candidates. For each event pair we use a while-loop to do iterations that each does one call to improve and one call to concolic. We use the integer variable i to count the number of iterations and we bound i by 1000

```
(Race set) Racageddon(Program p) {
  (EventPair set) candidates = hybrid(p)
  (Race set) races = ∅
  (Race set) r
  Schedule trace

  /* Phase 1: try the candidates */
  for each EventPair c ∈ candidates do {
    boolean done = false
    int i = 0
    traces = ϵ
    while (¬ done) ∧ (i ≤ 1000) {
      case improve(trace, c) of
        Schedule s : {
          (r, trace) = concolic(p, s)
          races = races ⊎ r
          done = present(c, trace)
        }
        none : {done = true}
      }
      i = i + 1
    }
  }

  /* Phase 2: try swaps of the races */
  (Race set) workset = swap(races)
  for each Race (c, v, s) ∈ workset do {
    (r, trace) = concolic(p, s)
    races = races ⊎ r
    workset = workset ⊎ swap(r)
  }

  return races
}
```

Figure 1. Racageddon.

to ensure that the search terminates, even if the search found no races. In practice, the highest number of calls to improve and concolic we did for any of our benchmarks was 197. So, none of our experiments exercised the condition $i \leq 1000$. We initialize $trace$ to the empty schedule, denoted by ϵ, such that the initial call to improve can work correctly; that call will return ϵ.

The while-loop uses a Boolean-variable $done$ to keep track of whether the search for a particular candidate can be terminated before i reaches 1000. We have two reasons for terminating the search early, which we do by setting $done$ to true. If the candidate pair c is present in the $trace$ executed by concolic, as found by the call present$(e, trace)$, then we can declare success and terminate the search. If the call to improve$(e, trace)$ returns none, then the search has stalled, and we abandon the search. While abandoning a search may seem sad, our experiments do it in some cases. One of the

reasons may be that the race candidate actually isn't a real race!

Notice how each iteration of the while-loop begins with $trace$, improves it to a schedule s (unless improve returns none), which then after execution of concolic turns into a new value for $trace$.

Phase 2 is a workset algorithm that uses the variable $workset$ that holds a set of races. The $workset$ variable holds a set of races still to be processed. Initially $workset$ is the set of races found in Phase 1, but swapped, in the sense that we now want to search for the "swapped" race. The main part of Phase 2 is a for-each-loop that iterates over the elements of $workset$. We use an advanced for-each-loop that works correctly even if elements are added to $workset$ during a run of the for-each-loop. Here, "works correctly" means that the for-each-loop does one iteration per element of $workset$, even if an element is added to $workset$ multiple times or added after the execution of the for-each-loop begins.

For each element of $workset$, Phase 2 makes one call to concolic and collects any races that may be found. For each new race found in Phase 2, we add the race to $workset$ such that we eventually can say that we tried the "swapped" version of every race that we found.

3.7 Example

We now present an example in which we walk through a run of Racageddon on this program with three shared variables and two threads:

x,y,z are shared variables
z has an initial value received from user input

Thread 1:	Thread 2:
l_1: x = 6	l_4: x = 2
l_2: if (z>4)	l_5: if ($z^2 + 5 < x^2$)
l_3: y = 5	l_6: y = 3

We use these abbreviations for events: $e_1 = (1, l_1)$, $e_2 = (1, l_2)$, $e_3 = (1, l_3)$, $e_4 = (2, l_4)$, $e_5 = (2, l_5)$, $e_6 = (2, l_6)$,

The call to hybrid produces two race candidates:

$$candidates = \{(e_1, e_4), (e_1, e_5)\}$$

Now we begin Phase 1 of Racageddon. Suppose the for-each loop first considers the candidate (e_1, e_4).

Now we run the first iteration of the while-loop. Initially $trace$ is the empty schedule so improve returns the empty schedule. Now we run concolic on the empty schedule. Suppose that the initial random input, which becomes the values of the shared variable z, is 0. Nondeterminism can lead to several traces; suppose we get

$$trace = e_1, e_2, e_4, e_5$$

Notice here that we don't get to e_3 because the condition in e_2 fails due to $0 < 4$, and we don't get to e_6 because the condition in e_5 fails due to $z^2 + 5 = 5$ and $x^2 = 4$ and $5 > 4$.

Now we run the second iteration of the while-loop. First we run improve on (e_1, e_4) and $trace$, which produces this permutation of $trace$:

$$trace = e_1, e_4, e_2, e_5$$

Now we run concolic on $trace$, and like above, let us suppose the initial random input leads to z = 0. The execution of concolic finds the race for which we are searching, so we can add that race to $races$:

$$races = \{\ ((e_1, e_4), 0, (e_1, e_4, e_2, e_5))\ \}$$

Like above, we don't get to execute e_3 or e_6; the conditions in e_2 and e_5 fails for the same reasons as above.

Next the for-each-loop in Phase 1 considers the candidate (e_1, e_5).

Now we run the first iteration of the while-loop. Let us assume that this iteration proceeds like the first iteration for (e_1, e_4) so we get:

$$trace = e_1, e_2, e_4, e_5$$

Now we run the second iteration of the while-loop. First we run improve on (e_1, e_5) and $trace$, which produces this permutation of $trace$:

$$trace = e_4, e_5, e_1, e_2$$

Notice that even though e_5 and e_1 occur consecutively, we won't terminate the search because we are looking for (e_1, e_5). Now we run concolic on $trace$, and which leads to an execution with this trace:

$$trace = e_4, e_5, e_1, e_2, e_3$$

for which z had the initial value 10. (We skip the constraints and merely note that they have solution 10, among other solutions.) Note that $trace$ contains e_3 because the condition in e_2 succeeds due to $10 > 4$.

Now we run the third iteration of the while-loop. First we run improve on (e_1, e_5) and $trace$, which produces this permutation of $trace$:

$$trace = e_4, e_1, e_5, e_2, e_3$$

Next, the execution of concolic finds the race for which we are searching, so we can add that race to $races$:

$$\begin{aligned} races \quad = \quad & \{\ ((e_1, e_4), 0, (e_1, e_4, e_2, e_5)), \\ & ((e_1, e_5), 10, (e_4, e_1, e_5, e_2, e_3)\ \} \end{aligned}$$

We don't get to execute e_6 because the condition in e_5 fails due to $z^2 + 5 = 105$ and $x^2 = 36$ and $105 > 36$.

Now the for-each-loop has processed both elements of the set $candidates$, so we are done with Phase 1 and can move

Name	LOC	# threads	Brief description
Sor	1270	5	A successive order-relaxation benchmark
TSP	713	10	Traveling Salesman Problem solver
Hedc	30K	10	A web-crawler application kernel
Elevator	2840	5	A real-time discrete event simulator
ArrayList	5866	26	ArrayList from `java.util`
TreeSet	7532	21	TreeSet from `java.util`
HashSet	7086	21	HashSet from `java.util`
Vector	709	10	Vector from `java.util`
RayTracer	1942	5	Measures the performance of a 3D raytracer
MolDyn	1351	5	N-Body code modeling dynamic
MonteCarlo	3619	4	A financial simulator, using Monte Carlo techniques to price products
Derby	1.6M	64	Apache RDBMS
Colt	110K	11	Open Source Libraries for High Performance Scientific and Technical Computing
ChordTest	62	11	Mini-benchmark; comes with the Chord race detector
Avrora	140K	6	AVR microcontroller simulator
Tomcat	535K	16	Tomcat Apache web application server
Batic	354K	5	Produces a number of Scalable Vector Graphics (SVG) images based on Apache Batic
Eclipse	1.2M	16	Non-GUI Eclipse IDE
FOP	21K	8	XSL-FO to PDF converter
H2	20K	16	Executes a JDBCbench-like in-memory benchmark
PMD	81K	4	Java Static Analyzer
Sunflow	108K	16	Tool for rendering image with raytracer
Xalan	355K	9	XML to HTML transformer
TOTAL	4587K		

Figure 2. Our benchmarks.

on to Phase 2. Notice that we found both candidate races to be real races.

In Phase 2 we consider swapped versions of the two races found in Phase 1:

$$\begin{aligned} workset &= \{\ ((e_4, e_1), 0, (e_4, e_1, e_2, e_5)), \\ &\quad\ ((e_5, e_1), 10, (e_4, e_5, e_1, e_2, e_3)\ \} \end{aligned}$$

Let us here focus on the run with the schedule (e_4, e_1, e_2, e_5). The call to concolic eventually executes $(e_4, e_1, e_2, e_5, e_3)$ and collects these constraints:

$$x = 6 \ \wedge\ z > 4 \ \wedge\ z^2 + 5 < x^2$$

that have solution z = 5. The next concolic execution therefore executes $(e_4, e_1, e_2, e_5, e_3, e_6)$, which contains the race (e_3, e_6). We add that race to $races$:

$$\begin{aligned} races &= \{\ ((e_1, e_4), 0, (e_1, e_4, e_2, e_5)), \\ &\quad\ ((e_1, e_5), 10, (e_4, e_1, e_5, e_2, e_3), \\ &\quad\ ((e_3, e_6), 5, (e_4, e_1, e_2, e_5, e_3, e_6))\ \} \end{aligned}$$

In summary, hybrid produced two candidates races, Phase 1 found both candidates to be real races, and Phase 2 found one additional race. The key reason why we detected the additional race (e_3, e_6) is that the swapping of events lead the concolic execution to a new program state that was not previously reached.

4. Experimental Results

We use the Lime concolic execution engine; Lime is open source, `http://www.tcs.hut.fi/Software/lime`. In our implementation, events are at the Java bytecode level; we use Soot [53] to instrument bytecodes. We ran all our experiments on a Linux CentOs machine with two 2.4 GHz Xeon quad core processors and 32 GB RAM.

4.1 Benchmarks

Figure 2 lists our 23 benchmarks which we have collected from seven sources:

- From ETH Zurich [1, 54]: Sor, TSP, Hedc, Elevator.
- From `java.util`, Oracle's JDK 1.4.2 [2–4, 36]: ArrayList, TreeSet, HashSet, Vector.
- From Java Grande [5, 51]: RayTracer, MolDyn, MonteCarlo.
- From the Apache Software Foundation: [6, 25]: Derby.
- From European Org. for Nuclear Research (CERN) [4, 24]: Colt.
- From the Chord distribution [7]: ChordTest.
- From DaCapo [8, 12]: Avrora, Tomcat, Batic, Eclipse, FOP, H2, PMD, Sunflow, Xalan.

The sizes of the benchmarks vary widely: we have 2 huge (1M+ LOC), 10 large (20K–1M LOC), 8 medium (1K–8K LOC), and 3 small (less than 1K LOC) benchmarks.

Figure 2 also lists the high watermark of how many threads each benchmark runs.

4.2 Race Detectors

We compare Racageddon with one static race detector, namely Chord [34], one hybrid race detector, namely the one that we call Hybrid [35], and four dynamic race detectors, namely FastTrack [23], Goldilocks [21], RaceFuzzer [47], and Pacer [14]. Additionally we compare with a combined dynamic technique that we call FGRP.

Chord is a static technique, and by design it may report false positives; its main objective is to report all real races (or as many as possible).

We discussed Hybrid in Section 2.

FastTrack, Goldilocks, RaceFuzzer, Pacer, and Racageddon are all dynamic techniques that report only real races.

FastTrack and Goldilocks are based on the observation that a race happens if two accesses to a memory location (of which at least one access is a write) are not ordered by the happens-before relation. FastTrack uses a clever representation of the happens-before relation to achieve constant-time overhead for almost all monitored operations. Goldilocks uses a lockset-based algorithm to improve the precision of the computation of the happens-before relation.

RaceFuzzer performs random testing by choosing thread schedules at random and stopping a thread when it is about to execute a statement in a candidate race pair. RaceFuzzer and Racageddon have the following key similarities and differences. The main similarity is that both use Hybrid to generate race candidates and then they guide execution towards those race candidates. The main difference is that RaceFuzzer guides execution with a custom thread scheduler that controls thread interleaving, while Racageddon 1) uses the improve function get a better schedule ahead of execution and 2) interleaves calls to improve and concolic.

Pacer is a sampling-based data race detector that detects any race at a rate equal to the sampling rate. In our experiments, the sampling race was 100% and for each benchmark we used 100 trials. We used a sampling rate of 100% to make Pacer report as many races as possible, even though performance will be the slowest possible.

We use FGRP to stand for the union of FastTrack, Goldilocks, RaceFuzzer, and Pacer in following sense. We can implement FGRP as a tool that for a given benchmark starts runs of FastTrack, Goldilocks, RaceFuzzer, and Pacer in four separate threads, and if any one of them reports a race, then FGRP reports a race.

We implemented Hybrid and RaceFuzzer ourselves according to the published papers that describe them, while we got the implementations of Chord, FastTrack, Goldilocks, and Pacer from webpages and from their authors.

4.3 How we handle Reflection

Many of the benchmarks use reflection, yet each of the race detectors listed above either doesn't support reflection or supports reflection poorly. We overcome this problem with a the help of the tool chain TamiFlex [13].

The core of the problem is that all the race detectors do either a static analysis or some form of ahead-of-time instrumentation. Reflection tends to make static analysis unsound and to load uninstrumented classes. TamiFlex solves these problems in a manner that is sound with respect to a set of recorded program runs. If a later program run deviates from the recorded runs, TamiFlex issues a warning.

We have combined each of the race detectors with TamiFlex and we have run all our experiments without warnings. As a result, the race detectors all handle reflection correctly and in the same way.

4.4 Race Siblings

The seven race detectors differ in how they report race *siblings*. We define that two event pairs are siblings if they have one event in common. Our versions of Hybrid and RaceFuzzer may report race siblings, and also Chord, Pacer, and Racageddon may report race siblings. In contrast, FastTrack and Goldilocks report only one of two siblings. Intuitively, FastTrack and Goldilocks reports zero or one race per memory location, while the other race detectors may report multiple races per memory location. The reader should be aware of this difference when reading the measurements below.

4.5 Measurements

Figure 3 shows the numbers of races found in 23 benchmarks by Racageddon, including whether the races were found in Phase 1 or in Phase 2.

Figure 4 shows, for each benchmark, the number of schedules tried by Racageddon, the length of the longest schedule that found a race, and the number of branches in that longest schedule. One way to understand "the number of branches" is as follows. Suppose, for a given benchmark, we have the longest schedule that found a race and we want to reconstruct the input that led to execution of that schedule. We do that by first generating constraints from the branches in the schedule, and then solving the constraints, which produces an input that will work. Figure 4 shows the number of constraints that were generated in this manner.

Figure 5 shows the numbers of races found in 23 benchmarks by 7 techniques.

Figure 6 shows the time each of the runs took in minutes and seconds, and it shows the geometrical mean for each technique.

Some of the executions of Goldilocks crashed, which we indicate in Figure 5 and Figure 6 with "-". If we compare Figure 5 and Figure 6 we see that for ArrayList and Batic, we list that Goldilocks reported races while we list no execution times. The reason is that for ArrayList and Batic, our runs

	Number of races found		
Name	Total = Phase 1 + Phase 2		
Sor	3	2	1
TSP	2	2	0
Hedc	11	9	2
Elevator	8	5	3
ArrayList	7	7	0
TreeSet	3	3	0
HashSet	8	7	1
Vector	4	4	0
RayTracer	4	3	1
MolDyn	6	4	2
MonteCarlo	3	2	1
Derby	18	15	3
Colt	10	7	3
ChordTest	2	2	0
Avrora	13	12	1
Tomcat	21	19	2
Batic	29	23	6
Eclipse	51	46	5
FOP	18	16	2
H2	39	30	9
PMD	13	12	1
Sunflow	30	22	8
Xalan	41	39	2
TOTAL	344	291	53

Figure 3. Races found by Racageddon.

Name	number of schedules	longest schedule length	# branches
Sor	14	6,803	135
TSP	8	6,047	697
Hedc	28	249,268	4,084
Elevator	28	9,005	401
ArrayList	47	132,990	2,503
TreeSet	17	110,087	2,303
HashSet	38	139,553	2,979
Vector	40	6,308	108
RayTracer	9	71,084	520
MolDyn	188	4,680	362
MonteCarlo	24	12,061	994
Derby	105	108,302,900	39,103
Colt	63	948,033	9,418
ChordTest	2	505	10
Avrora	23	702,961	10,207
Tomcat	197	1,284,917	18,429
Batic	39	1,407,554	10,901
Eclipse	53	102,879,384	23,863
FOP	41	153,074	3,085
H2	35	297,655	7,310
PMD	48	310,049	7,201
Sunflow	37	1,624,320	8,821
Xalan	56	2,907,450	11,937

Figure 4. Schedules tried by Racageddon.

of Goldilocks crashed, yet the execution log contained some races that we report in Figure 5.

Figure 7 shows, for each benchmark, the lengths of the 72 schedules that lead to races found only by Racageddon.

Figure 8 shows, for each benchmark, how many of the races found by Hybrid are actually real races, as found by the combination of FGRP and Racageddon.

4.6 Evaluation

We now present our findings based both on the measurements listed above and on additional analysis of the races that were found.

Racageddon. We can see in Figure 3 that Racageddon found a total of 344 real races, including 291 races found in Phase 1 and 53 races found in Phase 2. The split between Phase 1 and Phase 2 demonstrates a subtlety of race directed scheduling: even when we have a schedule that finds a race, a swap of the race pair can lead to other races.

Number of schedules. We can see in Figure 4 that the number of schedules tried by Racageddon is rather modest and appears to be no worse than the product of a small constant and the number of race candidates. Note that in Racageddon, some runs of concolic finds multiple races. We can also see in Figure 4 that the longest schedules that found races can have lengths that are more than 100,000,000. This shows that the improve method scales to long schedules.

Racageddon versus other Dynamic Techniques. We can see in Figure 5 that Racageddon finds the most races (344) of all the dynamic techniques. Among those 344 races, 72 races were found only by Racageddon and are entirely novel to this paper, while 272 were also found by FGRP. Dually, 32 races were found only by FGRP. In summary, we have that the combination of FGRP and Racageddon found 376 races in the 23 benchmarks.

Found only by FGRP:	32
Found by both:	272
Found only by Racageddon:	72
Total:	376

Let us consider races that Racageddon found but FGRP missed. One such race is a bug in Eclipse, specifically in the class `HudsonSecurityManager` in the package `org.eclipse.hudson.security`. The effect of the race is that a plug-in may fail to load and that the user may have to restart Eclipse.

Dually, let us consider races that FGRP found but that Racageddon missed. One such race is in Derby, specifically in the class `Connection` in the package `org/apache/derby/client/am`, where we find this code:

	Static	Hybrid	Dynamic							
benchmarks	Chord	Hybrid	FastTrack	Goldilocks	RaceFuzzer	Pacer	FGRP	Racageddon		
								total	= new	+ FGRP
Sor	3	8	0	0	0	3	3	3	3	0
TSP	17	3	1	1	0	1	1	2	1	1
Hedc	143	5	3	1	1	11	11	11	4	7
Elevator	54	13	1	-	0	4	4	8	4	4
ArrayList	8	14	0	1	5	6	6	7	1	6
TreeSet	11	13	0	-	6	8	9	3	0	3
HashSet	0	11	0	-	8	7	8	8	0	8
Vector	17	9	0	-	5	5	5	4	0	4
RayTracer	159	2	1	1	1	3	3	4	1	3
MolDyn	92	43	0	1	2	5	5	6	1	5
MonteCarlo	101	5	0	0	1	2	2	3	1	2
Derby	1110	21	1	-	2	14	15	18	4	14
Colt	549	13	0	0	3	7	7	10	3	7
ChordTest	2	2	1	1	2	2	2	2	0	2
Avrora	1887	9	3	3	6	11	12	13	1	12
Tomcat	110061	52	12	11	11	20	20	21	3	18
Batic	970	12	9	10	9	32	35	29	7	22
Eclipse	9401	77	14	-	13	39	43	51	8	43
FOP	34	21	5	5	8	13	15	18	3	15
H2	869	19	5	-	9	25	26	39	13	26
PMD	292	14	9	8	4	13	13	13	0	13
Sunflow	353	16	8	11	9	19	21	30	11	19
Xalan	1003	23	6	9	10	36	38	41	3	38
TOTAL	127136	405	79	63	115	286	304	344	72	272

Figure 5. The numbers of races found in 23 benchmarks by 7 techniques.

```
public void accumulateWarning(SqlWarning e) {
    if (warnings_ == null) {
        warnings_ = e;
    } else {
        warnings_.setNextException(e);
    }
}

public void clearWarningsX() throws SqlException {
    warnings_ = null;
    accumulated440ForMessageProcFailure_ = false;
    accumulated444ForMessageProcFailure_ = false;
    accumulatedSetReadOnlyWarning_ = false;
}
```

The accesses to `warnings_` can form races. The effect of the race may be a null-pointer exception, which can happen in the following way. First a call to `accumulateWarning` sets `warnings_` to `e`, and then a call to `clearWarningsX` sets `warning_` to null, and finally the caller of the method `accumulateWarning` gets a null-pointer exception.

Race siblings. Are the 72 races found only by Racageddon genuinely new or are they merely siblings of other races found by Racageddon? As a step towards an answer to this question, let us define

B = the 272 races found by both Racageddon and FGRP

R = the 72 races found only by Racageddon

In terms of B and R, the question is whether the races in R have siblings in B or R. The answer is that 36 races in R have no siblings at all, each of 28 races in R has a sibling in B, and R contains 6 sibling pairs. Notice that the races in two of the sibling pairs in R also have siblings in B. We conclude that Racageddon finds 36+4 = 40 races that are genuinely new and don't have siblings among the other races found by Racageddon.

FastTrack versus Pacer. Pacer is based on FastTrack and as expected, every race found by FastTrack is also found by Pacer. Pacer finds many more races (286) than FastTrack (79) so our experiments confirm that Pacer is a highly worthwhile extension of FastTrack.

FGRP details. The combined dynamic technique FGRP found 304 races. Pacer was the biggest contributor to that collection of 304 races. Among those 304 races, Pacer found 286, some of which were also found by Goldilocks and RaceFuzzer. The remaining 304-286=18 races were found Goldilocks (10 races) and RaceFuzzer (8 races). In more detail, Goldilocks found additional races in Avrora (1), Batic

	Static	Hybrid	Dynamic				
benchmarks	Chord	Hybrid	FastTrack	Goldilocks	RaceFuzzer	Pacer	Racageddon
Sor	2:18	0:49	0:08	0:44	2:29	9:44	4:49
TSP	2:22	0:55	0:03	0:10	1:50	11:23	4:37
Hedc	4:07	1:00	0:08	0:25	2:01	5:00	3:08
Elevator	1:10	0:39	0:03	-	1:11	3:58	2:40
ArrayList	2:40	0:50	0:05	-	1:18	5:18	4:11
TreeSet	3:11	0:18	0:06	-	0:44	7:02	3:25
HashSet	2:58	0:21	0:06	-	0:59	4:57	2:43
Vector	0:43	0:15	0:01	-	0:38	5:05	2:52
RayTracer	1:24	0:09	0:03	0:38	0:26	4:18	2:22
MolDyn	0:38	1:42	0:02	1:08	2:49	15:36	6:45
MonteCarlo	2:31	2:02	0:04	1:16	4:01	16:31	6:58
Derby	35:09	1:26	0:13	-	1:50	11:34	5:02
Colt	4:37	0:04	0:10	0:23	0:09	4:48	2:23
Chord-Test	0:05	0:01	0:01	0:02	0:05	0:54	0:10
Avrora	19:37	2:40	0:39	4:57	3:19	23:03	11:17
Tomcat	12:01	3:57	0:41	4:11	6:01	45:12	19:00
Batic	27:29	3:01	0:18	-	3:55	30:01	14:54
Eclipse	41:11	3:50	0:35	-	4:14	48:46	19:15
FOP	6:50	0:17	0:12	0:36	0:25	13:21	4:49
H2	8:38	0:31	0:09	-	0:49	18:50	7:31
PMD	15:48	0:16	0:14	1:03	0:38	17:41	7:22
Sunflow	16:00	0:41	0:23	2:01	1:06	18:17	6:03
Xalan	33:11	2:39	0:20	3:00	3:47	30:37	13:19
geom. mean	4:36	0:40	0:08	-	1:16	10:41	4:51

Figure 6. Timings in minutes and seconds.

(3), FOP (2), SunFlow (2), and Xalan (2) (and RaceFuzzer found none of those 10 races). RaceFuzzer found additional races in TreeSet (1), HashSet (1), Derby (1), Eclipse (4), and H2 (1) (and GoldiLocks found none of those 8 races). We conclude that Goldilocks, RaceFuzzer, and Pacer are all worthwhile techniques that each finds races that the other techniques don't find. As a combined dynamic technique FGRP is highly powerful.

Chord. Chord is possibly the best current static race detector, yet our experiments strongly suggest that Chord finds a large number of false positives. We conclude that accurate static race detection continues to be an open problem.

Timings. The geometrical means of the execution times for each technique show that FastTrack and Hybrid are the fastest, while Pacer is the slowest. Racageddon is more than twice as fast as Pacer yet Racageddon finds significantly more races. Note that the timings for RaceFuzzer and Racageddon include the time to execute Hybrid. Note also that we implemented RaceFuzzer ourselves in a rather unoptimized fashion. As a result, our implementation of RaceFuzzer is significantly slower than FastTrack, while the paper on RaceFuzzer [47] reported that the original implementation of RaceFuzzer is faster than FastTrack!

Rare races. Burckhardt, Kothari, Musuvathi, and Nagarakatte [16] characterized the depth of a bug as the minimum number of scheduling constraints required to find that bug. In the spirit of their characterization, Figure 4 lists the longest schedule that Racageddon used to find a race for each benchmark, along with the number of branches in that schedule. Six of those schedules have more than a million events, including one schedule with more than 100 million events. For 18 of those longest schedules, the result was that Racageddon found a race that FGRP didn't find. The exceptions are TSP, Elevator, Vector, MolDyn, and ChordTest, and we notice that those five benchmarks have some of the shortest "longest schedules" among the benchmarks.

Figure 7 lists the lengths of the 72 schedules that lead to races found only by Racageddon. We can groups those lengths as follows:

lengths	#
$10^3 - 10^4$	9
$10^4 - 10^5$	4
$10^5 - 10^6$	28
$10^6 - 10^7$	22
$10^7 - 10^8$	6
$10^8 - 10^9$	3

The table shows that many of those schedules are long. Specifically, 31 races were found with schedules that have

Name	Lengths
Sor	6462, 6661, 6803
TSP	5623
Hedc	57327, 224341, 236804, 249268
Elevator	6573, 7924, 8673, 8914
ArrayList	132990
TreeSet	-
HashSet	-
Vector	-
RayTracer	71084
MolDyn	4305
MonteCarlo	12061
Derby	58483566, 98555637, 105053813, 108302900
Colt	824877, 919592, 948033
ChordTest	-
Avrora	702961
Tomcat	1066481, 1169274, 1284917
Batic	182982, 323737, 760003, 1379402, 1393478, 1400516, 1407554
Eclipse	1697703, 3068331, 3429715, 16605080, 16785570, 77145639, 98049000, 102879384
FOP	134705, 150499, 153074
H2	32742, 116085, 217288, 232170, 241100, 264912, 273842, 276819, 279795, 285748, 294678, 296133, 297655
PMD	-
Sunflow	374598, 730944, 1283212, 1348185, 1478131, 1494379, 1543108, 1575594, 1608075, 1620019, 1624320
Xalan	2674854, 2849301, 2907450

Figure 7. The lengths of the 72 schedules that lead to races found only by Racageddon.

between 1 million and 108 million events, which suggests that they are rare and hard-to-find races.

Hybrid. Both RaceFuzzer and Racageddon use Hybrid to produce race candidates. RaceFuzzer focuses solely on the race candidates, while Racageddon discovers additional race candidates. Overall, Hybrid produces a worthwhile starting point for those two dynamic techniques. We can see in Figure 8 that for our benchmarks, Hybrid reports 405 race candidates of which 238 (59%) are real races. Future work may be able to show that some of the remaining 405-238=167 race candidates are real races.

5. Related Work

In Section 2 we discussed two techniques for race detection, namely one by O'Callahan and Choi [35] and one by Said, Wang, Yang, and Sakallah [41], that we use in Racageddon. In Section 4 we discussed five additional techniques, namely Chord [34], FastTrack [23], Goldilocks [21], RaceFuzzer [47], and Pacer [14] that we have compared experimentally with Racageddon. The goal of this section is to highlight some other notable techniques and tools in the area of race detection and related areas.

Predictive race detectors. The technique in the paper by Said, Wang, Yang, and Sakallah [41] is an example of what some authors call *predictive* techniques. The terminology stems from that if a trace can be reordered into a trace that leads to a deadlock, then the technique will do that. Smaragdakis et al. [50] presented a sound, predictive technique for race detection that works in polynomial time. We can view Smaragdakis et al.'s technique as an example of an improve function. Both the improve function of Said et al. [41] and by Smaragdakis et al. [50] are sound. So, we expect that if we replace one with the other, we will find the same number of races, and possibly faster. We leave experimental confirmation of this expectation to future work.

Swapping and flipping. Racageddon's notion of event swapping in Phase 2 is reminiscent of jCute's notion of flipping [48, 49]. While Racageddon simply swaps the order of two consecutive events, jCute does something more radical: it puts the second event in the position of the first event, and then it delays the thread that executes the first event as long as possible. We have found that a simple swap works well.

Dynamic race detectors. FastTrack, Goldilocks, RaceFuzzer, and Pacer were some of the best dynamic race detectors for Java until now. A predecessor of Pacer, namely LiteRace [33] was the seminal paper that showed how to do race detection in a way that samples and analyzes selected portions of a programs execution. Prior to LiteRace, a paper by Jump, Blackburn, and McKinley [30] presented a sampling technique that they applied in the context of memory management.

	Number of races	
Name	*reported*	*real*
Sor	8	3
TSP	3	1
Hedc	5	5
Elevator	13	7
ArrayList	14	6
TreeSet	13	8
HashSet	11	7
Vector	9	4
RayTracer	2	2
MolDyn	43	5
MonteCarlo	5	3
Derby	21	17
Colt	13	9
ChordTest	2	2
Avrora	9	9
Tomcat	52	20
Batic	12	11
Eclipse	77	46
FOP	21	15
H2	19	17
PMD	14	12
Sunflow	16	6
Xalan	23	23
TOTAL	405	238

Figure 8. Hybrid; *real* = found by FGRP ∪ Racageddon.

Some well known dynamic race detectors work for other languages than Java, including the seminal Eraser [44], and a tool by Sack et al. [40].

Arnold and M. Vechev and E. Yahav [11] presented the QVM run-time environment that continuously monitors an execution and potentially detects defects, including races.

Hybrid race detectors. The technique by O'Callahan and Choi [35] that we call Hybrid continues to be one of the best and most scalable hybrid techniques for race detection. Other hybrid techniques include one by von Praun and Gross [54], RaceTrack [56], and MultiRace [38]. We leave to future work to do a large-scale study of those three hybrid techniques like we did for Hybrid. In particular, future work should evaluate how well those techniques perform when we want to use their output as race candidates for other tools such as RaceFuzzer and Racageddon.

Static race detectors. Chord remains one of the best among the scalable static race detectors to date, hence it was our choice for experimental comparison in this paper. Among the other static race detectors, some use static analysis, including Warlock [52], RacerX [22], LockSmith [39], and Relay [55], some use model checking, including an approach by Henzinger, Jhala, and Majumdar [29], and some use type systems, including an approach based on ownership by Boyapati, Lee, and Rinard [15], and approaches that capture common synchronization patterns by Freund [26] and later by Abadi, Flanagan, and Freund [9]. A related approach based on type systems by Sasturkar, Agarwal, Wang, and Stoller [43] enables specification and check of atomicity. Finally, Effinger-Dean, Boehm, Chakrabarti, and Joisha [20] presented a characterization of extended interference-free regions of C programs in which variables cannot be modified by other threads. All the static approaches may produce false positives and thus have a goal that is dual to our objective to find real races.

Other techniques. We implemented an early version of Racageddon as an extension of Java PathFinder [28]. Our Java PathFinder extension is effective at exploring all execution paths yet doesn't scale up to our current benchmarks.

Collingbourne et al. [19] presented a sound analysis technique for GPU-oriented languages like OpenCL and CUDA, which have concurrency models that are rather different from Java. One of the applications of their technique is to race detection. Intuitively, their main result is that a program has a race if and only a modified version of the program has a race when the threads are executed in lock step.

In the setting of distributed memory and distributed systems, Park et at. [37] presented a race detector for distributed memory along with an implementation for UPC, and Sasnauskas et al. [42] presented a technique for symbolic execution of distributed systems.

6. Conclusion

Racageddon implements a new technique that we call race directed scheduling. Our experiments show that race directed scheduling is efficient and useful, and ultimately that a combination of techniques is currently the best path to successful race detection.

For a large benchmark suite, our tool Racageddon found 72 real races that were missed by earlier techniques, including 31 races that were found with schedules that have between 1 million and 108 million events, which suggests that they are rare and hard-to-find races.

Our experiments also show that a combination of the four tools Goldilocks, Calfuzzer, Pacer, and Racageddon finds a total of 376 real races in our benchmarks. As far as we know, this is the most comprehensive list of real races for those benchmarks that is reported in the literature.

Our experiments validates Hybrid [35] as an excellent choice for producing race candidates. Across our benchmarks, Hybrid produces at most 41% false positives.

Our technique is applicable beyond Java, particularly to any language with a concurrency model based on threads and locks. The main requirements are that (1) the technique embodied in the Hybrid tool [35] applies, (2) the technique embodied in the improve function [41] applies (3) concolic execution [27] can be implemented. The main limitations of Racageddon are due to limitations of the constraint solvers used by concolic and improve.

Acknowledgments. We thank Michael Bond who gave us access to Pacer, and Can Bekar, Tayfun Elmas, Cormac Flanagan, Steve Freund, and Serdar Tasiran who gave us access to Goldilocks. We thank the PPOPP 2014 reviewers for a wealth of suggestions that helped us improve the paper.

References

[1] `https://code.google.com/p/jchord/issues/attachmentText?id=53&aid=530000001&name=SOR.java&token=aYxeQMsIH8M6_VpOVEgmST9XHnw%3A1385586794467`.

[2] `http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/util/ArrayList.java`.

[3] `http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/util/TreeSet.java`.

[4] `http://web.engr.illinois.edu/~sorrent1/penelope/experiments.html`.

[5] `http://www2.epcc.ed.ac.uk/computing/research_activities/jomp/grande.html`.

[6] `http://db.apache.org/derby/derby_downloads.html`.

[7] `https://code.google.com/p/jchord/downloads/detail?name=chord-src-2.1.tar.gz`.

[8] `http://sourceforge.net/projects/dacapobench/files`.

[9] M. Abadi, C. Flanagan, and S. N. Freund. Types for safe locking: Static race detection for Java. *ACM Transactions on Programming Languages and Systems*, 28(2):207–255, 2006.

[10] Sarita V. Adve, Mark D. Hill, Barton P. Miller, and Robert H. B. Netzer. Detecting data races on weak memory systems. In *Proceedings of ISCA, International Symposium on Computer Architecture*, pages 234–243, 1991.

[11] M. Arnold, M. Vechev, and E. Yahav. Qvm: An efficient runtime for detecting defects in deployed systems. In *OOPSLA, ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 143–162, 2008.

[12] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee Intel, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovic, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA'06, 21st annual ACM SIGPLAN conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 169–190, 2006.

[13] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *ICSE, 33rd International Conference on Software Engineering*, May 2011.

[14] Michael D. Bond, Katherine E. Coons, and Kathryn S. McKinley. Pacer: Proportional detection of data races. In *Proceedings of PLDI'10, ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 121–133, 2010.

[15] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *OOPSLA, ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 211–230, 2002.

[16] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. A randomized scheduler with probabilistic guarantees of finding concurrency bugs. In *ASPLOS, International Conference on Architectural Support for Programming Languages and Operating Systems*, March 2010.

[17] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *Proc. 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 443–446, 2008.

[18] Cristian Cadar, Paul Twohey, Vijay Ganesh, and Dawson Engler. Exe: A system for automatically generating inputs of death using symbolic execution. In *Proceedings of 13th ACM Conference on Computer and Communications Security*, 2006.

[19] Peter Collingbourne, Alastair F. Donaldson, Jeroen Ketema, and Shaz Qadeer. Interleaving and lock-step semantics for analysis and verification of gpu kernels. In *Proceedings of ESOP, European Symposium on Programming*, pages 270–289. Springer-Verlag (*LNCS*), 2013.

[20] Laura Effinger-Dean, Hans-J. Boehm, Dhruva Chakrabarti, and Pramod Joisha. Extended sequential reasoning for data-race-free programs. In *ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, June 2011.

[21] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Goldilocks: Efficiently computing the happens-before relation using locksets. In *Proceedings of FATES/RV, Workshop on Formal Approaches to Testing and Runtime Verification*, pages 193–208, 2006.

[22] D. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *SOSP, Nineteenth ACM Symposium on Operating Systems Principles*, pages 237–252, 2003.

[23] Cormac Flanagan and Stephen N. Freund. Fasttrack: Efficient and precise dynamic race detection. In *Proceedings of PLDI'09, ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 121–133, 2009.

[24] European Organization for Nuclear Research (CERN). Colt. `http://acs.lbl.gov/software/colt/`.

[25] Apache Software Foundation. Derby. `http://db.apache.org/derby`.

[26] S. N. Freund. Type-based race detection for Java. In *PLDI, ACM SIGPLAN 2000 Conference on Programming language design and implementation*, pages 219–232, 2000.

[27] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *Proceedings of PLDI'05, ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 213–223, 2005.

[28] Klaus Havelund and Thomas Pressburger. Model checking Java programs using Java pathfinder. *Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[29] Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Race checking by context inference. In *Proceedings of PLDI'04, ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–13, 2004.

[30] M. Jump, S. M. Blackburn, and K. S. McKinley. Dynamic object sampling for pretenuring. In *ICMM, ACM International Symposium on Memory Management*, pages 152–162, 2004.

[31] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, C–28,9:690–691, September 1979.

[32] Rupak Majumdar and Ru-Gang Xu. Directed test generation using symbolic grammars. In *Proceedings of the twenty-second IEEE/ACM International Conference on Automated Software Engineering*, 2007.

[33] Daniel Marino, Madanlal Musuvathi, and Satish Narayanasamy. Literace: Effective sampling for lightweight data-race detection. In *Proceedings of PLDI'09, ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 134–143, 2009.

[34] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for Java. In *Proceedings of PLDI'06, ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2006.

[35] Robert O'Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. In *Proceedings of the ACM Conference on Principles and Practice of Parallel Programming*, pages 167–178, 2003.

[36] Oracle. JDK, 1.4.2. `http://www.oracle.com/technetwork/java/javase/index-jsp-138567.html`.

[37] Chang-Seo Park, Koushik Sen, Paul Hargrove, and Costin Iancu. Efficient data race detection for distributed memory parallel programs. In *Proceedings of IEEE Conference on Supercomputing*, 2011.

[38] E. Pozniansky and A. Schuster. MultiRace: Efficient on-the-fly data race detection in multithreaded C++ programs. *Concurrency and Computation: Practice and Experience*, 19(3):327–340, 2007.

[39] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. LockSmith: Context sensitive correlation analysis for race detection. In *PLDI, ACM Conference on Programming Language Design and Implementation*, pages 320–331, 2006.

[40] P. Sack, B. E. Bliss, Z. Ma, P. Petersen, and J. Torrellas. Accurate and efficient filtering for the Intel thread checker race detector. In *ASID, 1st workshop on Architectural and System Support for Improving Software Dependability*, pages 34–41, 2006.

[41] Mahmoud Said, Chao Wang, Zijiang Yang, and Karem A. Sakallah. Generating data race witnesses by an SMT-based analysis. In *NASA Formal Methods*, pages 313–327, 2011.

[42] Raimondas Sasnauskas, Oscar Soria Dustmann, Benjamin Lucien Kaminski, Klaus Wehrle, Carsten Weise, and Stefan Kowalewski. Scalable symbolic execution of distributed systems. In *Proceedings of ICDCS, International Conference on Distributed Computing Systems*, pages 333–342, 2011.

[43] A. Sasturkar, R. Agarwal, L. Wang, and S. D. Stoller. Automated type-based analysis of data races and atomicity. In *PPoPP, Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 83–94, 2005.

[44] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *Transactions on Computer Systems*, 15(4):391–411, November 1997.

[45] Koushik Sen. Concolic testing. In *Proceedings of the twenty-second IEEE/ACM International Conference on Automated Software Engineering*, pages 571–572, 2007.

[46] Koushik Sen. Effective random testing of concurrent programs. In *IEEE/ACM International Conference on Automated Software Engineering*, pages 323–332, 2007.

[47] Koushik Sen. Race directed random testing of concurrent programs. In *Proceedings of PLDI'08, ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 11–21, Tucson, Arizona, June 2008.

[48] Koushik Sen and Gul Agha. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In *Proc. 18th International Conference on Computer Aided Verification*, pages 419–423, 2006.

[49] Koushik Sen and Gul Agha. A race-detection and flipping algorithm for automated testing of multi-threaded programs. In *Proceedings of Haifa Verification Conference*, pages 166–182, 2006.

[50] Yannis Smaragdakis, Jacob Evans, Caitlin Sadowski, Jaeheon Yi, and Cormac Flanagan. Sound predictive race detection in polynomial time. In *Proceedings of POPL'12, SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 387–400, 2012.

[51] L. A. Smith, J. M. Bull, and J. Obdrzalek. A parallel Java grande benchmark suite. November 2001.

[52] N. Sterling. WARLOCK – a static data race analysis tool. In *USENIX Winter Technical Conference*, pages 97–106, 1993.

[53] Raja Vallé-Rai, Etienne Gagnon, Laurie Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing Java bytecode using the soot framework: Is it feasible? In *Proceedings of CC'00, International Conference on Compiler Construction*. Springer-Verlag (*LNCS*), 2000.

[54] C. von Praun and T. R. Gross. Object race detection. In *OOPSLA, ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 70–82, 2001.

[55] J. W. Voung, R. Jhala, and S. Lerner. RELAY: Static race detection on millions of lines of code. In *European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 205–214, 2007.

[56] Y. Yu, T. Rodeheffer, and W. Chen. RaceTrack: Efficient detection of data race conditions via adaptive tracking. In *SOSP, ACM Symposium on Operating Systems Principles*, pages 221–234, 2005.