Chapter 10

Par Lab Progress on Correctness Tools for Parallel Programs

Jacob Burnim, Tayfun Elmas, Chang-Seo Park, and Koushik Sen

1 Introduction

The spread of multicore and manycore processors and graphics processing units (GPUs) has greatly increased the need for parallel correctness tools. Reasoning about multithreaded programs is significantly more difficult than for sequential programs due to the nondeterministic interleaving of parallel threads. As part of the Par Lab project, we proposed to develop practical tools for verifying, testing, and debugging parallel programs. At the beginning of the project, we focused on developing fully automated tools for finding parallel bugs, such as data races, deadlocks, and atomicity violations. Our effort was in unison with contemporary research on analysis of parallel and concurrent programs. After a couple of year’s effort, we started realizing that verification and testing of parallel programs is a hard problem and we need radically new ideas to tackle it. While working on a novel mechanism for specifying the deterministic behavior of parallel programs, we discovered that an elegant way to tackle the complexity is to find ways to separately specify, test, and reason about the correctness of a program’s use of parallelism and a program’s functional correctness. Driven by this belief, we developed two fundamental techniques for separating the parallelization correctness aspect of a program from the program’s sequential functional correctness. Towards the end of the project, we started realizing that it is unwise to fully automate testing and to keep programmers out of the loop when programmers can provide valuable insights on ‘what’ and ‘how’ to test. In the final year of the Par Lab project, we came up with a practical domain-specific language, called CONCURRIT, using which programmers can accelerate automated testing by providing useful insights about the program under test. CONCURRIT not only gives programmers full control over the testing and debugging process of parallel software, but it also incorporates the power of various automated testing techniques that we had previously developed. We next briefly introduce the three key projects that we have successfully pursued in the past 5 years.

2 Specifying Correctness of Parallel Programs

We developed two techniques for separating the parallelization correctness from the functional correctness of a parallel program. Our first technique is based on the observation that a key simplifying assumption available to the developer of sequential code is the determinism of the execution model. In contrast, parallel programs exhibit non-deterministic behavior due to the interleaving of parallel threads. This nondeterminism is essential to harness the power of parallel chips. But programmers often strive to preserve the determinism of their applications in the face of this nondeterminism—that is, to always compute the same output for the same input, no matter how the parallel threads...
of execution are scheduled. We argue that programmers should be provided with a framework that will allow them to express the natural, intended deterministic behavior of parallel programs directly and easily. Thus, we proposed extending programming languages with constructs for writing specifications, called bridge assertions, that focus on relating outcomes of two parallel executions differing only in thread-interleavings [6, 5, 7, 4]. We evaluated the utility of these assertions by manually adding deterministic specifications to a number of parallel Java benchmarks. We found it to be fairly easy to specify the correct deterministic behavior of the benchmark programs using our assertions, despite being unable in most cases to write traditional invariants or functional correctness assertions. Our papers on this work won an ACM SIGSOFT Distinguished Paper Award at [5] and the 2010 IFIP TC2 Manfred Paul Award for Excellence in Software: Theory and Practice at [7]. Our research has been featured in the CACM’s Research Highlights [6]. Our first paper on correctness is based on our CACM article.

Our second technique, in the same spirit of decomposing our correctness efforts, is suggested by the observation that manually parallelized programs often employ algorithms that are different from the sequential versions. In some cases, the algorithm itself is nondeterministic. Examples of such nondeterministic algorithms include branch-and-bound algorithms, which may produce multiple valid outputs for the same input. Such algorithmic nondeterminism is often tightly coupled with the functional correctness of the code, thus violating the premise of our determinism checking technique. To address these situations, we proposed to separate the reasoning about the algorithmic and the scheduler sources of nondeterminism. For such cases, we provide a specification technique in which a nondeterministic sequential (NDSEQ) version of a parallel program is the specification for the program’s use of parallelism [3, 1]. Using this technique, the programmer expresses the intended, algorithmic nondeterminism in his or her program—and the program’s parallelism is correct if the nondeterministic thread scheduling adds no additional program behaviors to those of the NDSEQ program. Testing, debugging, and reasoning about functional correctness can then be performed on the NDSEQ version, with controlled nondeterminism but with no interleaving of parallel threads. We developed an automatic technique [1] to test parallelism correctness, by checking that an observed parallel execution obeys its nondeterministic sequential specification.

### 3 Active Testing of Parallel and Concurrent Programs

To automatically check parallel correctness, we have developed active testing, a new scalable automated method for testing and debugging parallel and concurrent programs. Active testing combines the power of imprecise program analysis with the precision of software testing to quickly discover parallel and concurrent bugs and to reproduce discovered bugs on demand [19, 17, 14, 12, 15, 13, 11, 10, 8, 9, 16]. The key idea behind active testing is to control the thread scheduler in order to force the program into a state to expose a parallel bug—data race, deadlock, atomicity violation, or violation of sequential memory consistency. The technique starts with lightweight inexpensive dynamic analysis that identifies situations where there is suspicion that a parallel bug may exist. This first part of the analysis is imprecise because it trades-off precision for efficiency and it tries to increase the coverage of analysis by trying to predict potential bugs in other executions by analyzing a single execution. In the second step, a directed tester executes the program under a controlled thread schedule in an attempt to bring the program in the buggy state. If it succeeds, it has identified a real parallel bug; that is, the error report is guaranteed not be a false alarm, which is a serious problem with existing dynamic analyses. The actual method of controlling the thread schedule works as follows: once a thread reaches a state that resembles the desired state, it is paused as long as possible, giving a chance for other threads to catch up and complete the candidate buggy scenario.

We have implemented active testing in extensible and publicly available tools for Java, C/PThreads, and UPC, and have applied these tools to find many previously known and unknown parallel bugs in a number of programs, including several real-world applications containing more than 600K lines of code. Our active testing tool for UPC, called UPC-Thrille [16, 18], is the first implementation of an efficient and scalable data race detector for distributed memory programs that tracks both memory references and communication operations. On widely used benchmark programs, UPC-Thrille found previously unknown data races with at most 50% overhead when running on 2048 cores of a CrayXE6 system.

Active testing is an effective and practical technique that exposes a simple testing usage model to programmers while using sophisticated program analysis under the hood. A paper on this work won an ACM SIGSOFT Distinguished Paper Award at [15]. Our second paper in this book is based on the first paper on active testing [19].
4 A Testing Framework for Parallel Programs

While developing various push-button testing techniques for parallel and concurrent programs, we realized that testing could be made more targeted and efficient if we could somehow involve the programmer in the process. We have developed CONCURRIT [2], a domain-specific language (DSL) that can be used by programmers to write high-level tests for multithreaded programs. Using CONCURRIT, a programmer can write CONCURRIT scripts to formally and concisely specify a set of thread schedules to explore in order to find a schedule exhibiting the bug. Further, the programmer can specify how these thread schedules should be searched to find a schedule that reproduces the bug. CONCURRIT is comparable with various xUnit testing frameworks such as JUnit and NUnit; however, unlike existing xUnit frameworks, CONCURRIT targets parallel and concurrent programs, incorporating automated techniques for searching potentially buggy thread schedules specified by the programmer. We have implemented CONCURRIT as an embedded DSL in C++, which uses manual or automatic source instrumentation to partially control the scheduling of the software under test. Using CONCURRIT, we were able to write concise tests to reproduce parallel bugs in a variety of benchmarks, including the Mozilla’s SpiderMonkey JavaScript engine, Memcached, Apache’s HTTP server, and MySQL.

Bibliography


Abstract

The trend towards processors with more and more parallel cores is increasing the need for software that can take advantage of parallelism. The most widespread method for writing parallel software is to use explicit threads. Writing correct multithreaded programs, however, has proven to be quite challenging in practice. The key difficulty is non-determinism. The threads of a parallel application may be interleaved non-deterministically during execution. In a buggy program, non-deterministic scheduling will lead to non-deterministic results—some interleavings will produce the correct result while others will not.

We propose an assertion framework for specifying that regions of a parallel program behave deterministically despite non-deterministic thread interleaving. Our framework allows programmers to write assertions involving pairs of program states arising from different parallel schedules. We describe an implementation of our deterministic assertions as a library for Java, and evaluate the utility of our specifications on a number of parallel Java benchmarks. We found specifying deterministic behavior to be quite simple using our assertions. Further, in experiments with our assertions, we were able to identify two races as true parallelism errors that lead to incorrect non-deterministic behavior. These races were distinguished from a number of benign races in the benchmarks.

1 Introduction

The semiconductor industry has hit the power wall—performance of general-purpose single-core microprocessors can no longer be increased due to power constraints. Therefore, to continue to increase performance, the microprocessor industry is instead increasing the number of processing cores per die. The new “Moore’s Law” is that the number of cores will double every generation, with individual cores going no faster [6].

This new trend of increasingly parallel chips has made it clear that we have to write parallel code in order to run software efficiently. Unfortunately, parallel software is more difficult to write and debug than its sequential counterpart. A key reason for this difficulty is that parallel programs can show different behaviors depending on how the executions of their parallel threads interleave.

The fact that executions of parallel threads can interleave with each other in arbitrary fashion to produce different outputs is called internal non-determinism or scheduler non-determinism. Internal non-determinism is essential to make parallel threads execute simultaneously and to harness the power of parallel chips. However, most of the sequential programs that we write are deterministic—they produce the same output on the same input. Therefore, in order to make parallel programs easy to understand and debug, we need to make them behave like sequential programs, i.e. we need to make parallel programs deterministic.
A number of ongoing research efforts aim to make parallel programs deterministic by construction. These efforts include the design of new parallel programming paradigms \cite{47, 32, 28, 3, 31} and the design of new type systems and annotations that could retrofit existing parallel languages \cite{4, 9}. But such efforts face two key challenges. First, new languages see slow adoption and often remain specific to limited domains. Second, new paradigms often include restrictions that can hinder general purpose programming. For example, a key problem with new type systems is that they can make programming more difficult and restrictive.

The most widespread method for writing parallel programs, threads \cite{26, 8, 12, 27}, requires programmers to ensure determinism. To aid programmers in writing deterministic programs, a number of tools and techniques have been developed. These tools attempt to automatically find sources of non-determinism likely to be harmful (i.e. to lead to non-deterministic output) such as data races and high-level race conditions. A large body of work spanning over 30 years has focused on data race detection. A data race occurs when two threads concurrently access a memory location and at least one of the accesses is a write. Both dynamic \cite{13, 1, 41, 49, 11, 2, 43} and static \cite{45, 17, 10, 18, 24, 16, 38, 34} techniques have been developed to detect and predict data races in multi-threaded programs. Although the work on data race detection has significantly helped in finding determinism bugs in parallel programs, it has been observed that (1) the absence of data races is not sufficient to ensure determinism \cite{5, 22, 19}, and (2) data races do not always cause non-deterministic results. In fact, race conditions are often useful in gaining performance, while still ensuring high-level deterministic behavior \cite{7}.

We argue that programmers should be provided with a framework that will allow them to express deterministic behaviors of parallel programs directly and easily. Specifically, we should provide an assertion framework where programmers can directly and precisely express the necessary deterministic behavior. On the other hand, the framework should be flexible enough so that deterministic behaviors can be expressed more easily than with a traditional assertion framework. For example, when expressing the deterministic behavior of a parallel edge detection algorithm for images, we should not have to rephrase the problem as a race detection problem; neither should we have to write a state assertion that relates the output to the input, which would be complex and time-consuming. Rather, we should simply be able to say that, if the program is executed on the same image, then the output image remains the same regardless of how the program’s parallel threads are scheduled.

In this paper, we propose such a framework for asserting that blocks of parallel code behave deterministically. Formally, our framework allows a programmer to give a specification for a block $P$ of parallel code as:

\begin{verbatim}
  deterministic assume(Pre($s_0, s'_0$)) {
    $P$
  } assert(Post($s, s'$))
\end{verbatim}

This specification asserts the following: Suppose $P$ is executed twice with potentially different schedules, once from initial state $s_0$ and once from $s'_0$ and yielding final states $s_1$ and $s'_1$, respectively. Then, if the user-specified pre-condition $Pre$ holds over $s_0$ and $s'_0$, then $s$ and $s'$ must satisfy the user-specified post-condition $Post$.

For example, we could specify the deterministic behavior of a parallel matrix multiply with:

\begin{verbatim}
  deterministic assume($|A - A'| < 10^{-6}$ and $|B - B'| < 10^{-6}$) {
    C = parallel_matrix_multiply_float(A, B);
  } assert($|C - C'| < 10^{-6}$);
\end{verbatim}

Note the use of primed variables $A'$, $B'$, and $C'$ in the above example. These variables represent the state of the matrices $A$, $B$, and $C$ from a different execution. As such the predicates that we write inside assume and assert are different from state predicates written in a traditional assertion framework—these special predicates relate a pair of states from different executions. We call such a predicate a bridge predicate and an assertion using bridge predicates a bridge assertion. A key contribution of this paper is the introduction of these bridge predicates and bridge assertions. We believe that these novel predicates can be used not only for deterministic specification, but also be used for other purposes such as writing regression tests.

Our deterministic assertions provide a way to specify the correctness of the parallelism in a program independently of the program’s traditional functional correctness. By checking whether different program schedules can non-deterministically lead to semantically different answers, we can find bugs in a program’s use of parallelism even when unable to directly check functional correctness—i.e. that the program’s output is correct given its input. Inversely, by checking that a parallel program behaves deterministically, we can gain confidence in the correctness of its use of parallelism independently of whatever method we use to gain confidence in the program’s functional correctness.
We have implemented our deterministic assertions as a library for the Java programming language. We evaluated the utility of these assertions by manually adding deterministic specifications to a number of parallel Java benchmarks. We used an existing tool to find executions exhibiting data and higher-level races in these benchmarks and used our deterministic assertions to distinguish between harmful and benign races. We found it to be fairly easy to specify the correct deterministic behavior of the benchmark programs using our assertions, despite being unable in most cases to write traditional invariants or functional correctness assertions. Further, our deterministic assertions successfully distinguished the two known harmful races in the benchmarks from the benign races.

2 Deterministic Specification

In this section, we motivate and define our proposal for assertions for specifying determinism.

Strictly speaking, a block of parallel code is said to be deterministic if, given any particular initial state, all executions of the code from the initial state produce the exact same final state. In our specification framework, the programmer can specify that they expect a block of parallel code, say $P$, to be deterministic with the following construct:

\[
\text{deterministic} \{ \\
  P \\
\}
\]

This assertion specifies that if $s$ and $s'$ are both program states resulting from executing $P$ under different thread schedules from some initial state $s_0$, then $s$ and $s'$ must be equal. For example, the specification:

\[
\text{deterministic} \{ \\
  C = \text{parallel_matrix_multiply_int}(A, B); \\
\}
\]

asserts that for the parallel implementation of matrix multiplication (defined by function $\text{parallel_matrix_multiply_int}$), any two executions from the same program state must reach the same program state—i.e. with identical entries in matrix $C$—no matter how the parallel threads are scheduled.

A key implication of knowing that a block of parallel code is deterministic is that we may be able to treat the block as sequential in other contexts. That is, although the block may have internal parallelism, a programmer (or perhaps a tool) can hopefully ignore this parallelism when considering the larger program using the code block. For example, perhaps a deterministic block of parallel code in a function can be treated as if it were a sequential implementation when reasoning about the correctness of code calling the function.

Semantic Determinism The above deterministic specification is often too conservative. For example, consider a similar example, but where $A$, $B$, and $C$ are floating-point matrices:

\[
\text{deterministic} \{ \\
  C = \text{parallel_matrix_multiply_float}(A, B); \\
\}
\]

asserts that for the parallel implementation of matrix multiplication (defined by function $\text{parallel_matrix_multiply_float}$), any two executions from the same program state must reach the same program state—i.e. with identical entries in matrix $C$—no matter how the parallel threads are scheduled.

A key implication of knowing that a block of parallel code is deterministic is that we may be able to treat the block as sequential in other contexts. That is, although the block may have internal parallelism, a programmer (or perhaps a tool) can hopefully ignore this parallelism when considering the larger program using the code block. For example, perhaps a deterministic block of parallel code in a function can be treated as if it were a sequential implementation when reasoning about the correctness of code calling the function.

In many programming languages, floating-point addition and multiplication are not associative due to rounding error. Thus, depending on the implementation, it may be unavoidable that the entries of matrix $C$ will differ slightly depending on the thread schedule.

In order to tolerate such differences, we must relax the deterministic specification:

\[
\text{deterministic} \{ \\
  C = \text{parallel_matrix_multiply_float}(A, B); \\
  \text{assert}(|C - C'| < 10^{-6}); \\
\}
\]

This assertion specifies that, for any two matrices $C$ and $C'$ resulting from the execution of the matrix multiply from same initial state, the entries of $C$ and $C'$ must differ by only a small quantity (i.e. $10^{-6}$).

Note that the above specification contains a predicate over two states—each from a different parallel execution of deterministic block. We call such a predicate a bridge predicate, and an assertion using a bridge predicate a bridge...
assertion. Bridge assertions are different from traditional assertions in that they allow one to write a property over two program states coming from different executions whereas traditional assertions only allow us to write a property over a single program state.

Note also that such predicates need not be equivalence relations on pairs of states. In particular, the approximate equality used above is not an equivalence relation.

This relaxed notion of determinism is useful in many contexts. Consider the following example which adds in parallel two items to a synchronized set:

```java
Set set = new SynchronizedTreeSet();
deterministic {
    cobegin
    set.add(3);
    set.add(5);
    coend
} assert(set.equals(set'));
```

If set is represented internally as a red-black tree, then a strict deterministic assertion would be too conservative. The structure of the resulting tree, and its layout in memory, will likely differ depending on which element is inserted first, and thus the different executions can yield different program states.

But we can use a bridge predicate to assert that, no matter what schedule is taken, the resulting set is semantically equal. That is, for objects set and set\' computed by two different schedules, the equals method must return true because the sets must logically contain the same elements. We call this semantic determinism.

Preconditions for Determinism  So far we have described the following construct:

```java
deterministic {
    P
} assert(Post);
```

where Post is a predicate over two program states in different executions resulting from different thread schedules. That is, if s and s\' are two states resulting from any two executions of P from the same initial state, then Post(s, s\') holds.

The above construct could be rewritten in the following way:

```java
deterministic assume(s0 == s0\') {
    P
} assert(Post);
```

That is, if any two executions of P start from initial states s0 and s0\', respectively, and if s and s\' are the resultant final states, then s0 == s0\' implies that Post(s, s\') holds. The above rewritten specification suggests that we can further relax the requirement of s0 == s0\' by replacing it with a bridge predicate Pre(s0, s0\'}). For example:

```java
deterministic assume(set.equals(set')) {
    cobegin
    set.add(3);
    set.add(5);
    coend
} assert(set.equals(set'));
```

The above specification states that if any two executions start from sets containing the same elements, then after the execution of the code, the resulting sets after the two executions must still contain exactly the same elements.

---

1 Note that in the above construct we do not refer to the final states s and s', but we make them implicit by assuming that Post maps a pair of program states to a Boolean value.
Asserting and Checking Determinism for Multithreaded Programs

Comparison to Traditional Assertions  In summary, we propose the following construct for the specification of deterministic behavior.

```plaintext
deterministic assume(Pre) {
    P
} assert(Post);
```

Formally, it states that for any two program states $s_0$ and $s'_0$, if

- $\text{Pre}(s_0, s'_0)$ holds
- an execution of $P$ from $s_0$ terminates and results in state $s$
- an execution of $P$ from $s'_0$ terminates and results in state $s'$

then $\text{Post}(s, s')$ must hold.

Note that the use of bridge predicates $\text{Pre}$ and $\text{Post}$ has the same flavor as pre and post-conditions used for functions in program verification. However, unlike traditional pre and post-conditions, the proposed $\text{Pre}$ and $\text{Post}$ predicates relate pairs of states from two different executions. In traditional verification, a pre-condition is usually written as a predicate over a single program state, and a post-condition is usually written over two states—the states at the beginning and end of the function. For example:

```plaintext
foo() {
    assume(x > 0);
    old_x = x;
    x = x * x;
    assert(x == old_x*old_x);
}
```

The key difference between a post-condition and a $\text{Post}$ predicate is that a post-condition relates two states at different times along a same execution, whereas a $\text{Post}$ predicate relates two program states in different executions.

Advantages of Deterministic Assertions  Our deterministic specifications are a middle ground between the implicit specification used in race detection—that programs should be free of data races—and the full specification of functional correctness. It is a great feature of data race detectors that typically no programmer specification is needed. However, manually determining which reported races are benign and which are bugs can be time-consuming and difficult. We believe our deterministic assertions, while requiring little effort to write, can greatly aid in distinguishing harmful from benign data races (or higher-level races).

One could argue that a deterministic specification framework is unnecessary given that we can write the functional correctness of a block of code using traditional pre- and post-conditions. For example, one could write the following to specify the correct behavior of $\text{parallel_matrix_multiply_int}$:

```plaintext
C = parallel_matrix_multiply_int(A, B);
assert(C == A \times B);
```

We agree that if one can write a functional specification of a block of code, then there is no need to write deterministic specification, as functional correctness implies deterministic behavior.

The advantage of our deterministic assertions, however, are that they provide a way to specify the correctness of just the use of parallelism in a program, independent of the program’s full functional correctness. In many situations, writing a full specification of functional correctness is difficult and time consuming. But, a simple deterministic specification enables us to use automated technique to check for parallelism bugs, such as harmful data races causing semantically non-deterministic behavior.

Consider a parallel function $\text{parallel_edge_detection}$ that takes an image as input and returns an image where detected edges have been marked. Relating the output to the input image with traditional pre- and post-conditions would likely be quite challenging. However, it is simple to specify that the routine does not have any parallelism bugs causing a correct image to be returned for some thread schedules and an incorrect image for others:
The Berkeley Par Lab: Progress in the Parallel Computing Landscape

```java
deterministic assume(img.equals(img')) {
    result = parallel_edge_detection(img);
} assert(result.equals(result'));
```

where `img.equals(img')` returns true iff the two images are pixel-by-pixel equal.

For this example, a programmer could gain some confidence in the correctness of the routine by writing unit tests or manually examining the output for a handful of images. He or she could then use automated testing or model checking to separately check that the parallel routine behaves deterministically on a variety of inputs, gaining confidence that the code is free from concurrency bugs.

We believe that it is often difficult to come up with effective functional correctness assertions. However, it is often quite easy to use bridge assertions to specify deterministic behavior, enabling a programmer to check for harmful concurrency bugs. In the Evaluation section, we provide several case studies to support this argument.

## 3 Checking Determinism

There may be many potential approaches to checking or verifying a deterministic specification, from testing to model checking to automated theorem proving. In this section, we propose a simple and incomplete method for checking deterministic specifications at run-time.

The key idea of the method is that, whenever a deterministic block is encountered at run-time, we can record the program states \( s_{\text{pre}} \) and \( s_{\text{post}} \) at the beginning and end of the block. Then, given a collection of \((s_{\text{pre}}, s_{\text{post}})\) pairs for a particular deterministic block in some program, we can check a deterministic specification, albeit incompletely, by comparing pairwise the pairs of initial and final states for the block. That is, for a deterministic block:

```java
deterministic assume(Pre) {
    P
} assert(Post);
```

with pre- and post-predicates \( \text{Pre} \) and \( \text{Post} \), we check for every recorded pair of pairs \((s_{\text{pre}}, s'_{\text{pre}})\) and \((s_{\text{post}}, s'_{\text{post}})\) that:

\[
\text{Pre}(s_{\text{pre}}, s'_{\text{pre}}) \implies \text{Post}(s_{\text{post}}, s'_{\text{post}})
\]

If this condition does not hold for some pair, then we report a determinism violation.

To increase the effectiveness of this checking, we must record pairs of initial and final states for deterministic blocks executed under a wide variety of possible thread interleavings. Thus, in practice we likely want to combine our deterministic assertion checking with existing techniques and tools for exploring parallel schedules of a program, such as noise making [14, 46], active random scheduling [42, 43], or model checking [48].

In practice, the cost of recording and storing entire program states could be prohibitive. However, real determinism predicates often depend on just a small portion of the whole program state. Thus, we need only to record and store small projections of program states. For example, for a deterministic specification with pre- and post-predicate \( \text{set.equals(set')} \) we need only to save object set and its elements (possibly also the memory reachable from these objects), rather than the entire program memory.

## 4 Determinism Checking Library

In this section, we describe the design and implementation of an assertion library for specifying and checking determinism of Java programs.

Note that, while it might be preferable to introduce a new syntactic construct for specifying determinism, we instead provide the functionality as a library for simplicity of the implementation.

### 4.1 Overview

Figure 1 shows the core API for our deterministic assertion library. Functions `open` and `close` specify the beginning and end of a deterministic block. Deterministic blocks may be nested, and each block may contain multiple calls to functions `assume` and `assert`, which are used to specify the pre- and post-predicates of deterministic behavior.
class Deterministic {
    static void open()
    static void close()
    static void assume(Object o, Predicate p)
    static void assert(Object o, Predicate p)

    interface Predicate {
        boolean apply(Object a, Object b)
    }
}

Figure 1: Core deterministic specification API.

Each call assume(o, pre) in a deterministic block specifies part of the pre-predicate by giving some projection o of the program state and a predicate pre. That is, it specifies that one condition for any execution of the block to compute an equivalent, deterministic result is that pre.apply(o, o') return true for object o' from the other execution.

Similarly, a call assert(o, post) in a deterministic block specifies that, for any execution satisfying every assume, predicate post.apply(o, o') must return true for object o' from the other execution.

At run-time, our library records every object (i.e. state projection) passed to each assert and assume in each deterministic block, persisting them to some central location. We require that all objects passed as state projections implement the Serializable interface to facilitate this recording. (In practice, this does not seem to be a heavy burden. Most core objects in the Java standard library are serializable, including numbers, strings, arrays, lists, sets, and maps/hashtables.)

Then, also at run-time, a call to assert(o, post) checks post on o and all o' saved from previous, matching executions of the same deterministic block. If the post-predicate does not hold for any of these executions, a determinism violation is immediately reported. Deterministic blocks can contain many assert’s so that determinism bugs can be caught as early as possible and can be more easily localized.

For flexibility, programmers are free to write state projections and predicates using the full Java language. However, it is a programmer’s responsibility to ensure that these predicates contain no observable side effects, as there are no guarantees as to how many times such a predicate may be evaluated in any particular run.

So that the library is easy to use, it tracks which threads are in which deterministic blocks. Thus, a call to assume, assert, or close is automatically associated with the correct enclosing block, even when called from a spawned, child thread. The only restriction on the location of these calls is that every assume call in a deterministic block must occur before any assert.

Built-in Predicates For programmer convenience, we provide two built-in predicates that are often sufficient for specifying pre- and post-predicates for determinism. The first, Equals, returns true if the given objects are equal using their built-in equals method—that is, if o.equals(o'). For many Java objects, this method checks semantic equality—e.g. for integers, floating-point numbers, strings, lists, sets, etc. Further, for single- or multi-dimensional arrays (which do not implement such an equals method), the Equals predicate compares corresponding elements using their equals methods. Figure 2 gives an example with assert and assume using this Equals predicate.

The second predicate, ApproxEquals, checks if two floating-point numbers, or the corresponding elements of two floating-point arrays, are within a given margin of each other. As shown in Figure 3, we found this predicate useful in specifying the deterministic behavior of numerical applications, where it is unavoidable that the low-order bits may vary with different thread interleavings.

4.2 Concrete Example: mandelbrot

Figure 2 shows the deterministic assertions we added to one of our benchmarks, a program for rendering images of the Mandelbrot Set fractal from the Parallel Java Library [30].
main(String args[]) {
    // Read parameters from command-line.
    ...

    // Pre-predicate: equal parameters.
    Predicate equals = new Equals();
    Deterministic.open();
    Deterministic.assume(width, equals);
    Deterministic.assume(height, equals);
    ...
    Deterministic.assume(gamma, equals);

    // spawn threads to compute fractal
    int matrix[][] = ...;
    ...

    Deterministic.assert(matrix, equals);
    Deterministic.close();

    // write fractal image to file
    ...
}

Figure 2: Deterministic assertions for a Mandelbrot Set implementation from the Parallel Java Library [30].

The benchmark first reads a number of integer and floating-point parameters from the command-line. It then spawns several worker threads, which each compute the hues for different segments of the final image, storing them in shared array `matrix`. After waiting for all of the worker threads to finish, the program encodes and writes the image to a file given as a command-line argument.

To add determinism annotations to this program, we simply opened a deterministic block just before the worker threads are spawned and closed it just after they are joined. At the beginning of this block, we added an `assume` call for each of the seven fractal parameters, such as the image size and color palette. At the end of the block, we assert that the resulting array `matrix` should be deterministic, however the worker threads are interleaved.

Note that it would be quite difficult to add assertions for the functional correctness of this benchmark, as each pixel of the resulting image is a complex function of the inputs (i.e. the rate at which a particular complex sequence diverges). Further, there do not seem to be any simple traditional invariants on the program state or outputs which would help identify a parallelism bug.

### 4.3 Implementation

Due to the simple design, we were able to implement this deterministic assertion library in only a few hundred lines of Java code. We use the Java `InheritableThreadLocal` class to track which threads are in which deterministic blocks (and so that spawned child threads inherit the enclosing deterministic block from their parent).

Currently, pairs of initial and final states for the deterministic blocks of an application are just recorded in a single file in the application’s working directory. Blocks are uniquely identified by their location in an application’s source (accessible through, e.g., a stack trace). When a determinism violation is detected, a message is printed and the application is halted.

### 5 Evaluation

In this section, we describe our efforts to validate two claims about our proposal for specifying and checking deterministic parallel program execution:
main(String args[]) {
   ..
   // Pre-predicate: equal parameters.
   Deterministic.open();
   Predicate equals = new Equals();
   Deterministic.assume(mm, equals);
   Deterministic.assume(PARTSIZE, equals);

   // spawn worker threads
   double ek[] = ...;
   double epot[] = ...;
   double vir[] = ...;
   ..

   // Deterministic final energies.
   Predicate apx = new ApproxEquals(1e-10);
   Deterministic.assert(ek[0], apx);
   Deterministic.assert(epot[0], apx);
   Deterministic.assert(vir[0], apx);
   Deterministic.close();

   ..

   // worker thread
   void run() {
      .. 100 lines of initialization ...
      particle[] particles = ...;
      double force[] = ...;

      for (int i = 0; i < num_iters; i++) {
         // update positions and velocities
         ..
         synchronizeBarrier()
         Predicate pae =
            new ParticleApproxEquals(1e-10);
         Deterministic.assert(particles, pae);
         synchronizeBarrier()

         // update forces
         .. 100 lines plus library calls ...
         synchronizeBarrier()
         Predicate apx =
            new ApproxEquals(1e-10);
         Deterministic.assert(force, apx);
         synchronizeBarrier()

         // temperature scale + sum energy
         .. 40 lines ...
         synchronizeBarrier();
         Deterministic.assert(ek, apx);
         Deterministic.assert(epot, apx);
         Deterministic.assert(vir, apx);
         synchronizeBarrier();
      }
   }
}

Figure 3: Deterministic assertions for moldyn, a molecular dynamics simulator from the Java Grande Forum Benchmark Suite [15].
Table 1: Summary of experimental evaluation of deterministic assertions. A single deterministic block specification was added to each benchmark. Each specification was checked on executions with races found by the CALFuzzzer [43, 37, 29] tool.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Approximate Lines of Code (App + Library)</th>
<th>Lines of Specification (+ Predicates)</th>
<th>Threads</th>
<th>Data Races Found</th>
<th>Determinism Violations</th>
<th>High-Level Races Found</th>
<th>Determinism Violations</th>
</tr>
</thead>
<tbody>
<tr>
<td>sor</td>
<td>300</td>
<td>6</td>
<td>10</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sparsematmult</td>
<td>700</td>
<td>7</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>series</td>
<td>800</td>
<td>4</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>crypt</td>
<td>1100</td>
<td>5</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>moldyn</td>
<td>1300</td>
<td>6</td>
<td>10</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>lufact</td>
<td>1500</td>
<td>9</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>raytracer</td>
<td>1900</td>
<td>4</td>
<td>10</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>montecarlo</td>
<td>3600</td>
<td>4 + 34</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>pi</td>
<td>150 + 15,000</td>
<td>5</td>
<td>4</td>
<td>9</td>
<td>0</td>
<td>1+</td>
<td>1</td>
</tr>
<tr>
<td>keysearch3</td>
<td>200 + 15,000</td>
<td>6</td>
<td>4</td>
<td>3</td>
<td>0</td>
<td>0+</td>
<td>0</td>
</tr>
<tr>
<td>mandelbrot</td>
<td>250 + 15,000</td>
<td>10</td>
<td>4</td>
<td>9</td>
<td>0</td>
<td>0+</td>
<td>0</td>
</tr>
<tr>
<td>phylogeny</td>
<td>4400 + 15,000</td>
<td>8</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>0+</td>
<td>0</td>
</tr>
<tr>
<td>tsp</td>
<td>700</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

1. First, deterministic specifications are easy to write. That is, even for programs for which it is difficult to specify traditional invariants or functional correctness, it is relatively easy for a programmer to add deterministic assertions.

2. Second, deterministic specifications are useful. When combined with tools for exploring multiple thread schedules, deterministic assertions catch real parallelism bugs that lead to semantic non-determinism. Further, for traditional concurrency issues such as data races, these assertions provide some ability to distinguish between benign cases and true bugs.

To evaluate these claims, we used a number of benchmark programs from the Java Grande Forum (JGF) benchmark suite [15], the Parallel Java (PJ) Library [30], and elsewhere. The names and sizes of these benchmarks are given in Table 1. The JGF benchmarks include five parallel computation kernels—for successive order-relaxation (sor), sparse matrix-vector multiplication (sparsematmult), coefficients of a Fourier series (series), cryptography (crypt), and LU factorization (lufact)—as well as a parallel molecular dynamic simulator (moldyn), ray tracer (raytracer), and Monte Carlo stock price simulator (montecarlo). Benchmark tsp is a parallel Traveling Salesman branch-and-bound search [49]. These benchmarks are standard, and have been used to evaluate many previous analyses for parallel programs (e.g. [35, 19, 43]). The PJ benchmarks include an app computing a Monte Carlo approximation of π (pi), a parallel cryptographic key cracking app (keysearch3), an app for parallel rendering Mandelbrot Set images (mandelbrot), and a parallel branch-and-bound search for optimal phylogenetic trees (phylogeny). Note that the benchmarks range from a few hundred to a few thousand lines of code, with the Parallel Java benchmarks relying on an additional 10-20,000 lines of library code from the Parallel Java Library (for threading, synchronization, and other functionality).

5.1 Ease of Use

We evaluate the ease of use of our deterministic specification by manually adding assertions to our benchmark programs. One deterministic block was added to each benchmark.
The third column of Table 1 records the number of lines of specification (and lines of custom predicate code) added to each benchmark. Overall, the specification burden is quite small. Indeed, for the majority of the programs, an author was able to add deterministic assertions in only five to ten minutes per benchmark, despite being unfamiliar with the code. In particular, it was typically not difficult to both identify regions of code performing parallel computation and to determine from documentation, comments, or source code which results were intended to be deterministic. Figures 2 and 3 show the (slightly cleaned up) assertions added to the mandelbrot and moldyn benchmarks.

The added assertions were correct on the first attempt for all but one benchmark. (For phylogeny, the resulting phylogenetic tree was erroneously specified as deterministic, when, in fact, there are many correct optimal trees. The specification was modified to assert only that the optimal score must be deterministic.)

The two predicates provided by our assertion library were sufficient for all but one of the benchmarks. For the JGF montecarlo benchmark, the authors had to write a custom equals and hashCode method for two classes—34 total lines of code—in order to assume and assert that two sets, one of initial tasks and one of results, should be deterministic.

**Further Deterministic Assertions** Three of the benchmarks—sor, moldyn, and lufact—use barriers to synchronize their worker threads at many points during their parallel computations. These synchronization points provide locations where partial results of the computation can be specified to be deterministic. For example, as shown in Figure 3, we can assert in moldyn that the deterministic particle positions and forces should be computed in every iteration. Such intermediate assertions aid the early detection and localization of non-determinism errors.

For these three benchmarks, an author was able to add intermediate assertions at important synchronization barriers in only another fifteen to thirty minutes per benchmark. This adds roughly 25, 35, and 10 lines of specification, respectively, to sor, moldyn, lufact. Further, for the moldyn benchmark, this requires writing a custom predicate ParticleApproxEquals for comparing two arrays of particle objects for approximate equality of their positions and velocities, as well as customizing the serialization of particle objects.

Note, however, that care must be taken with such additional assertions to not capture an excessive amount of data. For example, it is not feasible to assert in every iteration of a parallel computation that a large intermediate matrix is deterministic—this requires serializing and checking a large enough quantity of data to have significant overhead.

**Discussion** More experience, or possibly user studies, would be needed to conclude decisively that our assertions are easier to use than existing techniques for specifying that parallel code is correctly deterministic. However, we believe our experience is quite promising. In particular, writing assertions for the full functional correctness of the parallel regions of these programs seemed to be quite difficult, perhaps requiring implementing a sequential version of the code and asserting that it produces the same result. Further, there seemed to be no obvious simpler, traditional assertions that would aid in catching non-deterministic parallelism.

Despite these difficulties, we found that specifying the natural deterministic behavior of the benchmarks with our bridge assertions required little effort.

### 5.2 Effectiveness

To evaluate the utility of our deterministic specifications in finding true parallelism bugs, we used a modified version of the CAlFuzzzer [43, 37, 29] tool to find real races in the benchmark programs, both data races and higher level races (such as races to acquire a lock). For each such race, we ran 10 trials using CALFuzzzer to create real executions with these races and to randomly resolve the races (i.e. randomly pick a thread to “win”). We turned on run-time checking of our deterministic assertions for these trials, and recorded all found violations.

Table 1 summarizes the results of these experiments. For each benchmark, we indicate the number of real data races and higher-level races we observed. Further, we indicate how many of these races led to determinism violations in any execution.

In these experiments, the primary computational cost is from CALFuzzzer, which typically has an overhead in the range of 2x-20x for these kinds of compute bound applications. We have not carefully measured the computational cost of our deterministic assertion library. For most benchmarks, however, the cost of serializing and comparing a computation’s inputs and outputs is dwarfed by the cost of the computation itself—e.g. consider the cost of checking that two fractal images are identical versus the cost of computing each fractal in the first place.

**Determinism Violations** We found two cases of non-deterministic behavior. First, a known data race in the raytracer benchmark, due the use of the wrong lock to protect a shared sum, can cause an incorrect final answer to be computed.
Second, the \( \pi \) benchmark can yield a non-deterministic answer given the same random seed because of insufficient synchronization of a shared random number generator. In each Monte Carlo sample, two successive calls to `java.util.Random.nextDouble()` are made. A context switch between these calls changes the set of samples generated. Similarly, `nextDouble()` itself makes two calls to `java.util.Random.nextDouble()`, which atomically generates up to 32 pseudo-random bits. A context switch between these two calls changes the generated sequence of pseudo-random doubles. Thus, although `java.util.Random.nextDouble()` is thread-safe and free of data races, scheduling non-determinism can still lead to a non-deterministic result. (This behavior is known—the Parallel Java library provides several versions of this benchmark, one of which does guarantee a deterministic result for any given random seed.)

**Benign Races** The high number of real data races for these benchmarks is largely due to benign races on volatile variables used for synchronization—for example, to implement a tournament barrier or a custom lock. Although CALFuzzer does not understand these sophisticated synchronization schemes, our deterministic assertions automatically provide some confidence that these races are benign because, over the course of many experiment runs, they did not lead to non-deterministic final results.

Note that it can be quite challenging to verify by hand that these races are benign. On inspecting the benchmark code and these data races, an author several times believed he had found a synchronization bug. But on deeper inspection, the code was found to be correct in all such cases.

The number of high-level races is low for the JGF benchmarks because all but `montecarlo` exclusively use volatile variables (and thread joins) for synchronization. Thus, all observable scheduling non-determinism is due to data races.

The number of high-level races is low for the Parallel Java benchmarks because they primarily use a combination of volatile variables and atomic compare-and-set operations for synchronization. Currently, the only kind of high-level race our modified CALFuzzer recognizes is a lock race. Thus, while we believe there are many (benign) races in the ordering of these compare-and-set operations, CALFuzzer does not report them. The one high-level race for `\( \pi \)`, indicated in the table and described above, was confirmed by hand.

**Discussion** Although our checking of deterministic assertions is sound—an assertion failure always indicates that two executions with matching initial states can yield non-matching final states—it is incomplete. Parallelism bugs leading to non-determinism may still exist even when testing fails to find any determinism violations.

However, in our experiments we successfully distinguished the known harmful races from the benign ones in only a small number of trials. Thus, we believe our deterministic assertions can help catch harmful non-determinism due to parallelism, as well as saving programmer effort in determining whether or not real races and other potential parallelism bugs can lead to incorrect program behavior.

### 6 Discussion

In this section, we compare the concepts of atomicity and determinism. Further, we discuss several other possible uses for bridge predicates and assertions.

#### 6.1 Atomicity versus Determinism

A concept complementary to determinism in parallel programs is atomicity. A block of sequential code in a multi-threaded program is said to be **atomic** [22] if for every possible interleaved execution of the program there exists an equivalent execution with the same overall behavior in which the atomic block is executed serially (i.e. the execution of the atomic block is not interleaved with actions of other threads). Therefore, if a code block is atomic, the programmer can assume that the execution of the code block by a thread cannot be interfered with by any other thread. This enables programmers to reason about atomic code blocks sequentially. This seemingly similar concept has the following subtle differences from determinism:

1. Atomicity is the property about a sequential block of code—i.e. the block of code for which we assert atomicity has a single thread of execution and does not spawn other threads. Note that a sequential block is by default deterministic if it is not interfered with by other threads. Determinism is a property of a parallel block of code. In determinism, we assume that the parallel block of code’s execution is not influenced by the external world.

2. In atomicity, we say that the execution of a sequential block of code results in the same state no matter how it is scheduled with other external threads, i.e. atomicity ensures that **external non-determinism** does not interfere
with the execution of an atomic block of code. In determinism, we say that the execution of a parallel block of code gives the same semantic state no matter how the threads inside the block are scheduled—i.e. determinism ensures that internal non-determinism does not result in different outputs.

In summary, atomicity and determinism are orthogonal concepts. Atomicity reasons about a single thread under external non-determinism, whereas determinism reasons about multiple threads under internal non-determinism.

Here we focus on atomicity and determinism as program specifications to be checked. There is much work on atomicity as a language mechanism, in which an atomic specification is instead enforced by some combination of library, run-time, compiler, or hardware support. One could similarly imagine enforcing deterministic specifications through, e.g., compiler and run-time mechanisms [4, 9].

### 6.2 Other Uses of Bridge Predicates

We have already argued that bridge predicates simplify the task of directly and precisely writing deterministic properties in parallel programs. However, we believe that bridge predicates could provide us a simple, but powerful tool to express correctness properties in many other situations. For example, if we have two versions of a program $P_1$ and $P_2$ and if we expect them to produce the same output on same input, then we can easily assert this using our framework as follows:

```plaintext
deterministic assume(Pre) {
    if (nonDeterministicBoolean()) {
        P1
    } else {
        P2
    }
} assert(Post);
```

where $Pre$ requires that the inputs are the same and $Post$ specifies that the outputs will be the same.

In particular, if a programmer has written both a sequential and parallel version of a piece of code, he or she can specify that the two versions are semantically equivalent with an assertion like:

```plaintext
deterministic assume(A==A' and B==B'){
    if (nonDeterministicBoolean()) {
        C = par_matrix_multiply_int(A, B);
    } else {
        C = seq_matrix_multiply_int(A, B);
    }
} assert(C==C');
```

where `nonDeterministicBoolean()` returns true or false non-deterministically.

Recall the way we have implemented our determinism checker in Java—we serialize a pair of projections of the input and output states for each execution to the file-system. This particular implementation allows us to quickly write regression tests simply as follows:

```plaintext
deterministic assume(Pre) {
    P
} assert(Post);
```

where $Pre$ asserts that the inputs are the same and $Post$ asserts that the outputs are the same. In the above code, we simply assert that the input-output behavior of $P$ remains the same even if $P$ changes over time, but maintains the same input-output behavior. The serialized input and output states implicitly store the regression test on the file-system.

Further, we believe there is a wider class of program properties that are easy to write in bridge assertions but would be quite difficult to write otherwise. For example, consider the specification:

```plaintext
deterministic assume(set.size() == set'.size()) {
    P
} assert(set.size() == set'.size());
```
This specification requires that sequential or parallel program block $P$ transforms set so that its final size is the same function of its initial size independent of its elements. The specification is easy to write even in cases where the exact relationship between the initial and final size might be quite complex and difficult to write. It is not entirely clear, however, when such properties would be important or useful to specify/assert.

7 Related Work

As discussed in Section 1, there is a large body of work attacking harmful program non-determinism by detecting data races. There has also been recent work on detecting or eliminating other sources of non-determinism such as high-level races [49, 5] and atomicity violations [21, 19, 20, 37].

For more than forty years, assertions—formal constraints on program behavior embedded in a program’s source—have been used to specify and prove the correct behavior of sequential [23, 25] and parallel [36] programs. More recently, assertions have found widespread use as a tool for checking at run-time for software faults to enable earlier detection and easier debugging of software errors [39, 33]. In this work, we propose bridge assertions, which relate pairs of states from different program executions.

Sadowski, et al., [40] propose a different notion of determinism, one that is a generalization of atomicity. They say that a parallel computation is deterministic if is both free from external interference (externally serializable) and if its threads communicate with each other in a strictly deterministic fashion (internal conflict freedom). That is, for a computation to be deterministic not only must it contain no data races, but the partially-ordered sequence of lock operations and other synchronization events must be identical on every execution. These conditions ensure that every schedule produces bit-wise identical results. Further, [40] proposes a sound dynamic determinism analysis that can identify determinism violations in a single execution of a program under test.

This form of determinism from [40] is much more strict than the determinism proposed in this work. Our deterministic specifications can be applied to programs, such as those using locks or shared buffers, in which internal threads communicate non-deterministically, but still produce deterministic final results. Further, we allow users to provide custom predicates to specify what is means for the results of two different thread schedules to be semantically deterministic.

Siegel, et al., [44] propose a technique for combining symbolic execution with model checking to verify that parallel, message-passing numerical programs compute equivalent answers to their sequential implementations.

8 Conclusion

We have introduced bridge predicates and bridge assertions for relating pairs of states across different executions. We have shown how these predicates and assertions can be used to easily and directly specify that a parallel computation is deterministic. And we have shown that such specifications can be useful in finding parallel non-determinism bugs and in distinguishing harmful from benign races. Further, we believe that bridge assertions may have other potential uses.

9 Acknowledgments

We would like to thank Nicholas Jalbert, Mayur Naik, Chang-Seo Park, and our anonymous reviewers for their valuable comments on previous drafts of this paper. This work supported in part by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227), by NSF Grants CNS-0720906 and CCF-0747390, and by a DoD NDSEG Graduate Fellowship.

Bibliography


Race Directed Random Testing of Concurrent Programs

Koushik Sen

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.

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 interleavings 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 hede, 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 $s$. 

• **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 $s$. 

• 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.
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.

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

\[
\begin{align*}
& e_i = \text{MEM}(\sigma_i, m_i, a_i, t_i, L_i) \land e_j = \text{MEM}(\sigma_j, m_j, a_j, t_j, L_j) \\
& \land t_i \neq t_j \land m_i = m_j \land (a_i = \text{WRITE} \lor a_j = \text{WRITE}) \\
& \land L_i \cap L_j = \emptyset \land \neg(e_i \prec e_j) \land \neg(e_j \prec e_i)
\end{align*}
\]

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 \( (\sigma_i, \sigma_j) \) 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 \( \text{notify}() \) on thread \( t_1 \) signals a \( \text{wait}() \) 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.

Koushik Sen 323
postponing threads, it may happen that several threads are about to execute a statement in RaceSet. In that scenario, we randomly resolve the race by allowing one thread between the two threads to execute the next statement to be executed. In such a case, we keep postponing all the threads that are about to execute a statement in RaceSet because they would access different dynamic shared memory locations when they execute their next statements.

Phase 2. RaceFuzzer.

Our key contribution is the second phase of the algorithm, which we next describe informally. Let \((\sigma_1, \sigma_2)\) 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 RaceSet, a set of two racing statements \(RaceSet\), 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 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

---

**Algorithm 1: Algorithm RaceFuzzer**

1: Inputs: the initial state \(s_0\), a set of two racing statements \(RaceSet\)
2: \(s := s_0\)
3: \(postponed := \emptyset\)
4: while \(Enabled(s) \neq \emptyset\) do
5: \(t := \text{a random thread in } Enabled(s) \setminus postponed\)
6: if \(NextStmt(s, t) \in RaceSet\) then
7: \(R := \text{Racing}(s, t, postponed)\)
8: if \(R \neq \emptyset\) then /* Actual race detected */
9: print “ERROR: actual race found”
10: if random boolean then /* Randomly resolve race */
11: \(s := \text{Execute}(s)\)
12: else
13: \(postponed := postponed \cup \{t\}\)
14: for all \(t' \in R\) do
15: \(s := \text{Execute}(s, t')\)
16: \(postponed := postponed \setminus \{t'\}\)
17: end for
18: end if
19: \(postponed := postponed \cup \{t\}\)
20: end if
21: else /* Wait for a race to happen */
22: \(s := \text{Execute}(s, t)\)
23: end if
24: if \(Active(s) \neq \emptyset\) then
25: print “ERROR: actual deadlock found”
26: end if
27: end while
28: return
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.

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 $L$.

---

**Algorithm 2:** Function Racing($s,t$, postponed)

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

---

Koushik Sen
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.

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.

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 pseudo-code 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 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 synchro-
nization 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 multi-threaded 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], hede, a web-crawler application kernel developed at ETH [53], webtech, 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.
Table 1: Experimental results.

<table>
<thead>
<tr>
<th>Program Name</th>
<th>SLOC</th>
<th>Average Runtime in sec.</th>
<th># of Races</th>
<th># of Exceptions</th>
<th>Probability of hitting a race</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Normal</td>
<td>Hybrid</td>
<td>RF</td>
<td>Hybrid</td>
</tr>
<tr>
<td>moldyn</td>
<td>1352</td>
<td>2.07</td>
<td>3600</td>
<td>42.37</td>
<td>59</td>
</tr>
<tr>
<td>raytracer</td>
<td>1924</td>
<td>3.25</td>
<td>3600</td>
<td>3.81</td>
<td>2</td>
</tr>
<tr>
<td>montecarlo</td>
<td>3619</td>
<td>3.48</td>
<td>3600</td>
<td>6.44</td>
<td>5</td>
</tr>
<tr>
<td>cache4j</td>
<td>3897</td>
<td>2.19</td>
<td>4.26</td>
<td>2.61</td>
<td>18</td>
</tr>
<tr>
<td>sor</td>
<td>17689</td>
<td>0.16</td>
<td>0.35</td>
<td>0.23</td>
<td>8</td>
</tr>
<tr>
<td>hedc</td>
<td>29948</td>
<td>1.10</td>
<td>1.35</td>
<td>1.11</td>
<td>9</td>
</tr>
<tr>
<td>weblech</td>
<td>35175</td>
<td>0.91</td>
<td>1.92</td>
<td>1.36</td>
<td>27</td>
</tr>
<tr>
<td>jspider</td>
<td>64933</td>
<td>4.79</td>
<td>4.88</td>
<td>4.81</td>
<td>29</td>
</tr>
<tr>
<td>jigsaw</td>
<td>381348</td>
<td>-</td>
<td>-</td>
<td>0.81</td>
<td>547</td>
</tr>
</tbody>
</table>

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 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:**

```java
synchronized(this){
    if(_sleep){
        sleep(_cleanInterval);
        interrupt();
    }
}
```

**Thread2:**

```java
try {
    sleep(_cleanInterval);
} catch (InterruptedException 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
containsAll1 method is implemented by the superclass AbstractCollection and the implementation uses iterator in a thread-unsafe way: a call to containsAll1(12) calls the synchronized iterator method on 12 and then goes over the iterator without holding the lock on 12. As a result, the iterator accesses the modCount field of 12 without holding the lock on 12. Therefore, any other method call on 12 that modifies modCount, such as removeAll1, 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 containsAll1 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 containsAll1 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 built-in 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.

7 Acknowledgments

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.

Bibliography


