DETERMIN: Inferring Likely Deterministic Specifications of Multithreaded Programs

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

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

ABSTRACT

The trend towards multicore processors and graphic processing units is increasing the need for software that can take advantage of parallelism. Writing correct parallel programs using threads, however, has proven to be quite challenging due to *nondeterminism*. The threads of a parallel application may be interleaved nondeterministically during execution, which can lead to nondeterministic results—some interleavings may produce the correct result while others may not. We have previously proposed an assertion framework for specifying that regions of a parallel program behave deterministically despite nondeterministic thread interleaving. The framework allows programmers to write assertions involving pairs of program states arising from different parallel schedules.

We propose an algorithm to dynamically infer likely deterministic specifications for parallel programs given a set of inputs and schedules. We have implemented our specification inference algorithm for Java and have applied it to a number of previously examined Java benchmarks. We were able to automatically infer specifications largely equivalent to or stronger than our manual assertions from our previous work.

We believe that the inference of deterministic specifications can aid in understanding and documenting the deterministic behavior of parallel programs. Moreover, an unexpected deterministic specification can indicate to a programmer the presence of erroneous or unintended behavior.

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, Documentation

Keywords

determinism, specification inference, parallel programs

ICSE '10, May 2-8 2010, Cape Town, South Africa
Copyright 2010 ACM 978-1-60558-719-6/10/05 ...$10.00.

1. INTRODUCTION

With the growing prevalence of multicore microprocessors and graphic processing units (GPUs), software engineers are increasingly required to write parallel programs. Unfortunately, parallel programs have proven to be much more difficult to write and debug than sequential software. A key culprit in 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 arbitrarily interleave with each other is called *internal nondeterminism* or *scheduler nondeterminism*.

Internal nondeterminism enables multiple threads to run simultaneously, which is essential to harness the power of multicore chips. However, internal nondeterminism in parallel programs could also result in nondeterministic outputs. 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 write, test, and debug, we need to make them behave like sequential programs, i.e. we need to make parallel programs deterministic.

The most widespread method for writing parallel programs, threads, requires programmers to ensure determinism. Ensuring deterministic behavior in such programs is generally very challenging. Thus, a variety of tools and techniques have been proposed to help ensure that multithreaded programs exhibit their intended determinism. These tools attempt to automatically find sources of nondeterminism likely to be harmful (i.e. to lead to nondeterministic output) such as data races [25] and high-level race conditions. However, the absence of data races does not always guarantee deterministic behavior [5, 16, 13]. On the other hand, the presence of data races may not lead to unintended nondeterministic behavior, and may aid in achieving high performance [6].

We have argued previously [8] that programmers should have a way to directly and easily specify that a parallel software application behaves deterministically. We proposed [8] a scheme for asserting that a block of parallel code exhibits the intended, user-specified *semantic determinism*. Formally, our framework allowed a programmer to give a specification for a block C of parallel code as:

```
deterministic assume(Pre(s_0, s'_0)) {
    C
} assert(Post(s, s'));
```

This specification asserts the following: Suppose C 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', respectively. 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`.

We argued [8] that such assertions allow a programmer to specify the correctness of the use of parallelism in an application independently of the functional correctness. That is, one can specify that different executions of a parallel program on the same input cannot erroneously produce non-equivalent outputs due to scheduling nondeterminism. This can be accomplished without having to specify anything about the correctness of individual outputs in terms of their corresponding inputs. Our experiments [8] showed that if the deterministic specification of a parallel program is provided, we can distinguish true races from benign ones in the program and find bugs in parallel programs that arise due to internal nondeterminism.

In this paper, we propose to automatically infer likely deterministic specifications for parallel programs. Specifically, given a set of test inputs and thread schedules, for each procedure P of a parallel program, we infer a deterministic specification for the body of a procedure P:

```
void $P$() {
     deterministic assume(Pre($s_0, s'_0$)) {
          ... body of $P$ ...
     } assert(Post($s, s'$));
}
```

A key challenge in inferring likely deterministic specification of a parallel program is that there could be several specifications for the program; however, not all of the specifications are interesting. For example, the following deterministic specification holds for any parallel program, where `Pre` is any predicate.

```
void $P$() {
     deterministic assume(Pre($s_0, s'_0$)) {
          ... body of $P$ ...
     } assert(true);
}
```

To address the problem of inferring "interesting" deterministic specifications, we argue that a `(Pre, Post)` pair is "interesting" if the following two conditions hold.

1. `Pre` is a weakest liberal precondition for `Post` and `Post` is a strongest liberal postcondition for `Pre`, and

2. `Post` is the strongest liberal postcondition for any possible `Pre`, which we show to be unique.

We give an algorithm, DETERMIN, to compute one such "interesting" deterministic specification from a set of executions observed on a given set of inputs and schedules. We formally prove that if the set of given inputs and schedules is the set of all inputs and schedules, then we infer an actual "interesting" deterministic specification of the program.

We have implemented DETERMIN for Java and have applied it to a number of previously examined Java benchmarks. We were able to infer specifications largely equivalent to or stronger than our manual assertions from [8].

We believe that the inference of deterministic specifications can aid in program understanding and documentation of deterministic behavior of parallel programs. Specifically, a correct inferred specification documents for programmers the deterministic aspect of the parallel behavior of an application. Moreover, an unexpected deterministic specification can indicate to a programmer the presence of buggy or otherwise unintended behavior. For example, consider a specification indicating a critical component of the program output is *not* deterministic; or consider a specification indicating that a program's determinism hinges on some believed-to-be insignificant portion of the input.

Related Work

There is a rich literature on invariant generation [12, 15, 4, 32, 30, 33, 3, 24, 28, 9]. Daikon [12] automatically infers likely program invariants using statistical inference from a program's execution traces. Csallner et al. [9] propose an approach, called DySy, that combines symbolic execution with dynamic testing to infer preconditions and postconditions for program methods. Hangal and Lam [18] propose DIDUCE, which uses online analysis to discover simple invariants over the values of program variables. Deryaft [24] is a tool that specializes in generating constraints of complex data structures. Logozzo [22] proposed a static approach that derives invariants for a class as a solution of a set of equations derived from the program source. Houdini [15] is an annotation assistant for ESC/Java [14]. It generates a large number of candidate invariants and repeatedly invokes the ESC/Java checker to remove unprovable annotations, until no more annotations are refuted. The problem of program invariant generation is related to the problem of automatic mining of temporal specifications of programs. Previous work [4, 32, 30, 33, 3, 1, 17] have approached this problem using both dynamic and static analysis techniques. The above mentioned techniques mostly focuses on generation of traditional specifications. Our approach is the first one to infer likely deterministic specifications of parallel programs. Unlike traditional specifications, our inferred specifications relate two program states coming from different executions.

A number of ongoing research efforts aim to make parallel programs deterministic by construction [29, 23, 19, 2, 21, 7]. 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, such as hard-to-use type systems, that can hinder general purpose programming.

Sadowski, et al., [27] propose a strict notion of determinism where they require the final output to be bitwise equal; therefore, their notion could not support semantic determinism. Kendo [26] proposes deterministic thread scheduling for race free programs. DMP [10] proposes hardware support for deterministic parallel execution.

2. BACKGROUND

In this section, we review the key features of our previously proposed deterministic specifications [8], quoting liberally from [8].

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 proposed 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
}
```

Semantic Determinism.

We observed [8] that the above deterministic specification is often too conservative. For example, consider the following example, where `A`, `B`, and `C` are floating-point matrices:

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

Floating-point addition and multiplication being non-associative due to rounding error, 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 relaxed 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.

This relaxed notion of determinism is useful in many contexts. Consider the following example which adds in parallel two items to a synchronized set represented internally as a red-black tree:

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

Here 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. 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.

Definition of Deterministic Specification.

In summary, we previously proposed [8] the following construct for the specification of deterministic behavior.

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

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.

More formally, let $P(s_0, \sigma)$ denote the resulting program state if we run procedure P on initial state s_0 and with thread schedule σ. Then, the above deterministic specification states that:

$$\forall s_0, s'_0, \sigma, \sigma'.\ \texttt{Pre}(s_0, s'_0) \implies \texttt{Post}(P(s_0, \sigma), P(s'_0, \sigma'))$$

We abbreviate this condition by:

$$\texttt{Pre} \implies_P \texttt{Post}$$

Note that, technically, only certain thread schedules are possible for each initial program state. That is, schedules σ should not be universally quantified, but must come from the set $\Sigma(s_0)$ of thread schedules for procedure P realizable from program state s_0. And function $P(s_0, \sigma)$ is defined only for $\sigma \in \Sigma(s_0)$. For simplicity, however, we omit any further references to $\Sigma(s_0)$.

Note also that, for certain initial states s_0 and thread schedules σ, procedure P may not terminate. In this case, function $P(s_0, \sigma)$ is not defined, as there is no resulting program state. We implicitly quantify over only terminating executions. Thus, our deterministic specifications are *partial*.

The advantage of our deterministic specifications 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. But, a simple deterministic specification enables us to use automated technique to check for parallelism bugs, such as harmful data races causing semantically nondeterministic behavior.

3. OVERVIEW OF DETERMIN

In this section, we give an informal overview of our algorithm for dynamically inferring likely deterministic specifications. Consider a procedure, bestTree, which, given a collection of DNA sequences, computes in parallel a most likely phylogenetic tree[1]:

```
void bestTree(int N, int[][] dna,
              int &score, int[] &tree)
{
    // Parallel branch-and-bound search
    // (with N threads) for an optimal
    // tree given DNA sequences.
    ...
}
```

Given two runs of this procedure on identical DNA sequence data, we would expect to get identical final likelihood scores. Because the procedure is a parallel branch-and-bound search, we cannot expect two different runs to necessarily compute the same phylogenetic tree—for the input data, there may be multiple trees with the same best score. Thus, we might manually specify the deterministic behavior of procedure bestTree as:

[1] A tree showing suspected evolutionary relationships—i.e. shared common ancestry—among a group of species or individuals.

```
void bestTree(int N, int[][] dna,
              int &score, int[] &tree)
{
    deterministic assume (dna == dna') {
        ...
    } assert (score == score');
}
```

Data Collection.

To infer a deterministic specification for procedure `bestTree`, we must first collect some sample of representative executions. Suppose the programmers who wrote the code have also constructed two DNA sequence data sets, D_1 and D_2, which they use for testing. (In the absence of hand-constructed test inputs, we could potentially use random or symbolic test generation to construct test cases.) Then, suppose we execute the procedure, perhaps as part of some existing application or test, twice on each input:

$$\begin{array}{lcl} N = 10,\ \mathit{dna} = D_1 & \longmapsto & \mathit{score} = 140,\ \mathit{tree} = t_1 \\ N = 10,\ \mathit{dna} = D_1 & \longmapsto & \mathit{score} = 140,\ \mathit{tree} = t_2 \\ N = 10,\ \mathit{dna} = D_2 & \longmapsto & \mathit{score} = 175,\ \mathit{tree} = t_3 \\ N = 10,\ \mathit{dna} = D_2 & \longmapsto & \mathit{score} = 175,\ \mathit{tree} = t_4 \end{array}$$

Specification Inference.

In theory, there are infinitely many possible bridge predicates relating pairs of inputs $(N, \mathit{dna}), (N', \mathit{dna}')$ or pairs of outputs $(\mathit{score}, \mathit{tree}), (\mathit{score}', \mathit{tree}')$. But we care only about a very restricted subset of these bridge predicates: predicates that compare only individual components across pairs of inputs or outputs, and that compare those components only for certain types of equality or approximate equality.

For example, for procedure `bestTree`, we are interested only in four bridge predicates as preconditions:

$$\mathit{true}, \quad N = N', \quad \mathit{dna} = \mathit{dna}', \quad N = N' \wedge \mathit{dna} = \mathit{dna}'$$

and four bridge predicates as postconditions:

$$\mathit{true},\ \mathit{score} = \mathit{score}',\ \mathit{tree} = \mathit{tree}',\ \mathit{score} = \mathit{score}' \wedge \mathit{tree} = \mathit{tree}'$$

More generally, by focusing on equality between components of input and output states, we only have to consider finitely-many possible deterministic specifications. (Although the number of possible specifications is still exponential in the number of input and output variables.)

Thus, we can think of the deterministic specification inference problem as having two parts. First, we should determine which of these possible deterministic specifications is *consistent* with our observed executions. Second, we must decide which of the consistent specifications to select as the final inferred specification.

There are six possible deterministic specifications consistent with the four above observed executions. Four of these specifications are of the form `Pre` $\Longrightarrow_{\texttt{bT}}$ *true*—that is, each of the four possible preconditions paired with the trivial postcondition. The other two possible deterministic specifications are:

$$(\mathit{dna} = \mathit{dna}) \ \Longrightarrow_{\texttt{bT}} \ (\mathit{score} = \mathit{score}') \qquad (1)$$

$$(N = N' \wedge \mathit{dna} = \mathit{dna}) \ \Longrightarrow_{\texttt{bT}} \ (\mathit{score} = \mathit{score}') \qquad (2)$$

In selecting one of these six potential deterministic specifications, we are guided by two principles: (1) First, we should select a specification with as strong of a postcondition as possible. Some parts of a procedures output may be scheduler-dependent and nondeterministic, but we would ideally like a specification that captures all parts of the output that are deterministic. (2) Second, for a given postcondition, we should select as weak of a precondition as possible.

For our running example, two of the possible specifications (i.e. specifications (1) and (2) shown above) have the strongest consistent postcondition $\mathit{score} = \mathit{score}'$. (Of course, no consistent postconditions contain $\mathit{tree} = \mathit{tree}'$ because we observed executions with identical inputs but different final values of *tree*.) Selecting the weaker of the two possible consistent preconditions for $\mathit{score} = \mathit{score}'$ gives us the deterministic specification:

$$(\mathit{dna} = \mathit{dna}) \ \Longrightarrow_{\texttt{bT}} \ (\mathit{score} = \mathit{score}')$$

For this example, the inferred deterministic specification is exactly the one we would have manually written. In general, however, there is always the danger that we will infer a postcondition that is too strong because we have observed no executions showing the nondeterminism of some output. Similarly, we may infer a precondition that is too weak because we have observed no executions showing that the deterministic behavior depends on a particular input. In the end, we must rely on having a sufficiently representative set of test inputs and running on sufficiently-many possible thread schedules to defend against inferring inaccurate deterministic specifications.

4. INFERRING DETERMINISTIC SPECS

In this section, we formally describe the problem of inferring deterministic specifications. Let P be a procedure that executes atomically and with internal parallelism. A procedure P in a given program is *atomic* [16] if, no other component of the program that can run in parallel with P can interfere with the execution of P. We say that procedure P has *internal parallelism* if, when P is executed, P performs a computation in parallel and P returns only after all parallel work has completed. For example, P may spawn several threads, but must join all of the threads before returning.

For the body of a procedure P, we want to infer a deterministic specification of the form:

```
void $P$() {
    deterministic assume(Pre($s_0, s_0'$)) {
        ... body of $P$ ...
    } assert(Post($s, s'$));
}
```

4.1 Deterministic Specification Model

In theory, the pre- and postconditions in a deterministic specification can be arbitrary bridge predicates. We restrict our attention, however, to a specific class of bridge predicates: conjunctions of *semantic equality* predicates.

We treat programs as having a finite set M of disjoint memory locations $\{m_1, \ldots, m_k\}$. Then, a program state s is a mapping from these global variables m_i to values $s(m_i)$ from set V of possible program values.

We further suppose that we have a finite set *EQ* of semantic equality predicates on program values. We require these predicates to be reflexive and symmetric relations on program values, and that this set include the strict equality predicate $v = v'$. (In our implementation, for example, we also include a approximate numeric equality $|v - v'| \leq \epsilon$ and semantic object equality $v.\texttt{equals}(v')$.)

Then, we consider the class of bridge predicates characterized by subsets of $M \times \mathit{EQ}$. For some $X \subseteq M \times \mathit{EQ}$, we define a bridge predicate φ_X by:

$$\varphi_X(s, s') \ = \bigwedge_{(m,\mathit{eq}) \in X} \mathit{eq}\Big(s(m),\, s'(m)\Big)$$

That is, for each pair of memory location m and equality predicate *eq* we compare the value of m from states s and s' using *eq*. The bridge predicate is the conjunction of all such equality predicates.

We justify this restriction by noting that this class of bridge predicates sufficed to manually specify the natural deterministic behavior of the benchmarks examined in our previous work [8].

An advantage of this restriction is that there exist only finitely many bridge predicates—one for each of the $2^{|M||EQ|}$ subsets of $M \times EQ$. Thus, there are only $2^{2|M||EQ|}$ possible deterministic specifications, consisting of one pre- and one postcondition, for a procedure P.

4.2 Specification Inference Problem

As described above, every pair of subsets of $M \times EQ$ defines a possible deterministic specification. For a given procedure P, many of these possible specifications may be true. That is, there may be many $pre, post \subseteq M \times EQ$ for which $\varphi_{pre} \Longrightarrow_P \varphi_{post}$.

Here we formally describe which of these true specifications, for a procedure P, we believe is the most natural and interesting choice. In short, we should infer specifications $\varphi_{pre} \Longrightarrow_P \varphi_{post}$ only where φ_{post} is the *strongest liberal postcondition* of φ_{pre} and φ_{pre} is a *weakest liberal precondition* of φ_{post}. Further, of such specifications, we should return one with the unique, strongest possible postcondition φ_{post}. (We will show that such a unique, strongest postcondition must exist.)

Lattice Structure of Deterministic Specifications.

The subsets of $M \times EQ$ naturally form a complete lattice under the ordering $\subseteq$ and with join $\cup$.

This induces a complete lattice on bridge predicates φ_X, with:

$$\varphi_X \sqsubseteq \varphi_Y \iff X \supseteq Y\,, \quad \varphi_X \sqcap \varphi_Y \;=\; \varphi_{X \cup Y}$$

Note that the lattice on predicates is reversed—*larger* sets yield *smaller* predicates, and the *meet* of two predicates is the *join* of the corresponding sets. This lattice has least and greatest elements:

$$\begin{aligned} \bot \;&=\; \varphi_{M \times EQ} &&= \forall (m, eq) \in M \times EQ.\ eq(s(m), s'(m)) \\ \top \;&=\; \varphi_{\emptyset} &&= \mathit{true} \end{aligned}$$

Note that, because every element of *EQ* is reflexive and symmetric, every predicate φ_X is reflexive and symmetric on program states. In particular, $\bot(s, s)$ for any state s.

We now state several simple but important properties of these lattices and their relation to the validity of deterministic specifications.

PROPOSITION 1. *The lattice operations $\sqcap$ and $\sqsubseteq$ on bridge predicates are exactly logical conjunction and implication:*

$$\begin{aligned} \varphi_X \wedge \varphi_Y \quad &= \quad \varphi_X \sqcap \varphi_Y \quad (=\ \varphi_{X \cup Y}) \\ \varphi_X \implies \varphi_Y \quad &\iff \quad \varphi_X \sqsubseteq \varphi_Y \quad (=\ X \supseteq Y) \end{aligned}$$

PROPOSITION 2. *Relation $\Longrightarrow_P$ distributes over the meet ($\sqcap$) operation on bridge predicates, and the join operation on subsets of $M \times EQ$, in the sense that:*

$$\varphi_X \Rightarrow_P \varphi_Y \;\;\wedge\;\; \varphi_X \Rightarrow_P \varphi_{Y'} \quad\iff\quad \varphi_X \Rightarrow_P (\varphi_Y \sqcap \varphi_{Y'})$$

or, equivalently:

$$\varphi_X \Rightarrow_P \varphi_Y \;\;\wedge\;\; \varphi_X \Rightarrow_P \varphi_{Y'} \quad\iff\quad \varphi_X \Rightarrow_P \varphi_{Y \cup Y'}$$

PROPOSITION 3. *Relation $\Longrightarrow_P$ is monotone in its second argument and anti-monotone in its first argument with respect to the lattice on bridge predicates:*

$$\begin{aligned} &\varphi_X \implies \varphi_{X'},\ \varphi_Y \implies \varphi_{Y'} \\ &\qquad\implies \quad (\varphi_{X'} \Rightarrow_P \varphi_Y \implies \varphi_X \Rightarrow_P \varphi_{Y'}) \end{aligned}$$

In light of Proposition 3, we will say that a deterministic specification (φ_X, φ_Y) is *stronger* or *more strict* than another specification $(\varphi_{X'}, \varphi_{Y'})$—denoted $(\varphi_X, \varphi_Y) \sqsubseteq (\varphi_{X'}, \varphi_{Y'})$—when $\varphi_{X'} \implies \varphi_X$ and $\varphi_Y \implies \varphi_{Y'}$.

Strongest Liberal Postcondition.

For any precondition φ_{pre} for a procedure P, we can define the *strongest liberal postcondition* $\texttt{SLP}_P(\varphi_{pre})$ of φ_{pre} as the least φ_{post} such that $\varphi_{pre} \Longrightarrow_P \varphi_{post}$. We show below that there is always a unique $\texttt{SLP}_P(\varphi_{pre})$.

PROPOSITION 4. *Let φ_{pre} be a precondition for procedure P.*

$$\mathit{SLP}_P(\varphi_{pre}) \quad = \quad \bigsqcap \{\varphi_{post} \mid \varphi_{pre} \Longrightarrow_P \varphi_{post}\}$$

PROOF. First, note that $\varphi_{pre} \Longrightarrow_P \top$ so the meet in the proposition is over a non-empty set. Let $\varphi_{\texttt{slp}}$ denote the meet over all postconditions that follow from φ_{pre}.

Then, $\varphi_{pre} \Longrightarrow_P \varphi_{\texttt{slp}}$, because $\Longrightarrow_P$ distributes over $\sqcap$.

Further, $\varphi_{\texttt{slp}}$ is clearly the least φ_{post} such that $\varphi_{\mathrm{pre}} \Longrightarrow_P \varphi_{post}$, because it is the meet of all such postconditions. □

COROLLARY 5. *Operator SLP_P is monotone. That is, if $\varphi_X \implies \varphi_Y$, then $\mathit{SLP}_P(\varphi_X) \implies \mathit{SLP}_P(\varphi_Y)$.*

PROOF. Suppose $\varphi_X \implies \varphi_Y$.

Because $\varphi_Y \Longrightarrow_P \texttt{SLP}_P(\varphi_Y)$ and by the anti-monotonicity of $\Longrightarrow_P$, we have $\varphi_X \Longrightarrow_P \texttt{SLP}_P(\varphi_Y)$. Therefore, because $\texttt{SLP}_P(\varphi_X)$ is the strongest postcondition of φ_X, we have $\texttt{SLP}_P(\varphi_X) \implies \texttt{SLP}_P(\varphi_Y)$. □

By the monotonicity $\texttt{SLP}_P$, the strongest postcondition that holds for P under any possible precondition, is $\texttt{SLP}_P(\bot)$. Note that, equivalently, this unique strongest postcondition is the meet over all true postconditions:

$$\bigsqcap \{\varphi_{post} \mid \exists \varphi_{pre}.\ \varphi_{pre} \Longrightarrow_P \varphi_{post}\}$$

Thus, in particular, postcondition $\texttt{SLP}_P(\bot)$ is the conjunction of the most individual equality predicates of any true postcondition.

Weakest Liberal Precondition.

We can similarly define the *weakest liberal precondition* of a postcondition φ_{post}. However, because we restrict our preconditions and postconditions to be conjunctions of equality predicates on individual memory locations, there may not be a unique weakest (or largest) precondition for a φ_{post}. Thus, we must define $\texttt{WLP}_P(\varphi_{post})$ to be the set of all weakest liberal preconditions:

DEFINITION 6. *$\varphi_{pre} \in \mathit{WLP}_P(\varphi_{post})$—i.e. is a weakest liberal precondition of φ_{post}—if and only if both:*

1. *$\varphi_{pre} \Longrightarrow_P \varphi_{post}$, and*
2. *If there exists a φ' such that $\varphi' \Longrightarrow_P \varphi_{post}$ and $\varphi_{pre} \implies \varphi'$, then $\varphi' = \varphi_{pre}$.*

Inferred Specification.

With these formal definitions, we can say that the deterministic specification inference problem for a procedure P is to compute, or to approximate as closely as possible, a deterministic specification $\varphi_{pre} \Longrightarrow_P \varphi_{post}$ where $\varphi_{post} = \texttt{SLP}_P(\bot)$ is the unique strongest possible postcondition for any precondition and where φ_{pre} is a weakest liberal precondition of φ_{post}.

Algorithm 1 Infer a likely deterministic specification given a set R of executions of procedure P.

```
1: $\texttt{Post}_R \leftarrow \texttt{SLP}_{P,R}(\bot)$
2: $\texttt{Pre}_R \leftarrow \texttt{WLP}_{P,R}(\texttt{Post}_R)$
3: return $(\texttt{Pre}_R, \texttt{Post}_R)$
```

5. DETERMIN ALGORITHM

In the previous section, we have defined the set of strongest true deterministic specifications for a given procedure P. When inferring a deterministic specification from a limited number of executions of a procedure P, however, we can only approximate the procedure's true specification.

Suppose we have a finite set R of observed executions $\{(s_1,\sigma_1,t_1),\ldots,(s_n,\sigma_n,t_n)\}$ of procedure P, where each t_i is the state $P(s_i,\sigma_i)$ resulting from executing P from initial state s_i on thread schedule σ_i. A deterministic specification $(\texttt{Pre},\texttt{Post})$ is satisfied *for the observed executions* R, which we abbreviate $\texttt{Pre} \Longrightarrow_{P,R} \texttt{Post}$, when:

$$\forall_{1 \le i,j \le n}.\ \texttt{Pre}(s_i,s_j) \implies \texttt{Post}(t_i,t_j)$$

Note that this definition is identical to that of $\Longrightarrow_P$, except that we only universally quantify over the observed inputs and thread schedules. We can similarly define the strongest liberal postcondition $SLP_{P,R}$ and weakest liberal preconditions $WLP_{P,R}$ over observed executions R.

Our overall inference algorithm is presented in Algorithm 1. Given a set of executions R of a procedure P, we will infer a likely deterministic specification $(\texttt{Pre}_R, \texttt{Post}_R)$.

The algorithm consists of two stages. First, we infer $\texttt{Post}_R$ by computing $\texttt{SLP}_{P,R}(\bot)$, the strongest liberal postcondition, given executions R of P, of precondition $\bot$. Recall that this is the strongest possible postcondition, given executions R, for any precondition. Second, we infer $\texttt{Pre}_R$ by computing $\texttt{WLP}_{P,R}(\texttt{Post}_R)$, a weakest liberal precondition, given executions R of P, of postcondition $\texttt{Post}_R$.

5.1 Computing the Strongest Postcondition

Algorithm 2 computes the strongest liberal postcondition, given executions R, of some φ_{pre}. The algorithm iterates over every pair of executions $(s_i,\sigma_i,t_i),(s_j,\sigma_j,t_j)$ that satisfy φ_{pre}. For each such pair, it computes the set of all individual equality predicates that hold on the resulting program states. The algorithm accumulates into *post* the intersection of all these sets. Thus, at the end of the algorithm, φ_{post} is the conjunction of all equality predicates that hold for pairs of post-states resulting from pre-states matching φ_{pre}. That is, φ_{post} is the strongest liberal postcondition of φ_{pre} for the observed executions R.

Checking the condition at line 3 and computing the set and the intersection in line 4 can all be done in $O(|M|\,|\mathit{EQ}|)$ time. Thus, the whole SLP computation requires $O(|M|\,|\mathit{EQ}|\,|R|^2)$ time.

5.2 Computing a Weakest Precondition

Algorithm 3 computes a weakest liberal precondition, given executions R, for some φ_{post}. The algorithm begins with $\varphi_{\mathit{pre}} = \bot = \varphi_{M \times \mathit{EQ}}$, and then greedily weakens φ_{pre} until it can be made no weaker while remaining a precondition for φ_{post} on the observed executions R. Lines 3-5 check if the current φ_{pre} can be safely weakened by removing the conjunct $\mathit{eq}(s(m),s'(m))$ from $\varphi_{\mathit{pre}}(s,s')$.

It is sufficient to consider each (m,eq) only once during the computation. Suppose it was not possible to weaken some pre_1 by removing (m,eq), but it was possible to weaken a later pre_2 by removing the same (m,eq). Because pre_2 comes later, $\mathit{pre}_1 \supseteq \mathit{pre}_2$ and thus $(\mathit{pre}_1 - \{(m,\mathit{eq})\}) \supseteq (\mathit{pre}_2 - \{(m,\mathit{eq})\})$ But, then if $\varphi_{\mathit{pre}_2-\{(m,\mathit{eq})\}} \Longrightarrow_{P,R} \varphi_{\mathit{post}}$, we must also have $\varphi_{\mathit{pre}_1-\{(m,\mathit{eq})\}} \Longrightarrow_{P,R} \varphi_{\mathit{post}}$, which is a contradiction.

Algorithm 2 Compute $\texttt{SLP}_{P,R}(\varphi_{\mathit{pre}})$

```
1: $\mathit{post} \leftarrow M \times \mathit{EQ}$
2: for all $(s_i,\sigma_i,t_i),(s_j,\sigma_j,t_j) \in R \times R$ do
3:     if $\varphi_{\mathit{pre}}(s_i,s_j)$ then
4:         $\mathit{post} \leftarrow \mathit{post} \cap \{(m,\mathit{eq}) \mid \mathit{eq}(t_i(m),t_j(m))\}$
5:     end if
6: end for
7: return $\varphi_{\mathit{post}}$
```

Algorithm 3 Compute a $\texttt{WLP}_{P,R}(\varphi_{\mathit{post}})$

```
Require: $\bot \Longrightarrow_{P,R} \varphi_{\mathit{post}}$
1: $\mathit{pre} \leftarrow M \times \mathit{EQ}$
2: for all $(m,\mathit{eq}) \in M \times \mathit{EQ}$ do
3:     if $\varphi_{\mathit{pre}-\{(m,\mathit{eq})\}} \Longrightarrow_{R,P} \varphi_{\mathit{post}}$ then
4:         $\mathit{pre} \leftarrow \mathit{pre} - \{(m,\mathit{eq})\}$
5:     end if
6: end for
7: return $\varphi_{\mathit{pre}}$
```

Note that, depending on the order in which the algorithm considers the elements of $M \times \mathit{EQ}$, it can return any of the possible weakest preconditions of φ_{post} under the observed executions.

Checking the condition at line 3 requires $O(|M|\,|\mathit{EQ}|\,|R|^2)$ time, to determine that $\varphi_{\mathit{pre}-\{(m,\mathit{eq})\}} \implies \varphi_{\mathit{post}}$ on every pair of observed executions. Thus, the entire computation of a $\texttt{Pre}_R$ requires $O(|M|^2\,|\mathit{EQ}|^2\,|R|^2)$ time.

5.3 Correctness

We now formally state several important properties of our deterministic specification inference algorithm. To conserve space, we omit the proofs of these statements—the proofs can be found in the accompanying technical report.

Most importantly, we show in Proposition 7 that the DETERMIN algorithm is correct. That is, for any inferred deterministic specification $(\texttt{Pre}_R, \texttt{Post}_R)$ for executions R of procedure P:

1. $\texttt{Pre}_R$ is a weakest liberal precondition for $\texttt{Post}_R$ and $\texttt{Post}_R$ is a strongest liberal postcondition for $\texttt{Pre}_R$, given the executions in R.

2. $\texttt{Post}_R$ is the unique strongest liberal postcondition for any possible precondition given the executions in R.

We further show (Corollary 9) that an inferred postcondition $\texttt{Post}_R$ will always be stronger than the strongest true postcondition $\texttt{SLP}_P(\bot)$. And the more executions R we observe, the weaker—i.e. closer to the true strongest postcondition—our inferred postcondition will be (Proposition 8).

Example 10 shows that we cannot make analogous guarantee for our inferred precondition $\texttt{Pre}_R$. Rather, we can only guarantee that additional executions will only strengthen the inferred precondition *as long as they do not weaken the postcondition* $\texttt{Post}_R$ (Propositions 11 and 12). And, if $\texttt{Post}_R$ is the true strongest postcondition for any precondition and for all executions, then as we observe additional executions our stronger and stronger inferred $\texttt{Pre}_R$ will approach a true weakest precondition for $\texttt{Post}_R$ (Corollaries 13 and 14).

PROPOSITION 7. *Let* $(\mathtt{Pre}_R, \mathtt{Post}_R)$ *be the specification inferred for executions R of P. Then,* $\mathtt{Pre}_R \in \mathtt{WLP}_{P,R}(\mathtt{Post}_R)$ *and* $\mathtt{Post}_R = \mathtt{SLP}_{P,R}(\mathtt{Pre}_R)$.

Further, for any $\varphi_{pre} \Longrightarrow_{P,R} \varphi_{post}$, we have $\texttt{Post}_R \implies \varphi_{post}$.

PROPOSITION 8. *Let $\texttt{Post}_R$ and $\texttt{Post}_{R'}$ be the inferred postconditions for $R \subseteq R'$. Then, $\texttt{Post}_R \implies \texttt{Post}_{R'}$.*

COROLLARY 9. *Let $\texttt{Post}_R$ be the inferred postconditions for observed executions R. Then, $\texttt{Post}_R \implies \texttt{SLP}_P(\bot)$. That is, $\texttt{Post}_R$ is stronger than the strongest true postcondition.*

EXAMPLE 10. *Consider the following contrived procedure operating on two global variables x and y:*

```
example() {
    <x = x + 1> || <y = 0> || <y = y + 1>;
}
```

Procedure `example` runs three atomic statements in parallel: an increment of x, an assignment of y to zero, and an increment of y. Suppose we observe the executions:

$$\begin{array}{lcl} x=0,\ \ y=0 & \longmapsto & x=1,\ \ y=0 \\ x=0,\ \ y=1 & \longmapsto & x=1,\ \ y=1 \\ x=1,\ \ y=1 & \longmapsto & x=2,\ \ y=0 \end{array}$$

Then, we will infer the specification precondition $x = x' \wedge y = y'$ and postcondition $x = x' \wedge y = y'$.

But, suppose we observe the additional execution:

$$x=0,\ \ y=0 \ \longmapsto\ x=1,\ \ y=1$$

Then we will see that $y = y'$ cannot be guaranteed, and we will infer the true specification $x = x' \Longrightarrow_{\texttt{example}} x = x'$, which has a weaker precondition.

PROPOSITION 11. *Let $\texttt{Post}$ be the inferred postcondition for both R and R', with $R \subseteq R'$. Further, let $\texttt{Pre}_R$ be an inferred precondition under R. Then, there is no strictly weaker inferred precondition $\texttt{Pre}'_R$.*

PROPOSITION 12. *Let $\texttt{Post}$ be the inferred postcondition for both R and R', with $R \subseteq R'$. Further, let $\texttt{Pre}_{R'}$ be an inferred precondition under R'. Then, there is a $\texttt{Pre}_R$ from $\texttt{WLP}_{P,R}(\texttt{Post})$—i.e. a possible inferred precondition for observed executions R— such that $\texttt{Pre}_{R'} \implies \texttt{Pre}_R$.*

COROLLARY 13. *Let the postcondition inferred for executions R be $\texttt{Post} = \texttt{SLP}_P(\bot)$. Further, let $\texttt{Pre}_R$ be an inferred precondition under R. Then, there are no true preconditions of $\texttt{Post}$, i.e. elements of $\texttt{WLP}_P(\texttt{Post})$, strictly weaker than $\texttt{Pre}_R$.*

COROLLARY 14. *Let the postcondition inferred for executions R be $\texttt{Post} = \texttt{SLP}_P(\bot)$. Further, let $\texttt{Pre}$ be a true precondition for $\texttt{Post}$. Then, there is a $\texttt{Pre}_R$ from $\texttt{WLP}_{P,R}(\texttt{Post})$—i.e. a possible inferred precondition under observed executions R— such that $\texttt{Pre} \implies \texttt{Pre}_R$.*

5.4 A More Conservative Precondition

Our algorithm for computing a precondition from $\texttt{WLP}_{P,R}(\texttt{Post}_R)$ finds a weakest liberal precondition $\texttt{Pre}_R$ such that no pair of executions from R falsifies $\texttt{Pre}_R \Longrightarrow_P \texttt{Post}_R$. When only a small number of executions or procedure inputs are examined, such a precondition may be too weak.

For example, consider a procedure P whose input consists of ten integers $x_0, \ldots, x_9$ and whose output is the sum *sum* of the integers. Suppose we observe executions R of this method from only two distinct initial states—one where $x_0 = \cdots = x_9 = 0$ and one where $x_0 = \cdots = x_9 = 1$. Then, the deterministic specification $x_3 = x'_3 \Longrightarrow_{P,R} sum = sum'$ is consistent with the data. That is, we observe no pair of executions that falsifies that $x_3 = x'_3$ is a necessary precondition for determinism—i.e. a pair in which $x_3 = x'_3$, but because some other input is not equal, the final *sum*s are not equal.

Algorithm 4 Compute a $\texttt{WLOP}_{P,R}(\varphi_{post})$

```
Require: $\bot \Longrightarrow_{P,R} \varphi_{post}$
// Find the occurring preconditions of $\varphi_{post}$.
$occurs \leftarrow \emptyset$
for all $(s_i, \sigma_i, t_i), (s_j, \sigma_j, t_j) \in R \times R$ do
    $pre \leftarrow \{(m, eq) \mid eq(s_i(m), s_j(m))\}$
    if $\varphi_{pre} \Longrightarrow_{P,R} \varphi_{post}$ then
        $occurs \leftarrow occurs \cup \{pre\}$
    end if
end for
// Select a weakest occurring precondition of $\varphi_{post}$.
for all $pre \in occurs$ do
    if $\neg\exists pre' \in occurs.\ pre' \subseteq pre$ then
        return $\varphi_{pre}$
    end if
end for
```

To combat such an inadequate test set, rather than report any weakest liberal precondition consistent with out observed executions, we can report a *weakest* ***occurring*** *liberal precondition*.

DEFINITION 15. *We say that precondition φ_{pre}* ***occurs*** *in a set R of observed executions iff there is a pair $(s_i, \sigma_i, t_i), (s_j, \sigma_j, t_j)$ from R, with $i \neq j$, such that φ_{pre} is the strongest bridge predicate satisfied by s_i and s_j. That is, pre is the set $\{(m, eq) \mid eq(s_i(m), s_j(m))\}$.*

We define the set $\texttt{WLOP}_{P,R}(\varphi_{post})$ of weakest liberal occurring preconditions for P of φ_{post} under observed executions R by:

DEFINITION 16. *$\varphi_{pre} \in \texttt{WLOP}_{P,R}(\varphi_{post})$ iff:*

1. *$\varphi_{pre} \Longrightarrow_{P,R} \varphi_{post}$,*
2. *φ_{pre} occurs in R, and*
3. *If φ'_{pre} occurs in R and $\varphi'_{pre} \Longrightarrow_{P,R} \varphi_{post}$ and $\varphi_{pre} \implies \varphi'_{pre}$, then $\varphi'_{pre} = \varphi_{pre}$.*

Algorithm 4 computes an element of $\texttt{WLOP}_{P,R}$ for a postcondition. We can compute an occurring weakest precondition $\texttt{Pre}_R$ by applying Algorithm 4 to $\texttt{Post}_R$.

Note that, unlike with a $\texttt{WLP}$, observing additional executions may strengthen or weaken $\texttt{WLOP}_{P,R}(\texttt{Post}_R)$, even if $\texttt{Post}_R$ does not change. This is because additional observations can now provide a weaker occurring precondition, in addition to falsifying a previous weakest precondition. However, in the limit of observing all possible executions of P, there is clearly no difference between $\texttt{WLOP}_{P,R}(\texttt{Post}_R)$ and $\texttt{WLP}_{P,R}(\texttt{Post}_R)$.

6. EVALUATION

In this section, we describe our efforts to experimentally evaluate the effectiveness of our algorithm for inferring likely deterministic specifications. We aim to show that, given a small number of representative executions, our algorithm can infer correct and useful deterministic specifications. That is, that our inferred specifications capture the intended natural deterministic behavior of parallel programs.

To evaluate these claims, we implemented our specification inference algorithm DETERMIN for Java applications and applied DETERMIN to the benchmarks to which we previously had manually added deterministic specifications in [8]. We then compared the quality and accuracy of the inferred and manual specifications.

6.1 Benchmarks

We evaluate DETERMIN on the benchmarks previously examined in [8]. These benchmarks are primarily from the Java Grande Forum (JGF) benchmark suite [11] and the Parallel Java (PJ) library [20]. The names and sizes of the benchmarks are given in Table 1. Benchmark `tsp` is a parallel Traveling Salesman branch-and-bound search [31]. The JGF benchmarks include five parallel computation kernels—for successive order-relaxation (`sor`), sparse matrix-vector multiplication (`sparsematmult`), computing the coefficients of a Fourier series (`series`), encryption and decryption (`crypt`), and LU factorization (`lufact`)—as well as a parallel molecular dynamic simulator (`moldyn`), ray tracer (`raytracer`), and Monte Carlo stock price simulator (`montecarlo`). The Parallel Java (PJ) benchmarks include an app for computing a Monte Carlo approximation of π (`pi3`), an app for cryptographic cracking a cryptographic key (`keysearch3`), an app for parallel rendering of a Mandelbrot Set image (`mandelbrot`), and a parallel branch-and-bound search for an optimal phylogenetic tree (`phylogenetic`). These benchmarks range from a few hundred to a few thousand lines of code, with the PJ benchmarks relying on an additional roughly 15,000 lines of library code from the Parallel Java Library for threading, synchronization, and other functionality.

In [8], we added a single deterministic specification block to each benchmark, around the benchmark's entire parallel computation.

6.2 Methodology

In order to apply the DETERMIN algorithm to these benchmarks, we need: (1) to decide for which regions in each benchmark to infer deterministic specifications, (2) to select a set of representative executions R of these regions as inputs to DETERMIN, (3) to define the sets of *memory locations* M and *semantic equality predicates EQ* for the benchmarks.

Regions for Deterministic Specification Inference.

In this work, we have proposed inferring deterministic specifications for procedures—either for all procedures detected to have internal parallelism or for some set of user specified procedures. Our manual deterministic specifications in [8], however, were written not at procedure boundaries, but around certain hand-chosen syntactic blocks of code containing internal parallelism. (Each such block is atomic because it is the only region in its benchmark that performs a parallel computation.) Thus, to enable a fair and direct comparison, we use DETERMIN to infer deterministic preconditions and postconditions at the beginning and end of the single deterministic block manually identified in [8]. That is, in each representative execution we record the program state at the beginning and end of the manually identified deterministic block.

Representative Executions.

We similarly ran each PJ benchmark and `tsp` twenty times—ten on each of of two selected inputs, half with five threads and half with ten threads. Benchmark `tsp`, all of the JGF benchmarks, and many of the PJ benchmarks come with test inputs. When available, we used two of these test inputs. Otherwise, we constructed inputs by hand.

The representative executions were run under the Sun JDK 6 on an eight-core Intel Xeon 2GHz Linux system.

Note that, due to the small number of test inputs, we compute the more conservative weakest liberal *occurring* precondition (WLOP), described in Algorithm 4, for our inferred postcondition, rather than a weakest liberal precondition (WLP).

Memory Locations and Equality Predicates.

For the Java program states recorded during the representative executions, we generate a set M of memory locations by enumerating all paths of field dereferences, up to some fixed length, through the programs' memory graphs starting at the local variables and static classes. (For example, n, *this.results.bestScore*, or *AppClass.N_THREADS*.) For completeness, we considered all paths of length up to 8, yielding from roughly 20 to 150 memory locations for each benchmark.

We use several equality predicates to compare these memory locations: Primitive types are compared using strict equality or approximate equality (equal to within 10^{-10}) for floating-point values. Objects are compared using their *equals*() methods. Object arrays, Lists, and Iterables can be compared element-by-element or compared as sets of elements.

6.3 Implementation

To capture and record program states at desired points in our benchmarks, the data collection component of our implementation uses the Java Reflection API to traverse and serialize a running program's memory graph. We manually instrumented the local variables in scope at the open and close of each deterministic block.

The specification inference portion of our implementation takes a set of these serialized and pre- and post-states as input and outputs an inferred deterministic strongest liberal postcondition and weakest liberal occurring precondition. Both components together are implemented in roughly 1000 lines of Java code.

Heuristics.

The above approach generates a large number of memory locations and equality predicates, leading to deterministic specifications with too many conjuncts in their preconditions and postconditions. We employ several heuristics to decrease the size and increase the relevancy of our deterministic specifications:

First, we remove from the inferred postconditions any locations not modified in at least one execution by the region of code under examination. Without this heuristic, the strongest postcondition (and thus also the precondition) for a region will contain a conjunct $v = v'$ for each variable v not modified by the region. While such an added conjunct is correct—we can guarantee the determinism of variables that are not modified—it is generally not relevant to computation being performed. On each of our benchmarks, this heuristic removes roughly from 10 to 60 conjuncts.

Second, we remove from the inferred precondition and postcondition any conjuncts that are satisfied by every pair of observed program executions. These locations tend to be global constants, such as hard-coded parameters and Class objects. As above, predicates involving such constants are typically not relevant. On each our benchmarks, this heuristic can remove as many as 75 conjuncts from the precondition or postcondition.

Third, we eliminate redundant conjuncts. For example, if a precondition contains the conjunct $o.equals(o)$ for an array o, then we will not include the redundant, weaker conjunct $o.f.equals(o.f')$. Or if our postcondition contains conjunct $x = x'$, we will not add the redundant conjunct $|x - x'| \leq 10^{-10}$. On each our benchmarks, this heuristic removes only a handful of conjuncts from the final preconditions and postconditions.

6.4 Results

The results of our experimental evaluation are shown in Table 1. We will argue the these results provide evidence for our claims that DETERMIN can automatically infer deterministic specifications that are both accurate and useful.

	Benchmark	Approximate Lines of Code (App + Library)	Precondition			Postcondition		
			# Manual Conjuncts	# Inferred Conjuncts	As Strong As Manual?	# Manual Conjuncts	# Inferred Conjuncts	As Strong As Manual?
JGF	`sor`	300	3	2	No	1	7	Yes
	`sparsematmult`	700	4	4	No	1	2	Yes
	`series`	800	1	3	Yes	1	1	Yes
	`crypt`	1100	1	5	Yes	2	2	Yes
	`moldyn`	1300	2	14	Yes	3	7	Yes
	`lufact`	1500	4	9	Yes	3	3	No*
	`raytracer`	1900	2	3	Yes	1	1	Yes
	`montecarlo`	3600	1	2	Yes	1	1	Yes
PJ	`pi3`	150 + 15,000	2	3	Yes	1	1	Yes
	`keysearch3`	200 + 15,000	3	5	Yes	1	3	Yes
	`mandelbrot`	250 + 15,000	7	11	Yes	1	5	Yes
	`phylogeny`	4400 + 15,000	3	5	Yes	2	11	Yes
	`tsp`	700	1	3	Yes	1	2	Yes

Table 1: Results of our experimental evaluation of DETERMIN. For each benchmark, we report the approximate size of the benchmark and the number of conjunctions in the manual deterministic precondition and postcondition added to the benchmark in [8]. We also report the number of conjuncts in the strongest liberal postcondition (SLP) and weakest liberal occurring precondition (WLOP) of the deterministic specification inferred by DETERMIN for each benchmark. Further, we indicate whether each inferred precondition and postcondition is at least as strict as its corresponding hand-specified condition.

Accuracy: Postconditions.

For every benchmark but `lufact`, our automatically inferred postcondition was at least as strong as the corresponding manually-specified postcondition from [8]. Further, the inferred postcondition for `lufact` is actually more accurate than our manual one. When writing the manual specification for `lufact` in [8], we wrote postcondition $a = a' \wedge \mathit{ipvt} = \mathit{ipvt}' \wedge x = x'$. But, in fact, the `lufact` routine writes no output into variable x. The relevant output—the solution to the linear system being solved—is written to variable b. The correct postcondition, inferred by DETERMIN, is $a = a' \wedge \mathit{ipvt} = \mathit{ipvt}' \wedge b = b'$

Of the other benchmarks, for all but three of them (`sor`, `moldyn`, and `tsp`), the inferred postcondition is equivalent to the manual one. Although the inferred postconditions contain more conjuncts, these postconditions hold for the same pairs of executions. For example, the manual postcondition for `mandelbrot` is simply $\mathit{matrix} = \mathit{matrix}'$. That is, the resulting image, stored as a matrix of hues, is deterministic. The inferred postcondition also contains $\mathit{image.myWidth} = \mathit{image.myWidth}'$. But this field always holds the width of *matrix*, and thus this conjunct does not strictly strengthen the postcondition.

Further, for benchmarks `sor` and `moldyn`, the inferred postconditions are still correct and are only slightly stronger than the previous manual ones. Both benchmarks retain various intermediate results past the end of their computations. Roughly speaking, our manual assertions for these benchmarks specify that the final answer is independent of the number of threads used, while the inferred specifications capture that these intermediate results are also deterministic for any fixed number of threads.

Accuracy: Preconditions.

For all but two benchmarks (`sor` and `sparsematmult`), our inferred preconditions are also as strong as our previous [8] manual deterministic specifications. Further, these inferred preconditions, except for `moldyn`'s and `keysearch3`'s, are equivalent to the manual ones although they contain more conjuncts.

The inferred precondition for `moldyn` contains $\mathit{nthreads} = \mathit{nthreads}'$, making it stronger than in our manual specification.

The stronger precondition for `keysearch3` actually highlights an error in the manual specification from [8]. One of the inputs (*partialkey*) to the main computation is missing from the manual precondition. But the conjunct $\mathit{partialkey} = \mathit{partialkey}'$ correctly appear in the inferred precondition.

Limitations.

For the `sor` benchmark, our inferred precondition is missing two input parameters on which the deterministic behavior depends. DETERMIN fails to include these two parameters because they each take on the same value in all of JGF test inputs for `sor`. Thus, DETERMIN sees no evidence that these parameters are important for determinism and removes them via our second heuristic. This example shows the need for a sufficiently diverse set of test inputs and executions in order to infer accurate deterministic specifications.

Similarly, the postcondition for `tsp` is incorrectly too strong, requiring that two runs on the same input return the same tour. In fact, two such runs could return different tours with the same minimal cost, but our particular test inputs appear to have unique solutions.

Discussion.

For nearly all of our benchmarks, DETERMIN infers deterministic preconditions and postconditions equivalent to, slightly stronger than, or more accurate than those in our previous, manual specifications. Thus, we argue that DETERMIN can capture the intended and natural deterministic behavior of parallel programs.

Further, although our automatically inferred specifications are somewhat larger than the manual ones from [8], the total number of inferred conjuncts remains quite small. In particular, we believe that pre- and postconditions with 5 to 15 conjuncts are small enough to be fairly easily understood by a programmer. Thus, we argue that such inferred specifications can help document the deterministic behavior of a routine or application for a programmer. For example, the inferred specification for `lufact` corrected our misunderstanding of the benchmark's behavior.

Further, we argue that such automatically-inferred deterministic specifications can be useful in discovering parallelism bugs through anomaly detection. That is, from observing "normal" program executions, DETERMIN infers a specification of the typical, expected deterministic behavior of a program. Then, if more in-depth testing finds executions that are anomalous—i.e. that violate the inferred specification—then those executions may exhibit bugs.

In [8], we combined deterministic specifications with a parallel software testing tool in order to distinguish benign from harmful races in these benchmarks. The specifications inferred by DETERMIN in this work are sufficiently similar to those manual specifications to serve the same purpose. In particular, these specifications would allow us to distinguish the harmful data race that exists in the `raytracer` benchmark from the other benign races.

7. ACKNOWLEDGMENTS

We would like to thank Chang-Seo Park and our anonymous reviewers for their valuable comments. 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 Sun Microsystems and by matching funding from UC MICRO (Award #08-113), by NSF Grants CNS-0720906 and CCF-0747390, and by a DoD NDSEG Graduate Fellowship.

8. REFERENCES

[1] M. Acharya, T. Xie, J. Pei, and J. Xu. Mining API patterns as partial orders from source code: from usage scenarios to specifications. In *6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on the Foundations of Software Engineering*. ACM, 2007.

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

[3] R. Alur, P. Cerny, G. Gupta, P. Madhusudan, W. Nam, and A. Srivastava. Synthesis of Interface Specifications for Java Classes. In *Proceedings of POPL'05 (32nd ACM Symposium on Principles of Programming Languages)*, 2005.

[4] G. Ammons, R. Bodik, and J. R. Larus. Mining specifications. In *ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL)*, 2002.

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

[6] G. Barnes. A method for implementing lock-free shared-data structures. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 261–270, 1993.

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

[8] J. Burnim and K. Sen. 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*. ACM, 2009.

[9] C. Csallner, N. Tillmann, and Y. Smaragdakis. DySy: Dynamic symbolic execution for invariant inference. In *30th ACM/IEEE International Conference on Software Engineering (ICSE)*, 2008.

[10] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: deterministic shared memory multiprocessing. In *ACM conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

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

[12] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 449–458, June 2000.

[13] 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.

[14] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2002.

[15] C. Flanagan and R. M. Leino. Houdini, an annotation assistant for ESC/Java. In *Proceedings of the International Symposium of Formal Methods Europe (FME)*, 2001.

[16] C. Flanagan and S. Qadeer. Types for atomicity. In *ACM SIGPLAN international workshop on Types in Languages Design and Implementation (TLDI)*, 2003.

[17] C. Goues and W. Weimer. Specification mining with few false positives. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2009.

[18] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the International Conference on Software Engineering*, 2002.

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

[20] 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.

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

[22] F. Logozzo. Automatic inference of class invariants. In *Proceedings of the 5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI '04)*, January 2004.

[23] 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.

[24] M. Z. Malik, A. Pervaiz, , and S. Khurshid. Generating representation invariants of structurally complex data. In *TACAS*, pages 34–49, 2007.

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

[26] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: Efficient deterministic multithreading in software. In *The International Conference on Architectural Support for Programming Languages and Operating Systems*, Mar 2009.

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

[28] M. Taghdiri and D. Jackson. Inferring specifications to detect errors in code. *Automated Software Engg.*, 14(1):87–121, 2007.

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

[30] N. Tillmann, F. Chen, and W. Schulte. Discovering likely method specifications. In *ICFEM*, pages 717–736, 2006.

[31] C. von Praun and T. R. Gross. Object race detection. In *ACM SIGPLAN conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2001.

[32] J. Whaley, M. C. Martin, and M. S. Lam. Automatic Extraction of Object-Oriented Component Interfaces. In *Proceedings of ACM SIGSOFT ISSTA'02 (International Symposium on Software Testing and Analysis)*, 2002.

[33] J. Yang and D. Evans. Dynamically inferring temporal properties. In *PASTE '04*, pages 23–28. ACM, 2004.