Race Directed Random Testing of Concurrent Programs

Koushik Sen
EECS Department, UC Berkeley, CA, USA.
ksen@cs.berkeley.edu

Abstract

Bugs in multi-threaded programs often arise due to data races. Numerous static and dynamic program analysis techniques have been proposed to detect data races. We propose a novel randomized dynamic analysis technique that utilizes potential data race information obtained from an existing analysis tool to separate real races from false races without any need for manual inspection. Specifically, we use potential data race information obtained from an existing dynamic analysis technique to control a random scheduler of threads so that real race conditions get created with very high probability and those races get resolved randomly at runtime. Our approach has several advantages over existing dynamic analysis tools. First, we can create a real race condition and resolve the race randomly to see if an error can occur due to the race. Second, we can replay a race revealing execution efficiently by simply using the same seed for random number generation—we do not need to record the execution. Third, our approach has very low overhead compared to other precise dynamic race detection techniques because we only track all synchronization operations and a single pair of memory access statements that are reported to be in a potential race by an existing analysis. We have implemented the technique in a prototype tool for Java and have experimented on a number of large multi-threaded Java programs. We report a number of previously known and unknown bugs and real races in these Java programs.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; D.2.5 [*Software Engineering*]: Testing and Debugging

General Terms Languages, Algorithms, Verification

Keywords race detection, dynamic analysis, random testing, concurrency

PLDI'08, June 7–13, 2008, Tucson, Arizona, USA.

1. Introduction

Multi-threaded programs often exhibit wrong behaviors due to data races. Such concurrent errors–such as data races and deadlocks–are often difficult to find because they typically happen under very specific interleavings of the executing threads. A traditional method of testing concurrent programs is to repeatedly execute the program with the hope that different test executions will result in different interleavings. There are a few problems with this approach. First, the outcome of such testing can be highly dependent on the test environment. For example, some interleavings may only occur on heavily-loaded test systems. Second, this kind of testing depends on the underlying operating system or the virtual machine for thread scheduling—it does not try to explicitly control the thread schedules; therefore, such testing often ends up executing the same interleaving many times. Despite these limitations, such testing is an attractive technique for finding bugs in concurrent systems for several reasons: 1) testing is inexpensive compared to sophisticated techniques such as model checking and verification, 2) testing often scales to very large programs.

Numerous program analysis techniques have been developed to detect and predict data races in multi-threaded programs. Despite recent advances, these techniques often report many data races that are false warnings. For example, a hybrid dynamic race detection tool [37] reports 51 data races for `tomcat`, out of which 39 are false warnings. Similarly, a static race detection tool [33] reports 19 data races in `hedc`, out of which 13 are false warnings. Moreover, being imprecise in nature, most of these tools require manual inspection to see if a race is real or not. Nevertheless, these tools are very effective in finding data races because they can predict data races that could potentially happen during a real execution—for such a prediction, they do not need to see an actual execution (in case of static race detection) or they need to see one real concurrent execution (in case of dynamic race detection.)

Imprecision in race detection can be eliminated by a *precise* dynamic race detection technique, called happens-before race detection [44]. However, it has three problems: first, it can only detect a race if it really happens in an execution and therefore, cannot predict a potential race. Second, this technique has a very large runtime overhead as it needs to track every shared memory access at runtime. Third, since it tracks shared memory accesses at runtime, it can perturb an execution and can prevent the observation of a race that can happen when memories are not tracked. Although, the second problem can be alleviated by using off-line analysis [34], there is no easy solution for the other two problems.

We propose a new technique for finding bugs in concurrent programs, called *race-directed random testing* (or RACEFUZZER.) RACEFUZZER combines race detection with a randomized thread scheduler in order to find real race conditions in a concurrent program with high probability and to discover if the detected real races could cause an exception or an error in the program. The technique works as follows. RACEFUZZER first uses an existing imprecise race detection technique, such as hybrid dynamic race detection, to compute a set of pairs of program statements that could potentially race in a concurrent execution. For each pair in the set, also called *a racing pair of statements*, RACEFUZZER then executes the program with a random schedule. In the random schedule, at each program state, a thread is picked randomly and its next statement is executed with the following exception. If the next statement of the randomly picked thread is contained in the racing pair of statements, then the execution of the statement is *postponed* until another thread is about to execute a statement in the racing pair and the execution of the statement results in a race with the execution of the postponed statement. We say that the execution of two statements are in race if they could be executed by different threads temporally next to each other and both access the same memory location and at least one of the accesses is a write. If RACEFUZZER discovers such a situation where the execution of the next statement by a thread could race with the execution of a postponed statement, then RACEFUZZER reports a real race. In this situation, RACEFUZZER also randomly picks one of the two statements to execute next and continues to postpone the execution of the other statement. Such a random resolution of real races helps RACEFUZZER to find if an exception or an error (such as an assertion violation) can happen due to the race. *In summary,* RACEFUZZER *actively controls a randomized thread scheduler of concurrent program based on potential data races discovered by an imprecise race detection technique.*

RACEFUZZER has several useful features.

- **Classifying real races from false alarms.** RACEFUZZER actively controls a randomized thread scheduler so that real race conditions get created with very high probability. (In Section 3.2, we explain our claim about high probability through an example and empirically validate the claim in Section 5.) *This enables the user of* RACEFUZZER *to automatically separate real races from false warnings, which is otherwise done through manual inspection.*
- **Inexpensive replay of a concurrent execution exhibiting a real race.** RACEFUZZER provides a concrete concurrent execution that exhibits a real race—two racing events in the concurrent execution are brought temporally next to each other. Moreover, it allows the user to *replay* the concrete execution by setting the same seed for random number generation. An appealing feature of this replay mechanism is that it requires no recording of events making the replay mechanism lightweight. The replay feature is a useful tool for *debugging real races*.
- **Separating some harmful races from benign races.** RACEFUZZER randomly re-orders two racing events. This enables RACEFUZZER to find if a race could cause a real exception in the program. As a result harmful races that could lead to errors get detected.
- **No false warnings.** RACEFUZZER gives *no false warnings* about races because it actually creates a race condition by bringing two racing events temporally next to each other.
- **Embarrassingly parallel.** Since different invocations of RACEFUZZER are independent of each other, performance of RACEFUZZER can be increased linearly with the number of processors or cores.

Although in RACEFUZZER, a randomized thread scheduler is directed by potential race conditions, we can bias the random scheduler by other potential concurrency problems such *as potential atomicity violations, atomic-set serializability violations [51], or potential deadlocks*. The only thing that the random scheduler needs to know is a set of statements whose simultaneous execution could lead to a concurrency problem. Such sets of problematic statements could be provided by a static or dynamic analysis technique [23, 22, 2].

We have implemented RACEFUZZER in a prototype tool for Java. The tool has been applied to a number of large benchmarks having a total of 600K lines of code. The results of these experiments demonstrate two hypotheses.

- RACEFUZZER can create real race conditions with very high probability. (We give also give intuitive reasons behind this claim using an example in Section 3.2.) RACEFUZZER can also effectively find subtle bugs in large programs.
- RACEFUZZER detects all known real races in known benchmarks. This shows that RACEFUZZER misses no real races that were predicted and manually confirmed by other dynamic analysis techniques.

To our best knowledge, RACEFUZZER is the first technique of its kind that exploits existing race detection techniques to make dynamic analysis of concurrent programs more effective and informative for debugging. Despite the various advantages of RACEFUZZER, it has some limitations. First, being dynamic in nature, RACEFUZZER cannot detect all real races in a concurrent program—it detects a real race if the race can be produced with the given test harness for some thread schedule. This can be alleviated by combining RACEFUZZER with a symbolic execution technique. Second, being random in nature, RACEFUZZER may not be able to separate all real races from potential races. However, this did not happen in our experiments with existing benchmarks. Third, RACEFUZZER may not be able to separate all harmful races from the set of real races because we say that a race is harmful only if it causes an exception or an error in the program. A harmful race may not raise an exception, but produce wrong results, in which case, RACEFUZZER cannot say if a race is harmful.

2. Algorithm

In this section, we give a detailed description of the RACEFUZZER algorithm. We describe RACEFUZZER using a simple abstract model of concurrent systems.

2.1 Background Definitions

We consider a concurrent system composed of a finite set of threads. Each thread executes a sequence of statements and communicates with other threads through shared objects. In a concurrent system, we assume that each thread terminates after the execution of a finite number of statements. At any point in the execution, a concurrent system is in a *state*. Let S be the set of states that can be exhibited by a concurrent system starting from the initial state s_0. A concurrent system evolves from one state to another state when a thread executes a statement of the program. We assume that a statement in the program can access at most one shared object—this can be achieved by translating a standard program into 3-address code. Next we introduce some definitions that we will use to describe our algorithms.

- `Enabled`(s) denotes the set of threads that are enabled in the state s. A thread is disabled if it is waiting to acquire a lock already held by some other thread (or waiting on a `join` or a `wait` in Java.)
- `Alive`(s) denotes the set of threads whose executions have not terminated in the state s. A state s is in *deadlock* if the set of enabled threads at s (i.e. `Enabled`(s)) is empty and the set of threads that are alive (i.e. `Alive`(s)) is non-empty.
- `Execute`(s, t) returns the state after executing the next statement of the thread t in the state s.
- `NextStmt`(s, t) denotes the next statement that the thread t would execute in the state s.

The following definitions are only required to briefly describe the hybrid race detection algorithm [37]. The execution of a concurrent program can be seen as a sequence of events $\langle e_i \rangle$ where an event denotes the execution of a statement by a thread. An event e can be of the following three forms.

- MEM(σ, m, a, t, L) denotes that thread t performed an access $a \in \{$ WRITE, READ $\}$ to memory location m while holding the set of locks L and executing the statement σ.
- SND(g, t) denotes the sending of a message with unique id g by thread t.
- RCV(g, t) denotes the reception of a message with unique id g by thread t.

An important relation that is used by the hybrid race detection algorithm is the *happens-before relation* on events exhibited by a concurrent execution. Given an event sequence $\langle e_i \rangle$, the happens-before relation $\prec$ is the smallest relation satisfying the following conditions.

- If e_i and e_j are events from the same thread and e_i comes before e_j in the sequence $\langle e_i \rangle$, then $e_i \prec e_j$.
- If e_i is the sending of the message g and e_j is the reception of the message g, then $e_i \prec e_j$.
- $\prec$ is transitively closed.

2.2 The RACEFUZZER Algorithm

In this section, we describe an algorithm that actively controls a random thread scheduler to create real races and to detect errors that could happen due to real races. The algorithm works in two phases. The first phase computes *a set of pairs of statements* that could potentially race during a concurrent execution. The second phase uses each element from the set to control the random scheduling of the threads in a way so that the real racing events could be brought temporally next to each other in the schedule. The first phase of the algorithm uses hybrid race detection [37], an imprecise, but effective, technique for detecting pairs of statements that could potentially race in a concurrent execution. Although we use hybrid race detection in the first phase, any other static or dynamic race detection technique could be used instead.

Phase 1: Hybrid-Race Detection.

We next briefly summarize the hybrid-race detection algorithm [37] that we have implemented in our tool. At runtime, the algorithm checks the following condition for each pair of events (e_i, e_j).

$$\begin{array}{l} e_i = \text{MEM}(\sigma_i, m_i, a_i, t_i, L_i) \wedge e_j = \text{MEM}(\sigma_j, m_j, a_j, t_j, L_j) \\ \wedge t_i \neq t_j \wedge m_i = m_j \wedge (a_i = \text{WRITE} \vee a_j = \text{WRITE}) \\ \wedge L_i \cap L_j = \emptyset \wedge \neg(e_i \prec e_j) \wedge \neg(e_j \prec e_i) \end{array}$$

The above condition states that two events are in race if in those events two threads access the same memory location without holding a common lock, at least one of the accesses is a write, and the two accesses are concurrent to each other (i.e. one access does not happens-before the other.) If the condition holds for a pair of events (e_i, e_j), then we say (σ_i, σ_j) is a racing pair of statements. The computation of the relation $\prec$ is done by maintaining a vector clock with every thread. The events that are classified as $\text{SND}(g, t)$ and $\text{RCV}(g, t)$ events are of the following types. If thread t_1 starts a thread t_2, then events $\text{SND}(g, t_1)$ and $\text{RCV}(g, t_2)$ are generated, where g is a unique message id. If thread t_1 calls t_2.`join()` and t_2 terminates, then events $\text{SND}(g, t_2)$ and $\text{RCV}(g, t_1)$ are generated, where g is a unique message id. If a o.`notify()` on thread t_1 signals a o.`wait()` on thread t_2, then events $\text{SND}(g, t_1)$ and $\text{RCV}(g, t_2)$ are generated, where g is a unique message id. Note that the above algorithm requires us to track every shared memory access and every lock acquire and release operations. Therefore, hybrid race detection can have significant runtime overhead. Several optimizations have been proposed [37] to reduce the runtime overhead.

Phase 2. RACEFUZZER.

Our key contribution is the second phase of the algorithm, which we next describe informally. Let (σ_1, σ_2) be a pair of statements that have been inferred to be potentially racing in the first phase. Due to the imprecision of the first phase, these two statements may not actually race in an actual execution. Therefore, in the second phase we try to control our scheduler randomly based on this pair. Specifically, we execute the various threads following a random schedule (i.e. at each state we pick an enabled thread randomly), but whenever a thread is about to execute a statement in $\{\sigma_1, \sigma_2\}$, we postpone the execution of the thread. The postponed thread keeps on waiting until another thread is about to execute a statement in $\{\sigma_1, \sigma_2\}$ and the execution of the statement actually races with the first thread, i.e. both threads access the same memory location if they execute their next statements and one of the accesses is a write. In that scenario, we randomly resolve the race by allowing one thread between the two threads to execute the next statement and keep postponing the other thread. Note that in the above scenario, we have detected a real race and we have also resolved the race randomly so that we can observe if something bad can happen due to the race. While postponing threads, it may happen that several threads are about to execute a statement in $\{\sigma_1, \sigma_2\}$, but they are not racing because they would access different dynamic shared memory locations when they execute their next statements. In such a case, we keep postponing all the threads that are about to execute a statement in $\{\sigma_1, \sigma_2\}$. At any point, if we manage to postpone all the threads, then we pick a random thread from the set to break the deadlock.

The formal description of the RACEFUZZER algorithm is given in Algorithm 1 and Algorithm 2. The algorithm takes as an input s_0, the initial state of the program, and *RaceSet*, a set of two statements that could potentially race in a concurrent execution. The algorithm maintains a set *postponed* that contains all the threads whose execution has been delayed in order to bring two racing events next to each other. The next statements to be executed by these threads belong to the set *RaceSet*.

Algorithm 1 Algorithm RACEFUZZER

Inputs: the initial state s_0, a set of two racing statements *RaceSet*
$s := s_0$
postponed $:= \emptyset$
while $\texttt{Enabled}(s) \neq \emptyset$ **do**
  $t :=$ a random thread in $\texttt{Enabled}(s) \setminus$ *postponed*
  if $\texttt{NextStmt}(s, t) \in$ *RaceSet* **then**
    $R := \texttt{Racing}\ (s, t, \textit{postponed})$
    if $R \neq \emptyset$ **then** /* Actual race detected */
      print "ERROR: actual race found"
      /* Randomly resolve race */
      if random boolean **then**
        $s := \texttt{Execute}(s, t)$
      else
        postponed $:=$ *postponed* $\cup \{t\}$
        for all $t' \in R$ **do**
          $s := \texttt{Execute}(s, t')$
          postponed $:=$ *postponed* $\setminus \{t'\}$
        end for
      end if
    else /* Wait for a race to happen */
      postponed $:=$ *postponed* $\cup \{t\}$
    end if
  else
    $s := \texttt{Execute}(s, t)$
  end if
  if *postponed* $= \texttt{Enabled}(s)$ **then**
    remove a random element from *postponed*
  end if
end while
if $\texttt{Active}(s) \neq \emptyset$ **then**
  print "ERROR: actual deadlock found"
end if

Algorithm 2 Function $\texttt{Racing}(s, t, \textit{postponed})$

Inputs: program state s, thread t, and set *postponed*
return $\{t' \mid t' \in \textit{postponed}$ s.t. $\texttt{NextStmt}(s, t)$ and $\texttt{NextStmt}(s, t')$ access the same memory location and at least one of the accesses is a write$\}$

The algorithm runs in a loop until there is no enabled thread in the execution. At the termination of the loop, RACEFUZZER reports an actual deadlock if there is at least one active thread in the execution. In each iteration of the loop, RACEFUZZER executes some statements of the program as follows. RACEFUZZER picks a random thread t that is enabled and that has not been postponed. If the next statement of the thread is not in the set *RaceSet*, then RACEFUZZER executes the next statement. This is the trivial case. Otherwise, if the next statement of t is in the set *RaceSet*, then RACEFUZZER computes a subset R of the set *postponed*. The set R contains all threads of *postponed*, such that the execution of the next statement of a thread in R access the same dynamic shared memory location as the next statement of the thread t and at least one of the accesses is a write. The computation of the set R is done by the function `Racing` described in Algorithm 2.

If the set R is non-empty, then RaceFuzzer has brought at least two threads, i.e. the thread t and any thread in R, such that the execution of the next statements by the two threads are in race. At this point, RaceFuzzer reports a real race. RaceFuzzer then randomly resolves these races either by executing the next statement of the thread t or by executing the next statements of all the threads in R. If RaceFuzzer chooses to execute the next statements of the threads in R, then the thread t is placed in the *postponed* set and the threads in R are removed from the *postponed* set. We next point out some key observations about the *postponed* and R sets. The execution of the next statements of the threads in *postponed* cannot mutually race because whenever a race happens, RaceFuzzer resolves the race by executing one element of a racing pair. This also implies that the execution of the next statements of the threads in R cannot mutually race. Another observation is that R can contain more than one element because the next statements of the threads in R can read access the same memory location.

If the set R is empty, then there is no real race. Therefore, RaceFuzzer adds t to the set *postponed* so that it can wait for a real race to happen. At the end of each iteration of the main loop in the RaceFuzzer algorithm, it may happen that the set *postponed* is equal to the set of all enabled threads. This results in a deadlock situation in the RaceFuzzer algorithm because in the next iteration RaceFuzzer has no thread for scheduling. RaceFuzzer breaks this deadlock situation by randomly removing one thread from the set *postponed*.

After the termination of the main loop in RaceFuzzer, the set of enabled threads is empty. This implies that either all the threads have died or some threads have reached a deadlock situation. In the latter case, RaceFuzzer reports a real deadlock.

In RaceFuzzer, we can trivially *replay* a concurrent execution by picking the same seed for random number generation. This is because RaceFuzzer ensures that at any time during execution only one thread is executing and it resolves all non-determinism in picking the next thread to execute by using random numbers. Deterministic replay is a powerful feature of RaceFuzzer because it allows the user to replay and debug a race condition.

3. Advantages of RaceFuzzer

3.1 Example 1 illustrating RaceFuzzer

Figure 1 shows a two-threaded program with a real race. For the simplicity of description, instead of using Java, we use pseudo code to describe the example program. The variables `x, y, z` and the lock `L` are shared between the two threads. The values of `x, y,` and `z` are initialized to 0.

If all statements of `thread1` execute first, then `ERROR1` is not reached. Otherwise, if all statements of `thread2` execute first, then `ERROR1` is reached. This happens due to a race over the variable `z`—statement 7 and statement 5 of the program can be executed by the two threads, respectively, without any synchronization between them. There is no race over the variable `y` because any access to `y` is protected by the lock `L`.

```
        Initially: x = y = z = 0;

thread1 {                 thread2 {
1:  x = 1;                7:   z = 1;
2:  lock(L);              8:   lock(L);
3:  y = 1;                9:   if (y==1) {
4:  unlock(L);            10:     if (x != 1){
                          11:        ERROR2;
5:  if (z==1)             12:     }
6:    ERROR1;             13:  }
}                         14:  unlock(L);
                          }
```

Figure 1. A program with a real race

The accesses to the variable `x` may appear to be in race because such accesses are not consistently protected by a single lock. However, the accesses are implicitly synchronized by the variable `y`. As such, the execution of the statements 1 and 10 (i.e. the statements accessing `x`) cannot be brought temporally next to each other in the two threads. Therefore, there is no race over the accesses to `x`. Hybrid race detection technique will, however, report that there is a race over the variable `x`.

We now illustrate the RaceFuzzer algorithm using the example. In the first phase of the algorithm, hybrid race detection will report that statement pairs $(5, 7)$ and $(1, 10)$ are in race. In the second phase, we will invoke Algorithm 1 with *RaceSet* initialized to $\{5, 7\}$ and $\{1, 10\}$. For each value of *RaceSet*, the algorithm will be invoked several times with different random seeds. Let us consider the two cases corresponding to two different initializations of *RaceSet*.

Case 1: *RaceSet* = $\{1, 10\}$. In this case, it is not possible for `thread2` to first reach statement 10. If `thread1` first reaches statement 1, then it will delay the execution of the thread until it sees the execution of statement 10 by `thread2`. However, since `y = 0`, `thread2` will not execute statement 10 and will terminate. Following the pseudocode at line 26 of Algorithm 1, `thread1` will be removed from *postponed* and it will execute the remaining statements. Therefore, no real race will be reported.

Case 2: *RaceSet* = $\{5, 7\}$. If `thread1` first reaches statement 5, then it starts waiting. `thread2` then reaches statement 7 and RaceFuzzer reports a real race. Depending on whether statement 7 or statement 5 is executed next, `ERROR1` is reached or not executed, respectively. The same happens if `thread2` first reaches statement 7.

The above example shows that RaceFuzzer can detect and create a real race situation without giving any false

```
     Initially: x = 0;

thread1 {                 thread2 {
1.  lock(L);              10.   x = 1;
2.  f1();                 11.   lock(L);
3.  f2();                 12.   f6();
4.  f3();                 13.   unlock(L);
5.  f4();                 }
6.  f5();
7.  unlock(L);
8.  if (x==0)
9.    ERROR;
}
```

Figure 2. A program with a hard to reproduce real race

warning. Hybrid race detection, or similar imprecise techniques, can, on the other hand, give false warnings. RaceFuzzer detects the only real race in the program. It also creates a couple of scenarios, or concurrent executions, to illustrate the race. One such scenario shows the reachability of `ERROR1`. Moreover, RaceFuzzer provides full functionality to replay these scenarios.

3.2 Example 2 illustrating that RaceFuzzer can detect races with high probability

We use the two-threaded program in Figure 2 to argue that RaceFuzzer can create a real race condition with high probability compared to an algorithm using the default scheduler or a simple random scheduler.

The program uses a shared variable `x` which is initialized to 0. The important statements in this program are statements 8, 9, and 10. We add the other statements in the program to ensure that statement 8 gets executed after the execution of a large number of statements by `thread1` and statement 10 gets executed by `thread2` at the beginning. This snippet represents a pattern in real-world programs.

If we run the program with the default scheduler or use a simple randomized scheduler, then the probability of executing statements 8 and 10 temporally next to each other is very low. In fact, with high probability, the execution of statements 8 and 10 will be separated by the acquire and the release of the lock `L`. As such a happens-before race detector will not be able to detect the race with high probability. Moreover, in this example, it is very unlikely that statement 10 will be executed after statement 8. This implies that `ERROR` will not be executed with very high probability. The probability of detecting the race and reaching the `ERROR` statement depends on the number of statements before statement 8. The probability becomes lower as the number of statements before statement 8 is increased.

We now show that RaceFuzzer creates the real race with probability 1 and reaches the `ERROR` statement with probability 0.5. Moreover, we show that this probability is independent of the number of statements before statement 8.

In the first phase, hybrid race detection will predict that statement 8 and statement 10 could be in race. The RaceFuzzer algorithm will then be invoked with *RaceSet* initialized to $(8, 10)$. For any thread schedule, either `thread1` will get postponed at statement 8 or `thread2` will get postponed at statement 10. Therefore, RaceFuzzer will create the race condition with probability 1. In either case, RaceFuzzer will resolve the race and execute `thread1` with probability 0.5. Therefore, the probability that `thread1` reaches the `ERROR` statement is 0.5.

The above example shows that in some situations even if two racing statements are separated by many statements in a real execution, they can be brought temporally next to each other with high probability by RaceFuzzer. As such RaceFuzzer can create real race conditions with very high probability. Our experimental results in Section 5.2 support this fact.

4. Implementation

RaceFuzzer can be implemented for any language that supports threads and shared memory programming, such as Java or C/C++ with pthreads. We have implemented the RaceFuzzer algorithm only for Java. The implementation is part of the CalFuzzer tool set [45] developed to experiment with various smart random testing algorithms. RaceFuzzer instruments Java bytecode to observe various events and to control the thread scheduler. Bytecode instrumentation allows us to analyze any Java program for which the source code is not available. The instrumentation inserts various methods provided by RaceFuzzer inside Java programs. These methods implement both hybrid-race detection and the RaceFuzzer algorithm.

The implementation of the hybrid-race detection algorithm is not an optimized one. This is because the goal of this work is to implement and experiment with the RaceFuzzer algorithm. As such the implementation of the hybrid-race detection algorithm runs slower than the optimized implementation reported in [37].

The instrumentor of RaceFuzzer modifies all bytecode associated with a Java program including the libraries it uses, except for the classes that are used to implement RaceFuzzer. This is because RaceFuzzer runs in the same memory space as the program under analysis. RaceFuzzer cannot track lock acquires and releases by native code. As such, there is a possibility that RaceFuzzer can go into a deadlock if there are synchronization operations inside uninstrumented classes or native code. To avoid such scenarios, RaceFuzzer runs a monitor thread that periodically polls to check if there is any deadlock. If the monitor discovers a deadlock, then it removes one thread from the set *postponed*.

RaceFuzzer can also go into livelocks. Livelocks happen when all threads of the program end up in the *postponed* set, except for one thread that does something in a loop without synchronizing with other threads. We observed such livelocks in a couple of our benchmarks including `moldyn`. In the presence of livelocks, these benchmarks work correctly because the correctness of these benchmarks assumes that the underlying Java thread scheduler is fair. In order to avoid livelocks, RACEFUZZER creates a monitor thread that periodically removes those threads from the *postponed* set that are waiting for a long time.

In [31], it has been shown that it is sufficient to perform thread switches before synchronization operations, provided that the algorithm tracks all data races. RACEFUZZER, therefore, only performs thread switches before synchronization operations. This particular restriction on thread switch keeps our implementation fast. Since RACEFUZZER only tracks synchronization operations and a racing statement pair, the runtime overhead of RACEFUZZER is significantly lower than that of hybrid-race detection and happens-before race detection techniques.

5. Empirical Evaluation

5.1 Benchmark Programs

We evaluated RACEFUZZER on a variety of Java multithreaded programs. The benchmark includes both closed programs and open libraries that require test drivers to close them. We ran our experiments on a Macbook Pro with a 2.2 GHz Intel Core 2 Duo processor and 2GB RAM. We considered the following closed benchmark programs in our experiments: `moldyn`, `montecarlo`, `raytracer`, three benchmarks from the Java Grande Forum, `cache4j`, a fast thread-safe implementation of a cache for Java objects, `sor`, successive order-relaxation benchmark from ETH [53], `hedc`, a web-crawler application kernel developed at ETH [53], `weblech`, a multi-threaded web site download and mirror tool, `jspider`, a highly configurable and customizable Web Spider engine, `jigsaw 2.2.6`, W3C's leading-edge Web server platform. The total lines of code in these benchmark programs is approximately 600,000. The bugs and real races discovered in the benchmark programs whose column 8 has an empty entry, were previously unknown.

The open programs consist of several synchronized Collection classes provided with Sun's JDK, such as `Vector` in JDK 1.1, `ArrayList`, `LinkedList`, `HashSet`, and `TreeSet` in JDK 1.4.2. Most of these classes (except the `Vector` class) are not synchronized by default. The `java.util` package provides special functions `Collections.synchronizedList` and `Collections.synchronizedSet` to make the above classes synchronized. In order to close the Collection classes, we wrote a multi-threaded test driver for each such class. A test driver starts by creating two empty objects of the class. The test driver also creates and starts a set of threads, where each thread executes different methods of either of the two objects concurrently. We created two objects because some of the methods, such as `containsAll`, takes as an argument an object of the same type. For such methods, we call the method on one object and pass the other object as an argument.

We use our experiments two demonstrate the following **two hypotheses**:

1. RACEFUZZER can create real race conditions with very high probability. It can also show if a real race can lead to an exception.
2. The real races detected automatically by RACEFUZZER are same as the real races that are predicted and manually confirmed for a number of existing benchmark programs.

5.2 Results

Table 1 summarizes the results of our experiments. Column 2 reports the number of lines of code. The reported number of lines of code is always fewer than the actual number of lines of code. This is because we do not count lines in several libraries. Columns 3, 4, and 5 report the average runtime for the benchmark programs using normal execution, the hybrid-race detection algorithm, and RACEFUZZER, respectively. For the I/O intensive benchmarks, the runtime of RACEFUZZER is 1.1x-3x greater than normal execution time. However, the runtime is significantly greater for the high-performance computing applications. The runtime of the hybrid-race detection algorithm has many orders of magnitude higher runtime for the high-performance benchmarks. The runtime for RACEFUZZER is not that high because we only instrument the racing statements and synchronization operations in RACEFUZZER. Since RACEFUZZER is a tool for testing and debugging, we do not worry about runtime as long as the average runtime is less than a few seconds. Due to the interactive nature of the `jigsaw` webserver, we do not report the runtime for `jigsaw`.

Columns 6, 7, and 8 report the number of potential races detected by the hybrid algorithm, the number of real races reported by RACEFUZZER, and the number of real races known from case studies done by other researchers, respectively. In each case, we count the number of distinct pairs of statements for which there is a race. The fact that the numbers in column 7 are equal to the numbers in column 8 demonstrates our hypothesis 2, i.e., RACEFUZZER reports all real races that were reported by existing dynamic analysis tools. In case of `moldyn`, we discovered 2 real races (but benign) that were missed by previous dynamic analysis tools.

Column 9 reports the total number of distinct pairs of racing statements for which an exception has been thrown by a benchmark program. Column 10 reports the number of exceptions thrown by a benchmark when run with the JVM's default scheduler. We describe details of some of the exceptions detected by RACEFUZZER in the next section. The results in these two columns show that RACEFUZZER is far more effective in discovering insidious errors in concurrent programs compared to the default scheduler. Column 11

Program Name	SLOC	Average Runtime in sec.			# of Races			# of Exceptions		Probability of hitting a race
		Normal	Hybrid	RF	Hybrid	RF (real)	known	RF	Simple	
moldyn	1,352	2.07	> 3600	42.37	59	2	0	0	0	1.00
raytracer	1,924	3.25	> 3600	3.81	2	2	2	0	0	1.00
montecarlo	3,619	3.48	> 3600	6.44	5	1	1	0	0	1.00
cache4j	3,897	2.19	4.26	2.61	18	2	-	1	0	1.00
sor	17,689	0.16	0.35	0.23	8	0	0	0	0	-
hedc	29,948	1.10	1.35	1.11	9	1	1	1	0	0.86
weblech	35,175	0.91	1.92	1.36	27	2	1	1	1	0.83
jspider	64,933	4.79	4.88	4.81	29	0	-	0	0	-
jigsaw	381,348	-	-	0.81	547	36	-	0	0	0.90
vector 1.1	709	0.11	0.25	0.2	9	9	9	0	0	0.94
LinkedList	5979	0.16	0.26	0.22	12	12	-	5	0	0.85
ArrayList	5866	0.16	0.26	0.24	14	7	-	7	0	0.55
HashSet	7086	0.16	0.26	0.25	11	11	-	8	1	0.54
TreeSet	7532	0.17	0.26	0.24	13	8	-	8	1	0.41

Table 1. Experimental results.

shows that in most cases RACEFUZZER can create a real race with very high probability. In order to roughly estimate the probability, we ran RACEFUZZER 100 times for each racing pair of statements. The above results demonstrate our hypothesis 1.

5.3 Bugs Found

RACEFUZZER discovered a number of previously unknown uncaught exceptions in the benchmark programs. We next describe a couple of them. RACEFUZZER discovered an uncaught exception in `cache4j` that happens due to a race over the `_sleep` field in `CacheCleaner.java`. The code snippet causing the exception is shown below.

```
Thread1:

synchronized(this){
   if(_sleep){
       interrupt();
   }
}
```

```
Thread2:
_sleep = true;
try {
    sleep(_cleanInterval);
} catch (Throwable t){
} finally {
   _sleep = false;
}
```

If `_sleep` is set to `true` by `Thread2` before entering the try block and `Thread1` is executed next, then an uncaught `InterruptedException` is thrown causing `Thread2` to crash. Note that here `this` corresponds to `Thread2`.

We discovered some concurrency related problems in the JDK 1.4.2 classes `LinkedList`, `ArrayList`, `HashSet`, and `TreeSet`. Specifically, we discovered real races in the `containsAll` and `equals` methods of `LinkedList` and `ArrayList`, and in the `containsAll` and `addAll` methods of `HashSet` and `TreeSet`. For example, if we call `l1.containsAll(l2)` and `l2.removeAll()` in two threads, where `l1` and `l2` are synchronized `LinkedLists` (created using `Collections.synchronizedList`), then we can get both `ConcurrentModificationException` and `NoSuchElementException`. This is because the `containsAll` method is implemented by the superclass `AbstractCollection` and the implementation uses iterator in a thread-unsafe way: a call to `l1.containsAll(l2)` calls the synchronized `iterator` method on `l2` and then goes over the iterator without holding the lock on `l2`. As a result, the iterator accesses the `modCount` field of `l2` without holding the lock on `l2`. Therefore, any other method call on `l2` that modifies `modCount`, such as `removeAll`, would interfere with the iterator code leading to exceptions. The code works without exception in a single-threaded setting and probably the developers had a single-threaded setting in mind while implementing the unsynchronized `containsAll` method in `AbstractCollection`. However, while extending the `LinkedList` class to synchronized `LinkedList` using a decorator pattern in the `Collections` class, the developers did not override the `containsAll` method to make it thread-safe.

6. Related Work

A large body of research focuses on dynamic or static race detection [41, 35, 21, 43, 10, 14]. Type based techniques [20, 5, 6], which require programmer annotations, have been used to reduce the race detection problem to a type checking problem. Since annotation writing creates significant overhead, techniques [3] have been proposed to infer type annotations by looking at concurrent executions. Other language based techniques for static race detection include nesC [24] and Guava [5]. Several static race detection techniques [49, 19, 39] based on lockset [43] have been proposed. An important advantage of the static techniques is that they could find all potential race conditions in a program. A primary limitation of these techniques is that they report a lot of false races. More recent efforts on static race detection [33, 32] have significantly reduced the number of false warnings with minimal annotations, but the problem of false positives still remains. Moreover, these techniques could not infer if a race could lead to an exception in the program. Therefore, manual inspection is needed to separate real races and harmful races. Manual inspection often overwhelms the developers. RACEFUZZER tries to reduce the effort of manual inspection by exploiting the potential race reports generated by any imprecise race detection technique to guide a random thread scheduler.

Dynamic race detection techniques are often based on lockset [43, 53, 10, 36, 2] or on happens-before [44, 14, 1, 11, 30, 42, 13, 34]. Lockset based dynamic techniques could predict data races that did not happen in a concurrent execution; however, such techniques can report many false warnings. Happens-before based dynamic techniques are capable of detecting races that actually happen in an execution. Therefore, these techniques are precise, but they cannot give good coverage as lockset based algorithms. Specifically, happens-before race detectors cannot predict races that could happen on a different schedule or they cannot create a schedule that could reveal a real race. Recently happens-before race detection has been successfully extended to classify harmful races from benign races [34], but they suffer from the same limitations as happens-before techniques. Hybrid techniques [14, 37, 38, 54] combine lockset with happens-before to make dynamic race detection both precise and predictive. Despite the combination, hybrid techniques could report many false warnings. One characteristics that distinguishes RACEFUZZER from other dynamic techniques is that RACEFUZZER *actively* controls the thread scheduler, whereas the other techniques *passively* observe an execution.

Recently, a couple of random testing techniques [18, 50] for concurrent programs have been proposed. These techniques randomly seed a Java program under test with the `sleep()`, the `yield()`, and the `priority()` primitives at shared memory accesses and synchronization events. Although these techniques have successfully detected bugs in many programs, they have two limitations. These techniques are not systematic as the primitives `sleep()`, `yield()`, `priority()` can only advise the scheduler to make a thread switch, but cannot force a thread switch. Second, reproducibility cannot be guaranteed in such systems [50] unless there is builtin support for capture-and-replay [18]. RACEFUZZER removes these limitations by explicitly controlling the scheduler. We recently proposed an effective random testing algorithm, called RAPOS [45], to sample partial orders almost uniformly at random. However, we observed that RAPOS cannot often discover error-prone schedules with high probability because the number of partial orders that can be exhibited by a large concurrent program can be astronomically large. Therefore, we focused on testing "error-prone" schedules, i.e. schedules that exhibit a race condition.

Static verification [4, 16, 28, 40, 8] and model checking [17, 29, 25, 27, 52, 31] or path-sensitive search of the state space is an alternative approach to finding bugs in concurrent programs. Model checkers being exhaustive in nature can often find all concurrency related bugs in concurrent programs. Unfortunately, model checking does not scale with program size. Several other systematic and exhaustive techniques [7, 9, 48, 46] for testing concurrent and parallel programs have been developed recently. These techniques exhaustively explore all interleavings of a concurrent program by systematically switching threads at synchronization points. More recently, efforts [47] have been made to combine model checking with lockset based algorithms to prove the existence of real races; however, this technique suffers from scalability problem as in model checking.

Randomized algorithms for model checking have also been proposed. For example Monte Carlo Model Checking [26] uses random walk on the state space to give probabilistic guarantee of the validity of properties expressed in linear temporal logic. Randomized depth-first search [15] and its parallel extensions have been developed to dramatically improve the cost-effectiveness of state-space search techniques using parallelism.

Capture and replay techniques have been combined with delta-debugging [12] to pinpoint a program location where a thread switch could result in a program failure. The key difference between this technique and RACEFUZZER is that the former technique narrows down the difference between a successful schedule and a failure inducing schedule to pinpoint a bug. RACEFUZZER randomly controls thread schedules based on potential race conditions to determine if a race is real.

Acknowledgment

We would like to thank Ras Bodik, Jacob Burnim, and Shaunak Chatterjee for providing valuable comments on a draft of this paper. This work is supported in part by the NSF Grant CNS-0720906.

References

[1] S. V. Adve, M. D. Hill, B. P. Miller, and R. H. B. Netzer. Detecting data races on weak memory systems. In *18th annual International Symposium on Computer architecture (ISCA)*, pages 234–243. ACM, 1991.

[2] R. Agarwal, A. Sasturkar, L. Wang, and S. D. Stoller. Optimized run-time race detection and atomicity checking using partial discovered types. In *20th IEEE/ACM international Conference on Automated software engineering (ASE)*, pages 233–242. ACM, 2005.

[3] R. Agarwal and S. D. Stoller. Type inference for parameterized race-free java. In *Verification, Model Checking, and Abstract Interpretation, 5th International Conference (VMCAI)*, pages 149–160, 2004.

[4] A. Aiken and D. Gay. Barrier inference. In *25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 342–354. ACM, 1998.

[5] D. F. Bacon, R. E. Strom, and A. Tarafdar. Guava: a dialect of java without data races. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'00)*, pages 382–400, 2000.

[6] C. Boyapati and M. C. Rinard. A parameterized type system for race-free java programs. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'01)*, pages 56–69, 2001.

[7] D. Bruening. Systematic testing of multithreaded Java programs. Master's thesis, MIT, 1999.

[8] S. Burckhardt, R. Alur, and M. M. K. Martin. Checkfence: checking consistency of concurrent data types on relaxed memory models. In *CM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI)*, pages 12–21, 2007.

[9] R. H. Carver and Y. Lei. A general model for reachability testing of concurrent programs. In *6th International Conference on Formal Engineering Methods (ICFEM'04)*, volume 3308 of *LNCS*, pages 76–98, 2004.

[10] J. D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proc. of the ACM SIGPLAN Conference on Programming language design and implementation*, pages 258–269, 2002.

[11] J.-D. Choi, B. P. Miller, and R. H. B. Netzer. Techniques for debugging parallel programs with flowback analysis. *ACM Trans. Program. Lang. Syst.*, 13(4):491–530, 1991.

[12] J.-D. Choi and A. Zeller. Isolating failure-inducing thread schedules. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 210–220. ACM, 2002.

[13] M. Christiaens and K. D. Bosschere. Trade, a topological approach to on-the-fly race detection in java programs. In *JavaTM Virtual Machine Research and Technology Symposium (JVM)*, pages 15–15. USENIX Association, 2001.

[14] A. Dinning and E. Schonberg. Detecting access anomalies in programs with critical sections. In *Proc. of the ACM/ONR Workshop on Parallel and Distributed Debugging*, 1991.

[15] M. B. Dwyer, S. Elbaum, S. Person, and R. Purandare. Parallel randomized state-space search. In *29th International Conference on Software Engineering (ICSE)*, pages 3–12. IEEE, 2007.

[16] M. B. Dwyer, J. Hatcliff, Robby, and V. P. Ranganath. Exploiting object escape and locking information in partial-order reductions for concurrent object-oriented programs. *Form. Methods Syst. Des.*, 25(2–3):199–240, 2004.

[17] J. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT Press, 1999.

[18] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, , and S. Ur. Multithreaded Java program test generation. *IBM Systems Journal*, 41(1):111–125, 2002.

[19] D. R. Engler and K. Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *19th ACM Symposium on Operating Systems Principles (SOSP)*, pages 237–252, 2003.

[20] C. Flanagan and S. N. Freund. Type-based race detection for java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'00)*, pages 219–232, 2000.

[21] C. Flanagan and S. N. Freund. Detecting race conditions in large programs. In *Proc. of the Program Analysis for Software Tools and Engineering Conference*, 2001.

[22] C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 256–267, 2004.

[23] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 338–349, 2003.

[24] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *ACM SIGPLAN Conference on Programming language design and implementation*, pages 1–11, 2003.

[25] P. Godefroid. Model checking for programming languages using verisoft. In *24th Symposium on Principles of Programming Languages*, pages 174–186, 1997.

[26] R. Grosu and S. A. Smolka. Monte carlo model checking. In *11th International Conference Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, volume 3440 of *LNCS*, pages 271–286, 2005.

[27] K. Havelund and T. Pressburger. Model Checking Java Programs using Java PathFinder. *Int. Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[28] T. A. Henzinger, R. Jhala, and R. Majumdar. Race checking by context inference. *SIGPLAN Not.*, 39(6):1–13, 2004.

[29] G. Holzmann. The Spin model checker. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[30] J. Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *ACM/IEEE conference on Supercomputing*, pages 24–33. ACM, 1991.

[31] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *ACM Symposium on Programming Language Design and Implementation (PLDI'07)*, 2007.

[32] M. Naik and A. Aiken. Conditional must not aliasing for static race detection. In *34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 327–338, 2007.

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

[34] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and

B. Calder. Automatically classifying benign and harmful data races using replay analysis. In *ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI)*, pages 22–31, 2007.

[35] R. Netzer and B. Miller. Detecting data races in parallel program executions. In *Advances in Languages and Compilers for Parallel Computing*. MIT Press, 1990.

[36] H. Nishiyama. Detecting data races using dynamic escape analysis based on read barrier. In *Virtual Machine Research and Technology Symposium*, pages 127–138, 2004.

[37] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 167–178. ACM, 2003.

[38] E. Pozniansky and A. Schuster. Efficient on-the-fly data race detection in multithreaded c++ programs. In *Ninth ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP)*, pages 179–190. ACM, 2003.

[39] P. Pratikakis, J. S. Foster, and M. Hicks. LOCKSMITH: context-sensitive correlation analysis for race detection. In *ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, pages 320–331. ACM, 2006.

[40] S. Qadeer and D. Wu. Kiss: keep it simple and sequential. In *ACM SIGPLAN 2004 conference on Programming language design and implementation (PLDI)*, pages 14–24. ACM, 2004.

[41] B. Richards and J. R. Larus. Protocol-based data-race detection. In *Proc. of the SIGMETRICS symposium on Parallel and distributed tools*, pages 40–47, 1998.

[42] M. Ronsse and K. D. Bosschere. Recplay: a fully integrated practical record/replay system. *ACM Trans. Comput. Syst.*, 17(2):133–152, 1999.

[43] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.

[44] E. Schonberg. On-the-fly detection of access anomalies. In *ACM SIGPLAN '89 Conference on Programming Language Design and Implementation (PLDI)*, volume 24, pages 285–297, 1989.

[45] K. Sen. Effective random testing of concurrent programs. In *22nd IEEE/ACM nternational Conference on Automated Software Engineering (ASE'07)*, 2007.

[46] K. Sen and G. Agha. A race-detection and flipping algorithm for automated testing of multi-threaded programs. In *Haifa verification conference 2006 (HVC'06)*, Lecture Notes in Computer Science. Springer, 2006.

[47] O. Shacham, M. Sagiv, and A. Schuster. Scaling model checking of dataraces using dynamic information. *J. Parallel Distrib. Comput.*, 67(5):536–550, 2007.

[48] S. F. Siegel, A. Mironova, G. S. Avrunin, and L. A. Clarke. Using model checking with symbolic execution to verify parallel numerical programs. In *International symposium on Software testing and analysis (ISSTA)*, pages 157–168. ACM Press, 2006.

[49] N. Sterling. Warlock: A static data race analysis tool. In *USENIX Winter Technical Conference*, pages 97–106, 1993.

[50] S. D. Stoller. Testing concurrent Java programs using randomized scheduling. In *Workshop on Runtime Verification (RV'02)*, volume 70 of *ENTCS*, 2002.

[51] M. Vaziri, F. Tip, and J. Dolby. Associating synchronization constraints with data in an object-oriented language. In *33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 334–345, 2006.

[52] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *15th International Conference on Automated Software Engineering (ASE)*. IEEE, 2000.

[53] C. von Praun and T. R. Gross. Object race detection. In *16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications (OOPSLA)*, pages 70–82. ACM, 2001.

[54] Y. Yu, T. Rodeheffer, and W. Chen. Racetrack: efficient detection of data race conditions via adaptive tracking. *SIGOPS Oper. Syst. Rev.*, 39(5):221–234, 2005.