Asserting and Checking Determinism for Multithreaded Programs

Jacob Burnim
EECS Department, UC Berkeley, CA, USA
jburnim@cs.berkeley.edu

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

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.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.5 [**Software Engineering**]: Testing and Debugging; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms

Reliability, Verification

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 [47, 32, 28, 3, 31] and the design of new type systems and annotations that could retrofit existing parallel languages [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 [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 [13, 1, 41, 49, 11, 2, 43] and static [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 [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 [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:

```
deterministic assume(Pre(s₀, s'₀)) {
    P
} assert(Post(s, s'));
```

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:

```
deterministic assume(|A - A'| < 10^-6 and
                     |B - B'| < 10^-6) {
  C = parallel_matrix_multiply_float(A, B);
} assert(|C - C'| < 10^-6);
```

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:

```
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:

```
deterministic {
  C = parallel_matrix_multiply_int(A, B);
}
```

asserts that for the parallel implementation of matrix multiplication (defined by function `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:

```
deterministic {
  C = parallel_matrix_multiply_float(A, B);
}
```

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:

```
deterministic {
  C = parallel_matrix_multiply_float(A, B);
} 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:

```
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:

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

where `Post` is a predicate over two program states in different executions resulting from different thread schedules[1]. 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:

```
deterministic assume(s_0 == s'_0) {
   P
} assert(Post);
```

That is, if any two executions of `P` start from initial states s_0 and s'_0, respectively, and if s and s' are the resultant final states, then $s_0 == s'_0$ implies that `Post`(s, s') holds. The above rewritten specification suggests that we can further relax the requirement of $s_0 == s'_0$ by replacing it with a bridge predicate `Pre`(s_0, s'_0). For example:

```
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.

Comparison to Traditional Assertions.

In summary, we propose the following construct for the specification of deterministic behavior.

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

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

- `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 `Post`(s, s') must hold.

Note that the use of bridge predicates `Pre` and `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 `Pre` and `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:

```
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 `Post` predicate is that a post-condition relates two states at different times along a same execution, whereas a `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 `parallel_matrix_multiply_int`:

```
C = parallel_matrix_multiply_int(A, B);
assert(C == A × B);
```

[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.

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 `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:

```
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_{pre} and s_{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:

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

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

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

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].

```
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.

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 `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.

Each call `assume`$(o, \textit{pre})$ 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`$(o, \textit{post})$ 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].

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 thread 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 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.

```
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].

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:

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

```
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].

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 (`phylogenetic`). 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

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

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 CalFuzzer [43, 37, 29] tool.

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 CalFuzzer [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 CalFuzzer 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 CalFuzzer, 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.next()`, 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 `P1` and `P2` and if we expect them to produce the same output on same input, then we can easily assert this using our framework as follows:

```
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:

```
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:

```
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:

```
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' 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.

10. REFERENCES

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

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

[3] G. Agha. *Actors: A Model of Concurrent Computation*. MIT Press, 1986.

[4] Z. Anderson, D. Gay, R. Ennals, and E. Brewer. SharC: checking data sharing strategies for multithreaded c. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation (PLDI 2008)*, pages 149–158. ACM New York, NY, USA, 2008.

[5] C. Artho, K. Havelund, and A. Biere. High-level data races. *Software Testing Verification and Reliability*, 13(4):207–227, 2003.

[6] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. D. Kubiatowicz, E. A. Lee, N. Morgan, G. Necula, D. A. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. A. Yelick. The parallel computing laboratory at u.c. berkeley: A research agenda based on the berkeley view. Technical Report UCB/EECS-2008-23, EECS Department, University of California, Berkeley, Mar 2008.

[7] G. Barnes. A method for implementing lock-free shared-data structures. In *Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures*, pages 261–270. ACM New York, NY, USA, 1993.

[8] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming (PPOPP)*, 1995.

[9] R. Bocchino, V. Adve, S. Adve, and M. Snir. Parallel programming must be deterministic by default. In *First USENIX Workship on Hot Topics in Parallelism (HOTPAR 2009)*, March 2009.

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

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

[12] L. Dagum, R. Menon, and S. Inc. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science & Engineering*, 5(1):46–55, 1998.

[13] A. Dinning and E. Schonberg. Detecting access anomalies in

programs with critical sections. In *Proc. of the ACM/ONR Workshop on Parallel and Distributed Debugging*, 1991.

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

[15] Edinburgh Parallel Computing Centre. Java Grande Forum benchmark suite. `www2.epcc.ed.ac.uk/computing/research_activities/java_grande/index_1.html`.

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

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

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

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

[20] C. Flanagan, S. N. Freund, and J. Yi. Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 293–303. ACM, 2008.

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

[22] C. Flanagan and S. Qadeer. Types for atomicity. In *TLDI '03: Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 1–12, New York, NY, USA, 2003. ACM.

[23] R. Floyd. Assigning meanings to programs. *Mathematical aspects of computer science*, 19(19-32):1, 1967.

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

[25] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[26] IEEE. POSIX Part 1: System API- Amend. 1: Realtime Extension [C Language]. *IEEE Std 1003.1b-1993*, 1994.

[27] Intel®. Threading Building Blocks for open source. `http://threadingbuildingblocks.org/`.

[28] W. M. Johnston, J. R. P. Hanna, and R. J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36(1):1–34, 2004.

[29] P. Joshi, C.-S. Park, K. Sen, and M. Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'09)*, 2009.

[30] A. Kaminsky. Parallel Java: A Unified API for Shared Memory and Cluster Parallel Programming in 100% Java. In *21st IEEE International Parallel and Distributed Processing Symposium (IPDPS 2007)*, March 2007.

[31] E. A. Lee. The problem with threads. *Computer*, 39(5):33–42, May 2006.

[32] H. Loidl, F. Rubio, N. Scaife, K. Hammond, S. Horiguchi, U. Klusik, R. Loogen, G. Michaelson, R. Pena, S. Priebe, et al. Comparing parallel functional languages: Programming and performance. *Higher-Order and Symbolic Computation*, 16(3):203–251, 2003.

[33] B. Meyer. Applying 'design by contract'. *Computer*, 25(10):40–51, 1992.

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

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

[36] S. Owicki and D. Gries. Verifying properties of parallel programs: an axiomatic approach. *Commun. ACM*, 19(5):279–285, 1976.

[37] C.-S. Park and K. Sen. Randomized active atomicity violation detection in concurrent programs. In *SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2008.

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

[39] D. S. Rosenblum. Towards a method of programming with assertions. In *ICSE '92: Proceedings of the 14th international conference on Software engineering*, pages 92–104, New York, NY, USA, 1992. ACM.

[40] C. Sadowski, S. Freund, and C. Flanagan. SingleTrack: A Dynamic Determinism Checker for Multithreaded Programs. In *18th European Symposium on Programming (ESOP)*, 2009.

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

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

[43] K. Sen. Race directed random testing of concurrent programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*, 2008.

[44] S. F. Siegel, A. Mironova, G. S. Avrunin, and L. A. Clarke. Combining symbolic execution with model checking to verify parallel numerical programs. *ACM Trans. Softw. Eng. Methodol.*, 17(2):1–34, 2008.

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

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

[47] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A language for streaming applications. *Lecture Notes in Computer Science*, pages 179–196, 2002.

[48] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, 2003.

[49] C. von Praun and T. R. Gross. Object race detection. In *Proc. of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications (OOPSLA'01)*, pages 70–82, 2001.