DOI:10.1145/1743546.1743572

Asserting and Checking Determinism for Multithreaded Programs

By Jacob Burnim and Koushik Sen

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 *nondeterminism*. The threads of a parallel application may be interleaved nondeterministically during execution. In a buggy program, nondeterministic scheduling can lead to nondeterministic results—where some interleavings produce the correct result while others do not.

We propose an assertion framework for specifying that regions of a parallel program behave deterministically despite nondeterministic 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 nondeterministic 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.[2]

This new trend of increasingly parallel chips means that we will have to write parallel software in order to take advantage of future hardware advances. Unfortunately, parallel software is more difficult to write and debug than its sequential counterpart. A key reason for this difficulty is *nondeterminism*—i.e., that in two runs of a parallel program on the exact same input, the parallel threads of execution can interleave differently, producing different output. Such nondeterministic thread interleaving is an essential part of harnessing the power of parallel chips, but it is a major departure from sequential programming, where we typically expect programs to behave identically in every execution on the same input. We share a widespread belief that helping programmers manage nondeterminism in parallel software is critical in making parallel programming widely accessible.

For more than 20 years, many researchers have attacked the problem of nondeterminism by attempting to detect or predict *sources* of nondeterminism in parallel programs. The most notorious of such sources is the *data race*. A data race occurs when two threads in a program concurrently access the same memory location and at least one of those accesses is a write. That is, the two threads "race" to perform their conflicting memory accesses, so the order in which the two accesses occur can change from run to run, potentially yielding nondeterministic program output. Many algorithms and tools have been developed to detect and eliminate data races in parallel programs. (See Burnim and Sen[5] for further discussion and references.) Although the work on data race detection has significantly helped in finding determinism bugs in parallel programs, it has been observed that the absence of data races is not sufficient to ensure determinism.[1, 8, 9] Thus researchers have also developed techniques to find high-level races,[1, 16, 21] likely atomicity violations,[9, 8, 14] and other potential sources of nondeterminism. Further, such sources of nondeterminism are not always bugs—they may not lead to nondeterministic program behavior or nondeterminism may be intended. In fact, race conditions may be useful in gaining performance while still ensuring high-level deterministic behavior.[3]

More recently, a number of ongoing research efforts aim to make parallel programs deterministic *by construction*. These efforts include the design of new parallel programming paradigms[10, 12, 13, 19] and the design of new type systems, annotations, and checking or enforcement mechanisms that could retrofit existing parallel languages.[4, 15] 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 new type system may require complex type annotations and may forbid reasonable programs whose determinism cannot be expressed in the type system.

We argue that programmers should be provided with a framework that will allow them *to express deterministic*

The original version of this paper was published in *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, August 2009.

behaviors of parallel programs ***directly*** *and* ***easily***. Specifically, we should provide an assertion framework where programmers can *directly* and *precisely* express intended deterministic behavior. Further, 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 race detection; nor 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 input 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* and *s*′. Then, if the user-specified *precondition* `Pre` holds over s_0 and s'_0, then *s* and *s*′ must satisfy the user-specified *postcondition* `Post`.

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

```
deterministic assume(|A − A'| < 10⁻⁹ and
                     |B − B'| < 10⁻⁹) {
  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. Thus, the predicates that we write inside `assume` and `assert` are different from state predicates written in a traditional assertion framework—our predicates relate a pair of states from different executions. We call such predicates *bridge predicates* and assertions using bridge predicates *bridge assertions*. A key contribution of this paper is the introduction of these bridge predicates and bridge assertions.

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 nondeterministically lead to semantically different answers, we can find bugs in a program's use of parallelism even when unable to directly specify or 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 races known to lead to undesired nondeterminism from the benign races in the benchmarks.

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 in 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);
}
```

Limited-precision 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 the 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 the 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 {
    set.add(3); || set.add(5);
} 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 different parallel 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* the same. 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 from different executions with 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:

```
deterministic assume(s₀ = 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 resulting final states, then $s_0 = s'_0$ implies that `Post` (*s, s′*) holds. The above rewritten specification suggests that we can 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′)) {
    set.add(3);  ||  set.add(5);
} 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 must also contain 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 (1) `Pre` (s_0, s'_0) holds, (2) an execution of `P` from s_0 terminates and results in state *s*, and (3) 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 postconditions used for functions in program verification. However, unlike traditional pre- and postconditions, the proposed `Pre` and `Post` predicates relate pairs of states from two different executions. In traditional verification, a precondition is usually written as a predicate over

a single program state, and a postcondition is usually written over two states—the states at the beginning and end of the function. For example:

```
parallel_matrix_multiply_int(A, B) {
    assume(A.cols == B.rows);
    ...
    assert(C == A × B);
    return C;
}
```

The key difference between a postcondition and a `Post` predicate is that a postcondition relates two states at different times along a same execution—e.g., here relating inputs `A` and `B` to output `C`—whereas a `Post` predicate relates two program states from 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 postconditions. For example, one could write the following to specify the correct behavior of a paralell matrix multiply:

```
C = parallel_matrix_multiply_float(A, B);
assert(|C − A × B| < 10⁻⁶);
```

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 is 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. A simple deterministic specification, however, enables us to use automated techniques to check for parallelism bugs, such as harmful data races causing semantically nondeterministic behavior.

Consider a 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 postconditions 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 if 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 Section 5, 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, sound, 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_{pre}, s_{post}) pairs for a particular deterministic block in some program, we can check a deterministic specification 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 postbridge predicates `Pre` and `Post`, we check for every recorded pair of pairs (s_{pre}, s_{post}) and (s'_{pre}, s'_{post}) that:

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

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 and inputs. 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,[7, 18] active random scheduling,[16] or model checking.[20]

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 postpredicate `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. This storage cost sometimes can be further reduced by storing and comparing check-sums or approximate hashes.

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 provide the functionality as a library to simplify 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 postpredicates of deterministic behavior.

Each call `assume`(*o, pre*) in a deterministic block specifies part of the prepredicate 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, saving them to a central, persistent store. 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.)

Figure 1. Core deterministic specification API.

```
public 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);
  }
}
```

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 postpredicate 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.

Built-in Predicates: For programmer convenience, we provide two built-in predicates that are often sufficient for specifying pre- and postpredicates for determinism. The first, `Equals`, returns *true* if the given objects are equal using their built-in `equals` method—i.e., 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 multidimensional 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. We found this predicate useful in specifying the deterministic behavior of numerical applications, where it is often unavoidable that the low-order bits may vary with different thread interleavings.

Real-World Floating-Point Predicates: In practice, floating-point computations often have input-dependent error bounds. For example, we may expect any two runs of a parallel algorithm for summing inputs $x_1, \ldots, x_n$ to return answers

Figure 2. Deterministic assertions for a Mandelbrot Set implementation from the Parallel Java (PJ) Library.[11]

```
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[][] = ...;
  ...
  // join threads
  ...
  Deterministic.assert(matrix, equals);
  Deterministic.close();
  // write fractal image to file
  ...
}
```

equal to within $2N\epsilon\sum_i|x_i|$, where ϵ is the machine epsilon. We can assert:

```
sum = parallel_sum(x);
bound = 2 * x.length * ϵ * sum_of_abs(x);
Predicate apx = new ApproxEquals(bound);
Deterministic.assert(sum, apx);
```

As another example, different runs of a molecular dynamics simulation may be expected to produce particle positions equal to within something like ϵ multiplied by the sum of the absolute values of all initial positions. We can similarly compute this value at the beginning of the computation, and use an `ApproxEquals` predicate with the appropriate bound to compare particle positions.

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 (PJ) Library.[11]

The benchmark first reads a number of integer and floating-point parameters from the command-line. It then spawns several worker threads that each compute the hues for different segments of the final image and store the hues 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 complicated 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.

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 nondeterminism. 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,[17] the Parallel Java (PJ) Library,[11] and elsewhere. The names and sizes of these benchmarks are given in Table 1. We describe the benchmarks in greater detail in Burnim and Sen.[5] Note that the benchmarks range from a few hundred to a few thousand lines of code, with the PJ benchmarks relying on an additional 10–20,000 lines of library code from the PJ 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.

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[14, 16] tool.

	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	1,100	5	10	0	0	0	0
	moldyn	1,300	6	10	2	0	0	0
	lufact	1,500	9	10	1	0	0	0
	raytracer	1,900	4	10	3	**1**	0	0
	montecarlo	3,600	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	4,400 + 15,000	8	4	4	0	0+	0
	tsp	700	4	5	6	0	2	0

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 5 to 10 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. Figure 2 shows the assertions added to the `mandelbrot` benchmark.

The added assertions were correct on the first attempt for all but two benchmarks. 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. For `sparsematmult`, we incorrectly identified the variable to which the output was written. This error was identified during later work on automatically inferring deterministic specifications.[6]

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, are equivalent across executions.

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 nondeterministic 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[14, 16] 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 nondeterministic behavior. First, a known data race in the `raytracer` benchmark, due the use of the wrong lock to protect a shared sum, can yield an incorrect final answer.

Second, the `pi` benchmark can yield a nondeterministic 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 pseudorandom bits. A context switch between these two calls changes the generated sequence of pseudorandom doubles. Thus, although `java.util.Random.nextDouble()` is thread-safe and free of data races, scheduling nondeterminism can still lead to a nondeterministic result. (This behavior is known—the PJ 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—e.g., 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 experimental runs, they did not lead to nondeterministic 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 the benchmarks except `montecarlo` exclusively use volatile variables (and thread joins) for synchronization. Thus, all observable scheduling nondeterminism is due to data races.

The number of high-level races is low for the PJ 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 equivalent initial states can yield nonequivalent final states—it is incomplete. Parallelism bugs leading to nondeterminism may still exist even when testing fails to find any determinism violations.

However, in our experiments we successfully distinguished the races known to cause undesired nondeterminism from the benign races in only a small number of trials. Thus, we believe our deterministic assertions can help catch harmful nondeterminism due to parallelism, as well as save programmer effort in determining whether 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 bridge assertions.

6.1. Atomicity versus determinism

A concept complementary to determinism in parallel programs is atomicity. A block of sequential code in a multithreaded program is said to be *atomic*[9] 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 nondeterminism* 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 nondeterminism* does not result in different outputs.

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

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 runtime mechanisms.[4]

6.2. Other uses of bridge predicates

We have already argued that bridge predicates simplify the task of directly and precisely specifying deterministic behavior of parallel programs. We also believe that bridge predicates could provide 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`, that we expect to produce the same output on the 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` nondeterministically.

Similarly, a programmer can specify that the old and new versions of a piece of code are semantically equivalent:

```
deterministic assume(A==A′ and B==B′) {
   if (nonDeterministicBoolean()) {
      C = old_matrix_multiply_int(A, B);
   } else {
      C = new_matrix_multiply_int(A, B);
   }
} assert(C==C′);
```

Checking this specification is a kind of regression testing. In particular, if the code change has introduced a regression—i.e., a bug that causes the new code to produce a semantically different output then the old code for some input—then the above specification does not hold.

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 some 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 are important or useful to specify.

7. CONCLUSION

We have proposed bridge predicates and bridge assertions for specifying the user-intended semantic deterministic behavior of parallel programs. We argue that our specifications are much simpler for programmers to write than traditional specifications of functional correctness, because they enable programmers to compare pairs of program states across different executions rather than relating program outputs directly to program inputs. Thus, bridge predicates and bridge assertions can be thought of as a lightweight mechanism for specifying the correctness of just the parallelism in a program, independently of the program's functional correctness.

We have shown experimental evidence that we can effectively check our deterministic specifications. In particular, we can use existing techniques for testing parallel software to generate executions exhibiting data and higher-level races. Then our deterministic specifications allow us to distinguish from the benign races the parallel nondeterminism bugs that lead to unintended nondeterministic program behavior. Thus, we argue that it is worthwhile for programmers to write such lightweight deterministic specifications. In fact, later work[6] has suggested that, given the simple form of our specifications, it may often be possible to automatically infer likely deterministic specifications for parallel programs.

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.

References

1. Artho, C., Havelund, K., Biere, A. High-level data races. *Softw. Test. Ver. Reliab. 13*, 4 (2003), 207–227.
2. Asanovic, K., Bodik, R., Demmel, J., Keaveny, T., Keutzer, K., Kubiatowicz, J.D., Lee, E.A., Morgan, N., Necula, G., Patterson, D.A., Sen, K., Wawrzynek, J., Wessel, D., Yelick, K.A. *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, March 2008.
3. Barnes, G. A method for implementing lock-free shared-data structures. In *5th ACM Symposium on Parallel Algorithms and Architectures* (*SPAA*) (1993).
4. Bocchino, R.L., Jr., Adve, V.S., Dig, D., Adve, S.V., Heumann, S., Komuravelli, R., Overbey, J., Simmons, P., Sung, H., Vakilian, M. A type and effect system for deterministic parallel Java. In *24th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications* (*OOPSLA*) (2009).
5. Burnim, J., Sen, K. Asserting and checking determinism for multithreaded programs. In *7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering* (*ESEC/FSE*) (2009).
6. Burnim, J. Sen, K. DETERMIN: Inferring likely deterministic specifications of multithreaded programs. In *32nd ACM/IEEE International Conference on Software Engineering* (*ICSE*) (2010).
7. Edelstein, O., Farchi, E., Nir, Y., Ratsaby, G., Ur, S. Multithreaded Java program test generation. *IBM Syst. J. 41*, 1 (2002), 111–125.
8. Flanagan, C., Freund, S.N. Atomizer: A dynamic atomicity checker for multithreaded programs. In *31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (*POPL*) (2004).
9. Flanagan, C., Qadeer, S. A type and effect system for atomicity. In *ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation* (*PLDI*) (2003).
10. Johnston, W.M., Hanna, J.R.P., Millar, R.J. Advances in dataflow programming languages. *ACM Comput. Surv. 36*, 1 (2004), 1–34.
11. Kaminsky, A. 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).
12. Lee, E.A. The problem with threads. *Computer 39*, 5 (May 2006), 33–42.
13. Loidl, H., Rubio, F., Scaife, N., Hammond, K., Horiguchi, S., Klusik, U., Loogen, R., Michaelson, G., Pena, R., Priebe, S. et al. Comparing parallel functional languages: Programming and performance. *High. Order Symb. Comput. 16*, 3 (2003), 203–251.
14. Park, C.-S., Sen, K. Randomized active atomicity violation detection in concurrent programs. In *16th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (*FSE*) (2008).
15. Sadowski, C., Freund, S., Flanagan, C. SingleTrack: A dynamic determinism checker for multithreaded programs. In *18th European Symposium on Programming* (*ESOP*) (2009).
16. Sen, K. Race directed random testing of concurrent programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI'08*) (2008).
17. Smith, L.A., Bull, J.M., Obdrzálek, J. A parallel java grande benchmark suite. In *ACM/IEEE Conference on Supercomputing* (*SC*) (2001).
18. Stoller, S.D. Testing concurrent Java programs using randomized scheduling. In *2nd Workshop on Runtime Verification* (*RV*) (2002).
19. Thies, W., Karczmarek, M., Amarasinghe, S. StreamIt: A language for streaming applications. In *11th International Conference on Compiler Construction* (*CC*) (2002).
20. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F. Model checking programs. *Autom. Softw. Eng. 10*, 2 (2003), 203–232.
21. von Praun, C., Gross, T.R. Object race detection. In *16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (*OOPSLA*) (2001).

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

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