Input-Covering Schedules for Multithreaded Programs

Tom Bergan Luis Ceze Dan Grossman
University of Washington, Department of Computer Science & Engineering

ABSTRACT

We propose constraining multithreaded execution to small sets of *input-covering schedules*, which we define as follows: given a program P, we say that a set of schedules Σ *covers* all inputs of program P if, when given any valid input, P's execution can be constrained to some schedule in Σ and still produce a semantically-valid result.

Our approach is to first compute a small Σ for a given program P, and then, at runtime, constrain P's execution to always follow some schedule in Σ, and never deviate. We have designed an algorithm that uses symbolic execution to systematically enumerate a set of input-covering schedules, Σ. To deal with programs that run for an unbounded length of time, we partition execution into *bounded epochs*, find input-covering schedules for each epoch in isolation, and then piece the schedules together at runtime. We have implemented this algorithm and a constrained execution runtime, and we report early results.

Our approach has the following advantage: because all possible runtime schedules are known *a priori*, we can seek to validate the program by thoroughly testing each schedule in Σ, in isolation, without needing to reason about the huge space of thread interleavings that arises due to conventional nondeterministic execution.

1. INTRODUCTION

Multithreaded programs are notoriously difficult to test and verify. In addition to the already daunting task of reasoning about program behavior over all possible inputs, testing and verification tools must reason about a large number of possible thread interleavings for each input—the number of possible interleavings grows exponentially with the length of a program's execution. Tools can systematically explore the interleaving space in part, but in practice, the interleaving space is too massive to be explored exhaustively.

We want to avoid this problem entirely by constraining execution to a small, easily enumerable set of *input-covering schedules*. Given a program P, we say that a set of schedules Σ *covers* the program's inputs if, for all valid inputs, there exists some schedule $S \in \Sigma$ such that P's execution can be constrained to S and still produce a semantically-valid result. In our system, a schedule is a partial order of dynamic instances of program statements paired with thread ids, *i.e.*, a happens-before graph.

It is not obvious that small sets of input-covering schedules should exist for realistic multithreaded programs. The key word is *small*—an input-covering set Σ is of no help when it is so intractably large that it cannot be enumerated in a reasonable time. A primary contribution of this work is defining Σ in a way that makes the problem more tractable. Notably, programs that run for unbounded periods of time require unboundedly many schedules, making the set Σ intractably large. We avoid this problem by partitioning execution into *bounded epochs*—we find input-covering schedules for each epoch in isolation, and then piece those schedules together at runtime.

1.1 System Overview

Our proposed system has three parts: an *algorithm* to enumerate Σ for a given program, a *runtime system* that constrains execution to Σ, and a *testing strategy* that exploits input-covering schedules to more efficiently find bugs or prove their absence. Each part is summarized below:

Algorithm. We have designed an algorithm that uses symbolic execution to systematically enumerate input-covering schedules for a given program. Figure 1 gives a demonstration. On the right side of Figure 1 is a set of input-covering schedules, Σ, that our algorithm might produce when given the program on the left. Each schedule in Σ is paired with an *input constraint* that describes the set of inputs under which the schedule can be followed. Schedules are specified as a happens-before ordering of synchronization statements.

Runtime System. At runtime, we constrain execution to always follow schedules in Σ. We have implemented a custom runtime system that captures the program's inputs, finds a pair $(I,S) \in \Sigma$ such that the program's inputs satisfy input constraint I, and then constrains execution to S, ensuring that execution never deviates from S.

Testing Strategy. Finally, and most importantly, testing and verification become simpler under the assumption that programs always execute using our custom runtime system. Given this assumption, we observe that the input-covering set Σ contains the *complete* set of schedules that might be followed at runtime. As a result, verification tools can focus on schedules in Σ only, avoiding the need to reason about a massive nondeterministic interleaving space.

For a simple example, consider deadlocks. We can determine if a schedule deadlocks by simply looking at it—if the schedule does not terminate with a program exit statement, then the schedule deadlocks. We can perform this check for each schedule in Σ independently. If a deadlocking schedule is found, we can use the schedule's associated input constraint to present the programmer with a concrete input and schedule that leads to deadlock. If no deadlocking schedules are found, we have *proven* that we will never encounter a deadlock when execution is constrained by our runtime system.

Deadlocks are relatively simple because they are purely schedule-dependent, meaning that their presence depends on the choice of schedule only, not on schedule-irrelevant choices of data values or thread-local paths. Our approach simplifies the search for other, more complex schedule-dependent bugs as well. For example, consider the following code:

```
1   input X
2   global Lock A,B
3
4   Thread 1                Thread 2
5   for (i in 1..5) {       for (i in 1..5) {
6     if (X == 0) {           if (X == 0) {
7       lock(A)                 lock(A)
8       unlock(A)               unlock(A)
9     } else {                } else {
10      lock(B)                 lock(B)
11      unlock(B)               unlock(B)
12    }                       }
13  }                       }
```

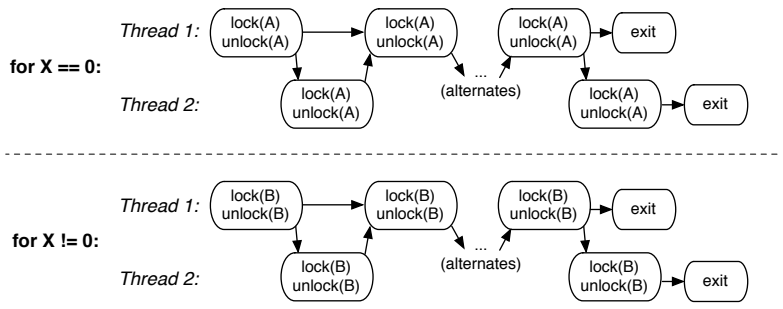

Figure 1: On the left is a simple multithreaded program. On the right is one set of input-covering schedules for the program.

Thread 1	Thread 2	Thread 3	Thread 4
lock(m[i])	lock(m[0])	lock(L)	lock(L)
d[0]++	d[0]++	if (x%2!=0)	if (x%2!=0)
unlock(m[i])	unlock(m[0])	x++	fail()
		unlock(L)	unlock(L)

There is a data race between threads T_1 and T_2 when T_1 executes with `i!=0`, and there is an assertion failure in T_4 when it executes before T_3 with an odd value for `x`. These bugs are difficult to find in conventional systems because they depend on specific combinations of input and schedule. Our approach uses a small number of schedules, so we can reason about each schedule in isolation, perhaps by serializing the original multithreaded program to each schedule in Σ to produce $|\Sigma|$ single-threaded programs (similarly to [8]). This reduces the worst-case number of possible program behaviors from $k! \cdot i$ to $|\Sigma| \cdot i$, where k is the length of execution and i is the number of possible inputs.

1.2 Paper Outline and Related Work

The primary contribution of this paper is the identification of the input-covering schedules problem, which to our knowledge has not been introduced previously. Our solution to this problem depends on a carefully determined representation of schedules that we describe in §2.

This paper describes a work-in-progress. We have designed an algorithm for finding input-covering schedules (§3), implemented that algorithm on the Cloud9 [2] symbolic execution engine, and implemented a runtime system to constrain execution to input-covering schedules produced by our algorithm (§4). Our implementation targets C programs that use pthreads. We have not yet explored testing strategies in any significant detail, though we have implemented a simple but imperfect deadlock checker that we describe in §4. We summarize early experimental results in §5.

Our system generalizes ideas introduced by Tern [4] and Peregrine [5]. Those systems memoize schedules from a few tested inputs, so they provide best-effort schedule memoization only, while our system enumerates a complete input-covering set. Computing complete input-covering sets requires solving a number of technical challenges not faced by any prior system. Another important difference between our work and [4, 5] is that we assume programs are data race free—this assumption simplifies our analysis in a number of important ways that we will mention later.

Our runtime system selects schedules deterministically for each input, giving our system all the benefits of determinism that have been championed by many prior authors (see [1] for a summary). However, our primary goal is not determinism *per se*—we could just as easily record multiple schedules for each input constraint in Σ and randomly select from those schedules at runtime. This added flexibility increases schedule diversity, which has potential benefits for security, fault-tolerance, and performance [1].

2. REPRESENTING SCHEDULES

We represent a schedule using a happens-before graph over a bounded execution trace, where graph nodes are labeled by the triple *(program-counter, thread-id, dynamic-counter)* and edges are induced from program order and synchronization in the usual way, such as between release and acquire operations on the same lock. The *program-counter* label represents a synchronization statement in the program, such as a call to `pthread_mutex_lock`, and the pair *(thread-id, dynamic-counter)* is a coordinate of a Lamport clock. Notice that ordinary memory accesses are not included in the happens-before graph, as we assume data race freedom.

We return to the example in Figure 1. On the left is a simple program in which each thread acquires a different global lock depending on the value of the input `X`. A conventional nondeterministic execution might follow one of 240 possible schedules (5! when `X==0`, and another 5! when `X!=0`). However, just *two* schedules are necessary to cover all inputs for this program—one schedule for `X==0`, and another for `X!=0`. This is illustrated by the right side of Figure 1, which shows one possible set of input-covering schedules, Σ. (The schedules have been abbreviated for space.)

Importantly, for each pair (I,S) $\in \Sigma$, the constraint I should include only those constraint terms that affect whether the schedule S can be followed. That is, constraint I should be a *weakest precondition* of the schedule S. For example, suppose we modify the program in Figure 1 to perform a complex computation in each loop iteration. As long as this computation does not mutate `X` or perform synchronization, the set of input-covering schedules shown in Figure 1 will work equally well for our modified program.

The above representation works well for programs that read their entire input up front (*e.g.*, from the command line or a file) and then perform a bounded-length computation on that input. We extend this representation to support unbounded-length programs in §2.1. To support programs that read inputs continuously, we follow the suggestion made by Tern [4] to represent schedules using a decision tree in which nodes are program statements that read new input and edges are partial schedules that start at an input statement and end at either another input statement or program exit. For brevity, we omit the details.

2.1 Bounded Epochs

We support programs of unbounded length by partitioning execution into *bounded epochs*. In practice, we care not only about programs of truly unbounded length, but also about programs that execute for a "very long" time. For example, consider the following simple program with two threads:

```
Thread 1              Thread 2
for (i in 1..X) {     for (i in 1..Y) {
  lock(L)               lock(L)
  unlock(L)             unlock(L)
}                     }
```

If `X` and `Y` are program inputs, then any set of input-covering schedules must have a unique schedule for each pair (`X`,`Y`). If `X` and `Y` are 32-bit integers, there are 2^{64} possible inputs, so any set of input-covering schedules *must* contain 2^{64} total schedules. Equally problematic: the longest of these schedules must contain 2^{64} total synchronization operations.

Our basic idea is to define schedules one loop iteration at a time. We do this by partitioning the program into bounded epochs that are separated by *epoch markers*. We statically analyze the program to find all loops that perform synchronization, and then place a epoch marker at the entry of such loops. We do this with a simple bottom-up traversal of the call graph starting from synchronization functions like `pthread_mutex_lock`. If some statement s in function f performs synchronization, and if s is contained in the body of a loop, we place an epoch marker at the entry of that loop. Otherwise, we continue up the call graph since callers of f may contain loops that need to be annotated.[1] The above program would be annotated as follows:

```
Thread 1              Thread 2
for (i in 1..X) {     for (i in 1..Y) {
  epochMarker()         epochMarker()
  lock(L)               lock(L)
  unlock(L)             unlock(L)
}                     }
```

Epoch markers act as barriers during program execution, forcing threads to execute in a bulk-synchronous manner. For example, suppose a program's threads begin executing from some initial state. The threads will execute concurrently until each thread is blocked on synchronization, has terminated, or has reached a future epoch marker (possibly the same epoch marker the thread started at, *e.g.*, if the thread started at the beginning of a loop). This quantum of execution corresponds to a single bounded epoch. Execution repeats in this bulk-synchronous manner until all threads terminate. We include "is blocked" in the end-of-epoch conditions to avoid deadlock when thread T_1 attempts to acquire a lock that is held by T_2 while T_2 is stalled at an epoch marker. Note that, in practice, we can use loop unrolling to reduce the frequency of epoch markers.

We now require a set of input-covering schedules for each bounded epoch. A bounded epoch E is named by a list of tuples $(pc_i, callstack_i)$, where pc_i represents the current program counter of thread T_i (*i.e.*, the pc of an epoch marker) and callstack_i is a list of return addresses that represents the calling context. Our algorithm, defined in §3, enumerates all reachable bounded epochs $\mathcal{E}$ and computes an input-covering set Σ_E for each $\text{E} \in \mathcal{E}$. The initial bounded epoch starts at program entry, and its inputs are the program's inputs. All other bounded epochs start from a point in the *middle* of a program's execution. The "input" to these epochs is, potentially, the entire state of memory.

[1]Our current implementation does not support recursive programs. This can be remedied by transforming recursive functions into equivalent functions that use loops.

2.2 Discussion

Bounded epochs make an intractable problem tractable—they limit state-space explosion by bounding both the length of each computed schedule as well as the total number of schedules—but they introduce necessary approximations, as we will demonstrate in §3.2. Further, bounded epochs do not eliminate all causes of explosion in the size of Σ. For example, consider a thread that determines which locks to acquire using a sequence of conditionals as in the following:

```
Thread 1
if (X[0]) {     if (X[1]) {          if (X[n]) {
  lock(L[0])      lock(L[1])    ...    lock(L[n])
  unlock(L[0])    unlock(L[1])         unlock(L[n])
}               }                    }
```

In this case, the set of locks acquired by thread T_1 is uniquely determined by the value of the bitvector `X`. If `X` has 32 bits, any set of input-covering schedules *must* have 2^{32} unique schedules. This is a source of state-space explosion that we can think of no good way to eliminate. Since our underlying problem is undecidable, anyway, we focus our current work on programs without such pathological behavior.

3. FINDING INPUT-COVERING SCHEDULES

Our algorithm for enumerating input-covering schedules is shown in Figure 2. The input is a program P, and the output is a mapping from epochs $\text{E} \in \mathcal{E}$ to set of input-coverings schedules Σ_E for each epoch, where $\mathcal{E}$ is a set of bounded epochs that *may* be reachable. The function `Search` traverses all reachable bounded epochs, starting from an initial epoch representing the call to `main()`. For each epoch E, `Search` invokes `SearchEpoch(E)`, which performs a depth-first search to enumerate a set of input-covering schedules for E along with the set of epochs reachable from E. We describe each function below.

3.1 Finding Schedules for a Single Epoch

The function `SearchEpoch` uses `ExecutePath` to symbolically execute a single path from a given initial state. This path completes when all threads have deadlocked, terminated, or reached an epoch marker. `ExecutePath` can follow any path from a branch and may context switch between threads arbitrarily, as long as it follows a path that is feasible given the initial input constraint. If the path did not end in program termination or deadlock, it ended at a new bounded epoch that we add to the set of reachable epochs (lines 34–35). `EpochId` extracts a unique epoch identifier (recall from §2.1 that an epoch is named by the calling context from which each of its threads begins execution).

For each path, we extract the schedule and then compute a conservative weakest precondition of the schedule using precondition slicing [3], where a *precondition slice* is computed from a dynamic trace and includes only those statements from the trace that might affect whether the final statement was executed. The set of branching statements in a precondition slice combine to form a *precondition* of the final statement. We have modified the algorithm from [3] to instead enumerate all statements from the trace that might

```
1  Search(p: Program) {
2    worklist = {MakeInitialState(p)}
3    output = {}
4
5    while (!worklist.empty()) {
6      // Explore another bounded epoch
7      state = worklist.remove()
8      (schedules, reachable) = SearchEpoch(state)
9
10     // Found an input-covering set for this epoch
11     output.add(EpochId(state), schedules)
12
13     // Add unexplored epochs to the worklist
14     for (e in reachable)
15       if (e not yet visited)
16         worklist.add(MakeStateForEpoch(e))
17   }
18
19   return output
20 }
21
22 SearchEpoch(initState: SymbolicState) {
23   reachableEpochs = {}
24   schedules = {}
25   constraints = {true}
26
27   while (!constraints.empty()) {
28     // Explore a new input constraint
29     state = initState.clone()
30     state.applyConstraint(constraints.remove())
31     (finalState, trace) = ExecutePath(state)
32
33     // Update set of reachable epochs
34     if (!IsTerminatedOrDeadlocked(finalState))
35       reachableEpochs.add(EpochId(finalState))
36
37     // Update set of schedules
38     slice = PrecondSlice(trace)
39     schedules.add(MakeConstraint(slice.branches),
40                   trace.schedule)
41
42     // Accumulate unexplored input constraints
43     inputConstraint = true
44     for (b in slice.branches) {
45       c = inputConstraint ∧ ¬b
46       if (c not yet covered)
47         constraints.add(c)
48       inputConstraint = inputConstraint ∧ b
49     }
50   }
51
52   return (schedules, reachableEpochs)
53 }
```

Figure 2: Searching for input-covering schedules

affect the set of synchronization operations that would be performed. We call this a *synchronization-preserving slice*.

The original algorithm in [3] works much like a standard dynamic backwards slicing algorithm: it iterates backwards over an execution trace, uses a *live* set to track data dependencies, and adds statements to the slice if they modify items in the *live* set. Branches are handled as shown in Figure 3: a branch is included in the slice if either (a) the current head-of-slice is control-dependent on the branch (this is the `Postdominates` check, which is computed with a standard postdominators analysis), or (b) some other path through the branch (not taken in the given trace) might modify an item in the *live* set (this is the `WritesLiveVarBetween` check, which is computed with a static alias analysis).

```
HandleBranch() {
  if (!Postdominates(thread.slice.head, branch)
        || WritesLiveVarBetween(branch, thread.slice.head)
        || SyncOpBetween(branch, thread.slice.head))
    Take(branch)
}
```

Figure 3: How precondition slicing handles branches (our additions are in *italics*)

We make three modifications. First, we include all synchronization statements in the slice to ensure that all control and data dependencies of synchronization are included in the slice. Second, we keep a separate head-of-slice per-thread so that all control-flow checks remain single-threaded. Finally, we include a branch in the slice if some other path through the branch (not taken in the given trace) might perform synchronization (this is the `SyncOpBetween` check in Figure 3). The final addition ensures that a branch is included in the slice if it may affect synchronization.

Because we assume data race freedom, our slicing algorithm does not need to account for potentially-racing accesses when computing data dependencies. Relaxing this assumption would involve a much more complicated implementation of `WritesLiveVarBetween` that would require a may-race analysis, much like the algorithm described in [5].

3.2 Exploring All Reachable Epochs

The function `Search` enumerates input-covering schedules for all epochs that are uncovered by `SearchEpoch`. In `Search`, the key is a call to `MakeStateForEpoch`, which computes, for a given epoch, an initial symbolic state that will be explored by `SearchEpoch`. Each symbolic state includes a set of calling contexts (one per thread), along with a set of constraints on memory. The calling contexts are provided directly by the epoch identifer, but the memory constraints must be computed by `MakeStateForEpoch`.

How does `MakeStateForEpoch` compute the initial memory constraints? The difficulty is that we must compute constraints that are abstract enough to cover all possible concrete initial states of the epoch. The most conservative option is to use a completely unconstrained initial memory, represented by the constraint *true*, but this is obviously an over-approximation—`SearchEpoch` will waste time exploring many infeasible paths. The most precise option is to symbolically enumerate all paths from program entry to the beginning of the epoch, then summarize those paths to compute a very precise initial state, but this will be prohibitively expensive—it suffers from exactly the sort of state-space explosion that bounded epochs are designed to avoid.

Our approach is to use a collection of static dataflow analyses as a comprimise between those two extremes. Each analysis is scalable, interprocedural, and *context-specific* in the sense that it analyzes the specific calling contexts from which the epoch begins. The analyses were designed to remove a few common sources of infeasible paths, but they are necessarily conservative. Included in this collection is a reaching definitions analysis, a lockset analysis [7], a form of barrier matching [9], and a few smaller analyses that we cannot describe in any detail due to space constraints.[2] The following example demonstrates the general idea:

[2]Our data race free assumption makes these analyses more effective. For example, it enables using interference-free regions to reason about cross-thread interference [6].

```
1  Thread 1                  Thread 2
2  void RunA() {             void RunB() {
3    Foo(&thelock)             Bar(&thelock)
4    ...                       ...
5  }                         }
6  void Foo(Lock *a) {       void Bar(Lock *b) {
7    for (i in 1..X) {         for (k in 1..Y) {
8      epochMarker()             epochMarker()
9      lock(a)                   lock(b)
10     ...                       ...
```

Suppose we are given an epoch in which threads T_1 and T_2 begin executing from line 8. To execute this epoch symbolically, `ExecutePath` needs to answer questions such as: Do a and b alias? (If so, the critical sections in T_1 and T_2 must be serialized.) And, does any thread hold lock a when the epoch begins? (If so, T_1 must block until the lock is released.) Our dataflow analyses enable precise answers to these questions in many common scenarios, including the above scenario. For example, our reaching definitions analysis learns that a and b refer to the same lock, and our lockset analysis learns that no locks are held at the beginning of the epoch. Since our ultimate goal is to enumerate input-covering *schedules*, we are especially interested in avoiding infeasible *schedules*—we have found synchronization invariants such as locksets especially helpful towards this goal.

3.3 Optimizations

3.3.1 Shortest-Path First

We have not specified *which* path `ExecutePath` should follow, only that it sould execute some path that is feasible given the initial constraints. The choice of path can have a significant effect on performance. An extreme example is an input-dependent loop that contains no synchronization (and thus does not contain epoch markers): a naïve `ExecutePath` would spin in this loop up to the maximum trip count, which can be quite large. Our insight is that we should always take the *shortest* feasible path, then allow precondition slicing to direct us down longer paths when necessary. Determining the true shortest feasible path is not decidable, so at each branch our heuristic is to select the branch edge with the shortest static distance to a statement that either returns from the current function or exits a loop.

3.3.2 Ignoring Prefix Schedules

Programs are often implemented using a defensive coding style: they frequently check for errors (*e.g.*, via assertions or by checking return codes from system calls) and terminate the program when a failure is detected. Since we include "thread exit" events in our schedules, it appears that enumerating a complete set of input-covering schedules requires enumerating all ways in which the program can exit. In the limit, this requires enumerating all feasible assertion failures, which is a very hard problem on its own.

We avoid this problem using the concept of *prefix schedules*. Suppose a thread executes the following code fragment:

```
lock(A)
if (X == 0) { abort() }
lock(B)
```

Concretely, there are two feasible schedules: (1) the thread locks `A` and then aborts the process, and (2) the thread locks `A` and then locks `B`. We consider the first schedule a *prefix* of the second schedule: at runtime, we can always execute following the second schedule, and then stop early if the `abort` statement is reached. To support prefix schedules, we modify `ExecutePath` and `PrecondSlice` to ignore paths that end in process-exit and do not perform synchronization. For the above fragment, our optimized algorithm outputs just the second schedule, paired with the input-constraint *true*.

3.3.3 Abstracting Input Constraints

Occasionally, symbolic execution eagerly enumerates a large set of input constraints when a single, more abstract constraint would suffice. The following code fragment, adapted from the `dedup` kernel in PARSEC, is a good example:

```
TreeNode* T = TreeSearch(x)
if (T) { lock(L) ... }
```

In this example, a thread searches for a value in a binary search tree, and then performs synchronization if the value is found. Our problem is that symbolic execution will eagerly enumerate *all* concrete heaps for which the expression `T!=0` evaluates to true. Specifically, it attempts to enumerate the following infinite set of input constraints:

```
root->x == x
root->x > x && root->left && root->left->x == x
root->x > x && root->left && root->left->x > x && ...
...
```

Our approach is a form of abstraction: instead of executing `TreeSearch` symbolically, we treat `TreeSearch` as a function that returns new symbolic "input" in the same way that the `getchar` function returns a new symbolic character. (Recall the discussion of programs that read continuous input in §2.) For now, we construct this abstraction manually by annotating `TreeSearch` as an *input function*. In practice, a few subtle conditions must be met for this annotation to be sound—*e.g.*, we must be careful if `TreeSearch` mutates global state—but we omit a discussion for brevity.

4. IMPLEMENTATION

We implemented the above algorithms in the Cloud9 [2] symbolic execution engine. Cloud9 executes C programs that compile to LLVM bytecode, and it includes symbolic models for the pthreads library and most Linux system calls. Programs that use other standard libraries are linked to a version of uClibc that has been compiled to LLVM bytecode. Our current implementation supports programs that synchronize via pthreads only—we do no support programs that synchronize through signals, pipes, or other OS services.

As an optimization, users of our tool can opt to ignore internal synchronization used by library functions such as `printf` to ensure consistency of uClibc data structures. With this option, our algorithm produces schedules that do not include internal uClibc synchronization—such synchronization will be performed nondeterministically at runtime. Our rationale is that developers are more concerned about testing their own code than library internals, so it is sensible to ignore library internals and construct input-covering schedules for application code only. This option works especially well with the prefix schedules optimization (§3.3.2), as programs often call `printf` just before aborting the program.

Checking For Deadlocks. We already check for deadlocks during our search for input-covering schedules. So, in a sense, we get deadlock checking "for free." Our algorithm either outputs a set of non-deadlocking schedules, in which case we are guaranteed to never deadlock at runtime, or its

output will include schedules that deadlock, in which case we *may* deadlock at runtime. Note that we cannot prove these schedules will deadlock, as they may actually be infeasible (recall §3.2). In this way, our deadlock checker is imperfect. Currently, we manually inspect deadlocking schedules to determine if they are actually feasible, but we hope to use more sophisticated strategies for removing infeasible paths in future work to make these manual checks unnecessary.

Runtime System. Our symbolic execution algorithm (§3) outputs a database of input-covering schedules that our runtime system follows faithfully. This is mostly straightforward. At the beginning of the program, or after the program reads new input, the runtime system compares the current inputs with the database of input constraints to select a new schedule. Similarly, epoch markers are turned into barriers, and when all threads reach an epoch barrier, a single thread is selected (arbitrarily) to determine the next schedule. Then, at each synchronization statement, the runtime system inspects the thread's current happens-before node, waits until all incoming happens-before dependencies are satisfied, and then advances to the next node.

The only non-trivial problem our runtime system must solve is the following: How do we check input constraints? The difficulty is that the choice of schedule at an epoch boundary can depend on thread-local variables. Somehow, we must gather all local values so input constraints can be checked. Our solution is to instrument the program to maintain a globally-visible shadow copy of each local variable that is used in input constraints—in practice this is a very small percentage of all variables. Note that we must also make shadow copies of variables that are needed to reach heap objects used in input constraints. For example, if a constraint depends on the value of `x->next->data`, where `x` is a local variable, then we maintain a shadow copy of `x` to ensure that the `data` field is globally reachable.

5. EVALUATION

The primary question we want answered is the following: Are sets of input-covering schedules enumerable, in a reasonable amount of time, for a number of interesting programs? We are still in the early stages of answering this question. Results so far are summarized below. Our methodology for each application is to mark all command-line parameters and input files as symbolic input, with the exception of the "num threads" parameter, which we fix to a small set of values—our happens-before representation of schedules makes the number of threads explicit, so we must compute a different set of input-covering schedules for each thread count.

Blackscholes (from PARSEC) is a simple program that uses fork-join parallelism with no other synchronization. Our algorithm very easily infers that blackscholes can execute under a *single* schedule for each thread count. Further, the entire program requires just one bounded epoch.

Lu (from SPLASH) is a more complex barrier-synchronized program. Our algorithm annotates Lu's main parallel loop with an epoch marker and produces three schedules: (1) a schedule to trace execution from program entry to the first loop iteration; (2) a schedule in which each thread executes one loop iteration; and (3) a schedule for the final loop iteration followed by program exit. Schedule (1) is in the initial bounded epoch, and schedules (2) and (3) comprise a second bounded epoch. When executing the second epoch at runtime, we decide between schedules (2) and (3) by considering the loop induction variable an "input."

Our static analysis (§3.2) infers that the loop executes in lockstep, so we avoid exploring infeasible paths in which different threads exit the loop at different iterations. However, calls to `pthread_join(tid[i])`, made by the main thread to join with all threads before exiting, were a problem. These calls occur in the second epoch. Our static analysis could not concretize each `tid[i]`, resulting in infeasible paths for two reasons: we could not prove that each `tid[i]` was a valid id, and we had to consider all happens-before orderings (*e.g.*, join with T_1, T_2, then T_3, or T_2, T_3, then T_1, and so on). We avoided these paths by replacing the `pthread_join` calls with a more high-level "`joinAllThreads`" call that is easier to reason about symbolically.

Dedup (from PARSEC) uses a five-stage parallel pipeline and is the most complex program we have analyzed. Each pipeline stage contains between one and three loops that are annotated with epoch markers. We have analyzed the "one thread per stage" pipeline configuration only. For initial parallel epoch, the average precondition slice contains between 10 and 15 branches, so we estimate there are about 2^{15} paths in that epoch. Our tool ran out of memory after analyzing just over 5000 paths in 5 hours, so we were unable to produce a final result. We are currently investigating the memory-exhaustion issue, and further, we believe our path completion rate can be improved, *e.g.*, by distributing execution as in [2] or by further optimizing the symbolic engine.

Unfortunately, based on a random sampling of the paths that we explored before timing out, it appears that almost all of those 2^{15} paths represent real behaviors that must be considered, so any further attempt to reduce the number of schedules would need to focus on schedule redundancies, as in §3.3.2, rather than on removing infeasible paths.

Discussion and Future Work. We are currently expanding our empirical evaluation, both to evaluate more applications and to more completely evaluate our system on a number of additional dimensions. Although our results so far are incomplete, we consider them promising. In the long term, we hope to build a set of testing and verification tools that exploit input-covering schedules to more completely investigate the benefits of our system as a whole.

References

[1] T. Bergan et al. The Deterministic Execution Hammer: How well does it actually pound nails? In *WoDet*, 2011.

[2] S. Bucur et al. Parallel Symbolic Execution for Automated Real-World Software Testing. In *EuroSys*, 2011.

[3] M. Costa et al. Bouncer: Securing Software by Blocking Bad Input. In *SOSP*, 2007.

[4] H. Cui et al. Stable Deterministic Multithreading Through Schedule Memoization. In *OSDI*, 2010.

[5] H. Cui et al. Efficient Deterministic Multithreading through Schedule Relaxation. In *SOSP*, 2011.

[6] L. Effinger-Dean et al. Extended Sequential Reasoning for Data-Race-Free Programs. In *MSPC*, 2011.

[7] J. Voung et al. RELAY: Static Race Detection on Millions of Lines of Code. In *FSE*, 2007.

[8] J. Wu et al. Sound and Precise Analysis of Parallel Programs through Schedule Specialization. In *PLDI*, 2012.

[9] Y. Zhang et al. Barrier Matching for Programs With Textually Unaligned Barriers. In *PPoPP*, 2007.