Input-Covering Schedules for Multithreaded Programs

Tom Bergan Luis Ceze Dan Grossman
University of Washington, Department of Computer Science & Engineering
{tbergan,luisceze,djg}@cs.washington.edu

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 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 along with a constrained execution runtime for pthreads programs, and we report 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 verifying each schedule in Σ, in isolation, without needing to reason about the huge space of thread interleavings that arises due to conventional nondeterministic execution.

Categories and Subject Descriptors D.1.3 [*Programming Languages*]: Concurrent Programming; D.3.4 [*Programming Languages*]: Processors—Compilers, Run-time environments

Keywords static analysis; symbolic execution; constrained execution; determinism

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 [8, 31].

We avoid this problem by constraining execution to a small 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 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 simply a partial order of dynamic instances of program statements paired with thread ids, *i.e.*, a happens-before graph.

Given a program P, we first enumerate a small input-covering set Σ using symbolic execution. Then, we attach a custom runtime system to P that constrains execution to follow only those schedules in Σ. This combination of program and runtime system is essentially a new program, P$'$, that accepts all possible inputs and produces semantically correct behavior, like the original program, but uses fewer schedules. We always run the constrained program P$'$ in deployment. The result is that Σ contains the *complete* set of schedules that might be encountered during deployment—this simplifies the verification problem by reducing the number of schedules that must be considered.

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. An important contribution of this work is defining Σ in a way that makes finding *small* input-covering sets more tractable. Notably, programs that run for unbounded periods of time can 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

Enumerating Σ. We use an algorithm based on symbolic execution to systematically enumerate input-covering schedules for a given program. Figure 1 gives a demonstration. On

```
input X
global Lock A,B

Thread 1                 Thread 2
for (i in 1..5) {        for (i in 1..5) {
  if (X == 0) {            if (X == 0) {
    lock(A)                  lock(A)
    unlock(A)                unlock(A)
  } else {                 } else {
    lock(B)                  lock(B)
    unlock(B)                unlock(B)
  }                        }
}                        }
```

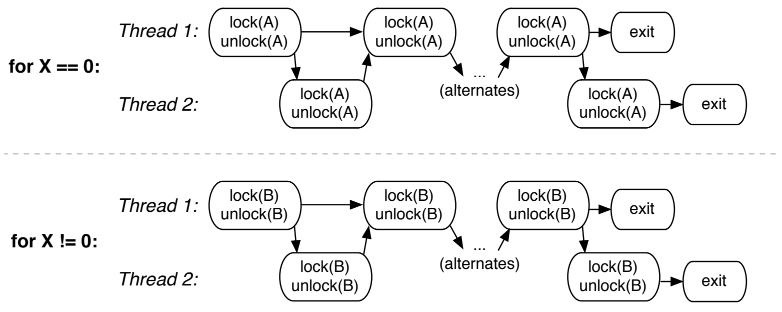

Figure 1. On the left is a simple multithreaded program. On the right is one set of input-covering schedules for the program.

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 happens-before orderings of synchronization statements.

Runtime System. At runtime, we constrain execution to 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.

Verification 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, the input-covering set Σ contains the *complete* set of schedules that might be followed at runtime, and 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 any thread does not terminate with an 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.

More generally, we can reason about each schedule in isolation by serializing the original multithreaded program into $|\Sigma|$ single-threaded programs, where each single-threaded program P_i is constructed by serializing the original multithreaded program P according to schedule $\mathrm{S}_i \in \Sigma$. This reduces the worst-case number of possible program behaviors from $\mathrm{O}(k! \cdot i)$ to $\mathrm{O}(|\Sigma| \cdot i)$, where k is the length of execution and i is the number of possible inputs. This approach is called *schedule specialization* and it has been shown to have real benefits [39]. For example, consider the following code:

```
Thread 1        Thread 2
lock(L)         lock(L)
if (x%2!=0)     if (x%2!=0)
  x++             fail()
unlock(L)       unlock(L)
```

There is an assertion failure in T_2 when it executes before T_1 with an odd value for x. This bug can be difficult to find in conventional systems because it depends on specific combinations of input (x) and schedule (ordering of lock acquires). Our approach computes just one schedule for this code snippet (say, T_1 before T_2), which reduces the verification problem from a hard thread interleaving problem to a simpler (but still difficult) single-threaded reachability problem. We refer to Wu *et al.* [39] and Yang *et al.* [41] for more detailed arguments in favor of schedule specialization.

Assumptions. Our schedule enumeration algorithm assumes data race freedom. When this assumption is broken, we do not compute a true input-covering set—execution may diverge from the expected schedule after a data race. We make this assumption to simplify our analysis in a number of important ways that will be mentioned later.

We also assume that each program has a bounded number of live threads at any given moment. If the number of live threads is input-dependent, we expect the programmer to supply an upper bound for that input. Bounding the number of threads allows us to represent each thread explicitly in the schedule, as shown in Figure 1.

1.2 Comparison to Related Work

Deterministic Execution. 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 [3] for a summary). Since we assume data race freedom, we provide *weak determinism* as in Kendo [34]. 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 [3].

Schedule Memoization. Our system generalizes ideas introduced by TERN [11] and PEREGRINE [12]. 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 input-covering sets requires solving a number of technical challenges not faced by any prior system.

1.3 Contributions and Outline

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. Additionally, we make the following contributions:

- We have designed an algorithm for finding input-covering schedules (§3). We use a number of optimizations to avoid combinatorial explosion problems (§4) and to improve the performance and utility of epochs (§5).
- We have implemented our algorithm on the Cloud9 [7] symbolic execution engine, and we have implemented a runtime system that constrains execution to input-covering schedules produced by our algorithm (§6). Our implementation targets C programs that use pthreads.
- We have performed the first empirical evaluation to address the fundamental question: "how large are sets of input-covering schedules?" (§7) We organize our evaluation as a set of case studies to carefully characterize the program analysis challenges inherent to enumerating input-covering schedules for realistic multithreaded C programs.

This paper focuses on the fundamental problem of enumerating input-covering schedules. We built a runtime system to demonstrate its feasibility and to check the correctness of our schedule enumeration algorithm, but have not yet optimized the runtime system thoroughly. We also have not yet explored verification strategies in any significant detail, though we have implemented a simple deadlock checker that we describe in §6.3. We end this paper with a formal statement of the guarantees provided by our system (§8), a more thorough discussion of related work (§9), and concluding remarks (§10).

2. Representing Schedules

We represent each schedule as a happens-before graph over a finite 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 Lamport timestamp [26]. 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 conditions 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 be equally correct 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 the above representation to support unbounded-length programs in §2.1. To support programs that read inputs continuously, we follow the suggestion made by TERN [11] to represent schedules using a *schedule 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. As most details of this approach are prior work, from here forward we make the simplifying assumption that programs read all inputs at program entry. We will return to continuous inputs in §6.2.

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 an epoch marker at the entry of such loops. The details of this process are explained in §3.1. In short, 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 went back around the same 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 condition 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 (see §5).

We now require a set of input-covering schedules for each bounded epoch. A bounded epoch E is named by a list of pairs $(pc_i, callstack_i)$, 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 E $\in \mathcal{E}$. The initial bounded epoch starts at program entry, and its inputs are the program's inputs. All other 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, which introduces challenges for our runtime system that we will address in §6.2.

2.2 Discussion

Bounded epochs make an intractable problem tractable—they limit combinatorial 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 §3.3. 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 explosion in the size of Σ 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.

Challenges. Bounded epochs introduce two challenges that we will address in §5. First, how can epochs be made performant? Since epochs are runtime barriers, a concern is *imbalance* of work across threads.

Second, since schedules terminate at epoch boundaries, how can verification be effective? We observe the following: any bug that can be detected by examining a single point of execution can be detected by examining a single epoch in isolation, perhaps by using schedule specialization on each epoch. Bugs identifiable from a single point of execution include deadlocks and assertion failures. However, as we will describe shortly, some schedules may actually be *infeasible*—leading to false-positives—and some mechanism of detecting those infeasible schedules is desirable. Other bugs can be detected only by examining sequences of instructions. Atomicity violations are one such example. To simplify detection of these bugs, epochs should be long enough so that *most* buggy instruction sequences will be contained within either one epoch or one short sequence of epochs.

3. Finding Input-Covering Schedules

Our algorithm for enumerating input-covering schedules is shown in Figures 2–5. The input is a program P, and the output is a mapping from epochs E $\in \mathcal{E}$ to a set of input-coverings schedules Σ_E for each epoch, where $\mathcal{E}$ is a set of bounded epochs that *may* be reachable.

We first invoke `PlaceEpochMarkers` to instrument the program with epoch markers. We then invoke `Search` to traverse all reachable bounded epochs, starting from an initial bounded epoch representing the call to `main()`. For each epoch E, `Search` invokes `SearchInEpoch(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 Placing Epoch Markers

The basic constraint for epoch marker placement is the following: we must ensure there are a bounded number of synchronization operations between each epoch marker. This ensures that schedules cannot grow to an unbounded length.

Naïvely, we could satisfy this requirement by placing epoch markers in all loops that perform synchroniza-

```
 1 PlaceEpochMarkers(p: Program) {
 2   covered = {"epochMarker","pthread_barrier_wait"}
 3   worklist = {all fns that directly perform sync}
 4   while (!worklist.empty()) {
 5     F = worklist.popfront()
 6     foreach (loop L in F, bottom-up) {
 7       if (L may perform synchronization
 8           && !IsTrivialLoop(L)
 9           && ∄ epoch marker that must-execute in L)
10         place epoch marker at L.entry
11     }
12     if (∃ epoch marker that must-execute in F)
13       covered.add(F)
14     worklist.pushback(immediate callers of F)
15   }
16 }
```

Figure 2. Algorithm to place epoch markers

tion, including loops that perform synchronization either directly (*e.g.*, by calling `pthread_mutex_lock`) or indirectly (*e.g.*, by calling a function that transitively calls `pthread_mutex_lock`).[1] However, it is important to minimize the number of epoch markers—a large number of epoch markers can lead to a large number of epochs in $\mathcal{E}$.

Our actual algorithm, `PlaceEpochMarkers`, is more careful. We use a bottom-up traversal of the call graph starting from functions that directly perform synchronization (lines 3–5 and 14). For each visited function, we place epoch markers in all loops that perform synchronization and are not pruned by one of the following three optimizations:

Ignore trivial loops. We ignore simple loops of the form:

```
while (!condition)
  pthread_cond_wait(cvar, mutex)
```

This is the common idiom for using pthreads condition variables. We observe that this loop can execute an unbounded number of synchronization operations only if the condition variable `cvar` can be notified an unbounded number of times. So, as long as we ensure that all loops containing notifications are covered by an epoch marker, we can avoid placing an epoch marker in the above loop.

Similarly, we ignore loops where the only form of synchronization is a call to `pthread_create` or `pthread_join`. These loops must be bounded since we assume a bounded number of threads are live at any given moment (recall §1.1).

Don't Cover the Same Loop Twice. We consider a loop *covered* when there exists an epoch marker that must-execute on each iteration of the loop. For example, if loop A contains loop B where B contains an epoch marker, and if at least one iteration of B must-execute for each iteration of A, then we can avoid placing an epoch marker in loop A because that is subsumed by the marker placed in loop B.

[1] Our current implementation does not support recursive functions that synchronize. This can be remedied by transforming recursive functions into equivalent iterative functions, either manually, or automatically as in [28].

```
let hd = thread.slice.head in
if (!Postdominates(hd, branch)
      || WritesLiveVarBetween(branch, hd)
      || SyncOpBetween(branch, hd))
  Take(branch)
```

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

We implement this optimization by visiting the loop forest bottom-up (line 6). Then, we ignore each loop that must-execute a previously placed epoch marker (line 9). The variable `covered` contains a set of functions that *must* execute an epoch marker, so the check at line 9 is implemented by checking if there must-exist a call to a function in the `covered` set—notice that at line 2, we initialize `covered` to include the `epochMarker` function.

Barriers are epoch markers. Since epoch boundaries are runtime barriers, we might as well end epochs at explicit program barriers. So, at line 2, we initialize `covered` to include `pthread_barrier_wait` so the optimization at line 9 will ignore loops that must-execute a barrier. In this way, each call to `pthread_barrier_wait` is treated as an implicit epoch marker.

3.2 Enumerating Schedules for a Single Epoch

The function `SearchInEpoch` (Figure 5) uses `ExecutePath` to symbolically execute a single path from a given initial state. This path completes when all threads have reached an epoch marker, terminated, or deadlocked. `ExecutePath` can follow any path 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 18–19). `EpochId` extracts the epoch identifier (recall from §2.1 that an epoch is named by the calling contexts 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 [10], where a *precondition slice* is computed from an execution 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 [10] to instead enumerate all statements from the trace that might affect the set of synchronization operations that would be performed. We call this a *synchronization-preserving slice*.

The original algorithm in [10] 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

```
Search(p: Program) {
  worklist = {MakeInitialState(p)}
  output = {}

  while (!worklist.empty()) {
    // Explore another bounded epoch
    state = worklist.remove()
    (schedules, reachable) = SearchInEpoch(state)

    // Found an input-covering set for this epoch
    output.add(EpochId(state), schedules)

    // Add unexplored epochs to the worklist
    for (e in reachable)
      if (e not yet visited)
        worklist.add(MakeStateForEpoch(e))
  }

  return output
}
```

Figure 4. Exploring all reachable epochs

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).

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 construct a separate slice for each thread so that all control-flow checks in Figure 3 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 conservative may-race analysis, as we describe in §9.

Shortest-Path First. It is correct for `ExecutePath` to follow any feasible path. However, it is optimal for `ExecutePath` to execute the *shortest* feasible path—longer paths should be executed only as necessary to cover inputs not covered by the shortest path. 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 the current loop.

```
SearchInEpoch(initState: SymbolicState) {
  reachableEpochs = {}
  schedules = {}
  constraints = {true}

  while (!constraints.empty()) {
    // Explore a new input constraint
    state = initState.clone()
    state.applyConstraint(constraints.remove())
    (finalState, trace) = ExecutePath(state)

    // Update set of schedules
    slice = PrecondSlice(trace)
    schedules.add(MakeConstraint(slice.branches),
                  trace.schedule)

    // Update set of reachable epochs
    if (!IsTerminatedOrDeadlocked(finalState))
      reachableEpochs.add(EpochId(finalState))

    // Accumulate unexplored input constraints
    inputConstraint = true
    for (b in slice.branches) {
      c = inputConstraint ∧ ¬b
      if (c not yet covered)
        constraints.add(c)
      inputConstraint = inputConstraint ∧ b
    }
  }

  return (schedules, reachableEpochs)
}
```

Figure 5. Enumerating schedules within a single epoch

3.3 Exploring All Reachable Epochs

The function `Search` enumerates input-covering schedules for all epochs that are uncovered by `SearchInEpoch`. In `Search`, the key is a call to `MakeStateForEpoch`, which computes, for a given epoch, an initial symbolic state that will be explored by `SearchInEpoch`. 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 identifier, 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—`SearchInEpoch` 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 uses a collection of static dataflow analyses as a compromise between those two extremes. The dataflow analyses were designed to remove a few common sources of infeasible paths, but they are necessarily conservative.[2] We refer to a separate technical report for the details [4], but the following example demonstrates the general idea:

```
 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.) The techniques described in [4] enable precise answers to these questions in many common scenarios, including the above scenario. For example, we learn that *a* and *b* refer to the same lock (namely, &theLock), and we learn that no locks are held at the beginning of the epoch at line 8.

Although the analyses from [4] are adept at removing common sources of imprecision, they are necessarily conservative, so infeasible paths may remain.

4. Avoiding Combinatorial Explosion

Avoiding combinatorial explosion is essential. This section describes two categories of optimization:

First, we define optimizations that exploit *redundant schedules* (§4.1). These optimizations allow us to cover more inputs with fewer schedules. Precondition slicing can be viewed as one such optimization, but the optimizations in §4.1 go further by observing that schedules that are not obviously the same can sometimes be treated as if they are.

Second, we deal with unbounded loops that contain synchronization using bounded epochs, but what about unbounded loops that do not contain synchronization? We are hesitant to place epoch markers in every loop since a large number of epoch markers can lead to a large number of epochs. Instead, we deal with unbounded synchronization-free loops using a technique we call *input abstraction* (§4.2).

[2] Our data race freedom assumption makes these analyses more effective. For example, it enables using interference-free regions to reason about cross-thread interference [13].

4.1 Pruning Redundant Schedules

4.1.1 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, execution can always follow the second schedule, and then stop early if the abort statement is reached. To support prefix schedules, we modify ExecutePath and PrecondSlice to ignore branches that exit the process before performing any synchronization. For the above fragment, our optimized algorithm outputs just the second schedule, paired with the input-constraint *true*. We arrive at this output by ignoring the *true* branch of if(X==0). Note, however, that we would require two schedules if there was a call to lock() just before the abort().

4.1.2 Ignoring Library Synchronization

Users of our tool can opt to ignore internal synchronization used by library functions such as printf to ensure consistency of internal library data structures. With this option, our algorithm produces schedules that do not include internal library 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 (§4.1.1), as programs often call printf just before aborting the program.

4.1.3 Symbolic Thread Ids

Redundant schedules can also arise across epoch boundaries. For example, consider a program in which N threads each execute the following code:

```
1  while(X) { epochMarker() ... }
2  ...
3  epochMarker()
```

If X evaluates differently for each thread, then naïvely, we need one epoch in which all threads start at the epoch marker at line 1, another in which T_1 starts at line 3 while all other threads start at line 1, another in which just T_2 starts at line 3, another in which just T_1 and T_2 start at line 3, and so on. In total, there are 2^N epochs.

The above combinatorial explosion arises if we assign each thread a *concrete* thread id during symbolic execution. We can avoid this problem by instead assigning each thread a *symbolic* thread id during symbolic execution. Now, we need to consider just $N+1$ total epochs: one epoch in which all threads start at line 1, another epoch in which one thread starts line 3, another in which two threads start at line 3, and so on. The idea is that the specific assignment of thread ids to calling contexts does not matter, so the `EpochId` function should return a *multiset* of calling contexts, rather than an ordered list. This optimization requires some cooperation with our runtime system. Specifically, at each runtime epoch boundary, we must dynamically map each symbolic thread id to a concrete thread id—we defer details to §6.2.

More generally, if we extend the above example to use k epoch markers, then we explore k^N epochs using concrete thread ids, and just $\left(\!\binom{k}{N}\!\right)$ epochs using symbolic thread ids, where $\left(\!\binom{k}{N}\!\right)$ is k-choose-N with repetitions. However, if we further modify the above example so that each thread T_i executes a unique function f_i, where each f_i includes k epoch markers, then we must explore k^N epochs because there are that many unique combinations of calling contexts.

4.1.4 Redundancy from Code Duplication

We run our schedule enumeration algorithm *after* a compiler optimization pass, as this has been shown to speed-up symbolic execution [9]. However, optimizations can sometimes introduce schedule redundancies by duplicating synchronization. One example is the following transformation, which is called *jump threading* in LLVM:

```
if (X == 0) { f() }         if (X == 0)
lock()                 =>   { f(); lock(); g() }
if (X == 0) { g() }         else { lock() }
```

Our algorithm starts by executing one path through the optimized code (on the right). Suppose we execute the *false* branch. Precondition slicing will notice the `lock()` call in the *true* branch and direct us down that path as well, and the end result is an input-covering set with two schedules, one for `X==0` and `X!=0`. However, both of these schedules are the *same* schedule—the choice of schedule has no real dependency on input `X`.

Our current approach is to disable all transformations that might duplicate code, but unfortunately, this is not always possible. Notably, we cannot disable the following loop transformation because it is fundamental to the way many compilers reason about loops:

```
while (foo()) {        if (foo()) {
  ...            =>      do { ... } while (foo())
}                      }
```

If `foo()` performs synchronization or contains an epoch marker, duplication of the call to `foo()` can lead to redundant schedules that we cannot avoid.

4.2 Abstracting Input Constraints

Unbounded synchronization-free loops can cause an explosion in the number of paths explored by `SearchInEpoch`. The following code fragment is a good example:

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

In this example, a thread searches for a value in a binary tree, and then performs synchronization if the value is found. Our problem is that symbolic execution will eagerly enumerate *all* concrete trees 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 an *uninterpreted function* and add `TreeSearch(x)!=0` to the path constraint. There are two subtleties in this approach:

What Do We Abstract? We abstract all synchronization-free loops and recursive functions that produce *live-out* values that might affect synchronization. Notice that we do *not* abstract loops that contain synchronization, since those loops are already bounded by epoch markers. We start by assuming that no loops or recursive functions need to be abstracted. Then, during each call to `PrecondSlice` (line 13 of Figure 5), we check if any value added to the slice's *live* set was defined in a synchronization-free loop L or recursive function R. If such an L or R is found, it must be abstracted.

How Do We Construct Abstractions? We construct a symbolic function $F_L(\overline{x}) = \overline{y}$, where $\overline{x}$ is the set of *live-ins* for loop L and $\overline{y}$ is the set of *live-outs*, where $\overline{x}$ and $\overline{y}$ can potentially be constructed with some form of summarization, such as the summarization algorithms proposed by Godefroid *et al.* [16, 18] (see §9 for a discussion). However, this is difficult in general since $\overline{x}$ and $\overline{y}$ can each include unboundedly many heap objects. Due to this difficulty, we currently construct each F_L manually. This process is interactive: we first run our algorithm from §3; if our algorithm finds a loop L that must be abstracted, it halts and reports L; we then produce a hand-written abstraction for L and re-run our algorithm.

During symbolic execution, we execute the abstraction F_L in place of the actual loop L. Each F_L should model the *terminating* behaviors of L. We require each F_L to terminate to ensure that our algorithm terminates as well. Of course, the actual loop L may not terminate, and we preserve that behavior—when the program is executed with our runtime system, we execute the actual loop L, not F_L.

Each F_L is allowed to be an *over-approximation* of loop L's terminating behaviors. This eases construction of F_L but adds potential to explore infeasible paths. Producing each F_L is usually not hard in practice, as the loops to abstract are often hidden behind natural abstraction boundaries. Continuing the above example, suppose the binary tree interface includes `TreeAdd` and `TreeDelete`. These appear difficult to abstract since they can mutate unboundedly many heap objects (*e.g.*, to rebalance the tree), but as long as all modifications and traversals are performed behind the `Tree*` interface, we can conservatively model `TreeAdd` and `TreeDelete` by simply generating a fresh symbolic value that represents the new root of the tree.

Although the above explanation is phrased in terms of loops, recursive functions can be abstracted in the same way.

5. Forming Efficient Bounded Epochs

So far we have assumed that in a given epoch, no thread executes beyond its *next* epoch marker. Why might this be inefficient? First, runtime performance is optimal when threads execute a *balanced* amount of work per epoch, but naïvely stopping at the next epoch marker can lead to imbalance. Second, epochs should be long enough so that ordering-dependent bugs, such as atomicity violations, are usually contained within a single epoch.

It is more efficient to allow each thread to bypass a finite number of epoch markers within each bounded epoch. Since epoch markers are placed in loops, we consider this is a form of *loop unrolling*. This optimization coordinates with our runtime system as follows: for each epoch marker bypassed by `ExecutePath`, we add a special node to the current happens-before schedule so that our runtime system will bypass that marker at runtime. We use the following heuristics to bypass epoch markers:

Minimum Epoch Length. A large body of prior work has made the empirical observation that most ordering-dependent bugs occur over a short execution window containing at most W instructions per thread. For example, [30] estimates that W is "thousands of instructions" in the worst case, but "hundreds" in the common case; [29] estimates $W = 3000$; [8] and [35] support this observation but do not give concrete estimates for W, although [8] observes that many "hard" atomicity violations have the form `if(x) compute(x)`, where W spans the short window between the condition and the computation. This prior work suggests the simple heuristic that each thread should execute a minimum of W instructions per epoch.

Balanced Epoch Lengths. Each thread should execute approximately the same number of instructions per epoch. For example, suppose we are about to end an epoch with T_1 and T_2 stalled at epoch markers. If $len(T_1) > len(T_2) + k$, where $len(T_i)$ is the number of instructions executed by T_i in the current epoch and k is a heuristically-chosen constant, then we continue executing T_2 up to its next epoch marker.

6. Implementation

6.1 Symbolic Execution Engine

We implemented the above algorithms in a version of the Cloud9 [7] symbolic execution engine extended with the techniques described in [4]. Cloud9 executes multithreaded C programs that use pthreads and compile to LLVM bytecode. To support unmodified C programs, Cloud9 includes hand-written symbolic models for the Linux system call layer and the pthreads library, and it models other C library functions by linking with an actual libc implementation (uClibc). We have instrumented Cloud9's pthreads library to dynamically capture a happens-before schedule during symbolic execution.

Limitations. Our implementation has a few limitations that we consider minor but list for completeness: async signals, C++ libraries, and floating point arithmetic. First, we do not support asynchronous delivery of POSIX signals. This has not been a problem so far. Should it become an issue, we can support asynchronous delivery by buffering signals until epoch boundaries, similarly to [5] or [24]—such a buffering scheme would eliminate the need to reason about a combinatorial explosion of possible signal delivery points.

Second, Cloud9 ships with a standard C library (uClibc) but not a standard C++ library, and this limits our ability to run C++ programs. Third, our underlying theorem prover, STP [15], does not support floating-point arithmetic. Cloud9 makes progress through floating point arithmetic by concretizing values, which means the resulting path constraints will be incomplete for paths that branch on the result of a floating-point computation. This is often not an issue for our algorithm in practice, since many programs compute floating-point results but do not using floating-point values to decide when to synchronize. However, this does prevent us from analyzing some programs, as we discuss in §7.

Challenges. The effectiveness of precondition slicing is heavily dependent on the presence of a good whole-program alias analysis. The critical operation is the `WritesLiveVar-Between` check (Figure 3)—alias analysis imprecision can lead to the incorrect belief that a live variable was written, which results in an overly strong schedule precondition, which results in the exploration of redundant schedules.

Our implementation uses DSA [27], which, in whole-program mode, degrades to a field-sensitive Steensgaard (equality-based) analysis. Our experience suggests that an inclusion-based analysis is *vital*. The problem intensifies because we link with an entire C library—all pointer variables passed to library functions are effectively merged in the points-to graph. We unfortunately could not find a publicly available alias analysis for LLVM that is more powerful, so we duct-taped this problem by dividing pointer variables into two classes: application code and library code. Variables in the later class are assumed to alias anything, while variables in the former class are analyzed with DSA.

Global state

```
struct ScheduleFragment {
  nextSelectorId: int
  schedule: map (threadId, list of H-B-Node)
}

selectors: map(int, (void)->ScheduleFragment*)
currCallstacks: map(threadId, int)
currFragments: list of ScheduleFragment*
```

Schedule selection at epochs

```
EpochBarrier() {
  isLast = barrier.arrive()
  if (isLast) {          // last thread?
    epochId = hash(sort(currCallstacks.values))
    currFragments.clear()
    currFragments.append(selectors[epochId]())
    barrier.release()
  } else {
    barrier.wait()
  }
}
```

Figure 6. Key components of our runtime system

6.2 Compiler Instrumentation and Runtime System

Our symbolic execution engine outputs a database of input-covering schedules that our runtime system follows faithfully. Recall from §3 that this database maps each epoch E $\in \mathcal{E}$ to an input-covering set Σ_E for E.

At a high-level, our runtime system is mostly straightforward. The global variable `currFragments` contains the happens-before schedule for the currently executing epoch. At the beginning of the program, we compare the current inputs with the database of input constraints to select the initial schedule. Similarly, epoch markers are turned into barriers, and when all threads reach an epoch barrier, a single thread is selected (arbitrarily) to update `currFragments` for the next epoch. Then, at each synchronization statement, the runtime system inspects the calling thread's current happens-before node, waits until all incoming happens-before dependencies are satisfied, and then advances to the next node.

A more detailed view is given in Figure 6. We map each epoch E to a *schedule selector function* F_E for each epoch. Schedules contain a list of happens-before nodes for each thread. There are three technical challenges: At each epoch barrier, how do we efficiently determine the next epoch id E? How do schedule selector functions check input constraints? And, how do we deal with continuous inputs?

Determining the Next Epoch. Each epoch id E is defined by a multiset of per-thread call stacks (recall §4.1.3). We instrument the program to record each thread's call stack in a globally-visible location. Then, the last thread to arrive at an epoch barrier can compute the next epoch id E by sorting this list of call stacks (note that a multiset can be represented by a sorted list). The sorting operation is made efficient by representing callstacks using hash values as in [6]. The algorithm in [6] has only probabilistic guarantees that each calling context is given a unique hash value, but since we know the complete set of epoch ids, we can ensure *a priori* that a unique hash value is computed for each epoch.

Schedule Selector Functions. When invoked, the selector F_E looks for a pair (I,S) $\in \Sigma_E$ such that constraint I matches the current input, then it return S. This is implemented by compiling Σ_E into a decision tree. Our current implementation selects each schedule as a deterministic function of the given input, though this could easily be changed to select schedules nondeterministically when multiple options are available.

Recall that an epoch's *input* can include the state of memory. The difficulty is that the choice of schedule can depend on thread-local variables. Since F_E is executed by one thread only, how does it reason about state local to other threads? 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, as we demonstrate in §7.3. 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 must maintain a shadow copy of `x` to ensure that the `data` field is globally reachable.

Supporting Continuous Inputs. We represent schedules as a tree of *schedule fragments*. At the beginning of an epoch, each thread follows the initial fragment, represented in Figure 6 as `currFragments[0]`. Each fragment f ends in program exit, in an epoch boundary, or with a new input read by thread T. In the later case, thread T invokes the selector function named by f`->nextSelectorId`, then appends the selected fragment to `currFragments`. As other threads arrive at the end of fragment f, they must wait for T to select the next fragment before proceeding. We ignore inputs that are pruned by precondition slicing (§3.2), so updates to `currFragments` occur only after the arrival of inputs that can affect synchronization.

Continuous inputs introduce a further challenge, best illustrated by the following sequence of events:

```
1 EpochBarrier()
2 z += 5
3 ReadInput(&x)
4 if (x == z && y == w) { lock() }
```

The selector function invoked at line 3 will evaluate the term `x == ` z_0+5, where z_0 is the value of `z` at the beginning of the epoch. This value has been lost due to the update at line 2, so we need to *snapshot* `z` at line 1. Note, however, that we do not need to snapshot `y` or `w`—the condition `y==w` does not depend on input `x`, so it can be lifted into the epoch's selector function that is invoked at line 1.

Chances for Further Optimization. Runtime system optimization has not been our focus. We see at least three potential improvements: (1) we can apply a transitive reduction [40] on each happens-before schedule to reduce cross-thread synchronization; (2) for each term evaluated by selector function F_E, we can memoize the value of that term as computed by F_E to avoid recomputation during actual program execution; and (3) we can parallelize F_E to avoid serializing F_E at each epoch boundary (this last proposed improvement is perhaps the most complex).

6.3 Verifying Deadlock Freedom

We already check for deadlocks during our search for input-covering schedules (see Figure 5, line 18). 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 at least one pair (I,S) where schedule S deadlocks, in which case we *may* deadlock at runtime. In the later case, we cannot prove that deadlock will actually occur at runtime because input constraint I may be infeasible (recall §3.3). 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.

7. Evaluation

Our evaluation is organized in three parts. We start with a set of case studies (§7.1) that evaluate the effectiveness of our schedule enumeration algorithm on a range of applications. Our case studies include selections from the SPLASH2 and PARSEC benchmark suites, as well as `pfscan`, a parallel implementation of `grep`. We also characterize the effectiveness of our optimizations (§7.2) and runtime system (§7.3).

We ran all experiments on a 4-core (2-way hyper-threaded) 2.4 GHz Intel Xeon E5462 with 10GB RAM. For each application, we marked all command-line parameters as input, with the exception of the "num threads" parameter, which we fix to the values 2, 4, and 8 to reveal how our analysis and our runtime system scale with increasing thread counts. Capturing command-line inputs required a minor code change of about 10 lines per application, though other inputs (*i.e.*, values returned by system calls) are captured automatically. In some applications, we made two additional code changes as described in §7.1.2 and §7.1.4, respectively.

We attempted to analyze most programs that were analyzed by the related schedule memoization system PEREGRINE [12], but occasionally ran into limitations of our implementation (recall §6.1 and footnote 1). Specifically, we could not run: `barnes` and `ffm` from SPLASH2, which perform synchronization in recursive functions; `pbzip`, which uses C++ libraries; and `ocean`, `fluidanimate`, and `streamcluster`, which use floating-point arithmetic to control synchronization.

7.1 Case Studies

For each case study, we address the following major questions: Is a set of input-covering schedules enumerable in a reasonable amount of time? And if so, how large is $\mathcal{E}$ and how large is each Σ_E? We also attempt to characterize how many of those schedules are infeasible.

Overall results for our *fully optimized* algorithm are summarized in Table 1. Column 2 gives the maximum number of threads live at any given instant (this is a function of the application's "num threads" parameter, which we fix to 2, 4, and 8 as described above). Columns 3–9 summarize our algorithm's final output: Column 3 is the number of reachable epochs ($|\mathcal{E}|$); Columns 4–6 give statistics that summarize the number of schedules per epoch ($|\Sigma_E|$); and Columns 7–9 give statistics that summarize schedules across all epochs ($|\Sigma|$), including the total number of infeasible schedules and deadlocking schedules. Column 10 states the number of input abstractions needed for the given application (recall §4.2). Column 11 gives the overall analysis runtime in seconds (s), minutes (m), or *dnf* when our algorithm did not finish within 2 hours.

For all but one program, we proved that Σ was deadlock-free. We determined the number of infeasible schedules through manual inspection of Σ. For `pfscan`, the schedules were too numerous for manual inspection, so we give a lower bound in Table 1.

Table 2 characterizes the benefits of our optimizations. For brevity, Table 2 includes just a representative subset of applications. Columns 3–4 give the number of epoch markers added by `PlaceEpochMarkers`, both with and without optimizations described in §3.1. Columns 5–16 show the results of running our algorithm with specific optimizations disabled. For "naïve §3.3", we use a naïve implementation of `MakeStateForEpoch` that leaves memory completely unconstrained, and for the remaining column groups, we disable the stated optimization. In each group of columns, $|\mathcal{E}|$ is the number of enumerated epochs, $|\Sigma|$ is the total number of enumerated schedules (summed across all epochs), and *time* is the analysis runtime. We refer to Table 2 in the case studies below, and give a further discussion in §7.2.

7.1.1 The Trivial Case: Fork-Join Parallelism

blackscholes (from PARSEC) uses fork-join parallelism with no other synchronization. It is so simple that we consider it the "hello world" of synchronization analysis. **swaptions** (also from PARSEC) is equally simple. Our algorithm easily infers that these applications need exactly *one* schedule for a given thread count.

7.1.2 Case Study: Barrier-Synchronized Parallelism

fft (from SPLASH2) uses fork-join parallelism with a static number of barriers. Our algorithm infers that `fft` needs just

			$\|\Sigma_E\|$			Summary of Schedules ($\|\Sigma\|$)			# Input	Analysis
App	# Thr	$\|\mathcal{E}\|$	min	max	avg	Total	Infeasible	Deadlocked	Abstracts.	Runtime
blackscholes	3,5,9	1	1	1	1	1	0	0	0	5 s, 6 s, 14 s
swaptions	3,5,9	1	1	1	1	1	0	0	0	4 s, 9 s, 65 s
fft	2,4,8	1	2	2	2	2	0	0	0	7 s, 306 s, 90 m
lu*	2,4,8	2	1	2	1.5	3	0	0	0	8 s, 7 s, 11 s
radix*	2	5	1	2	1.8	9	2	0	0	9 s
radix*	4	5	1	8	3.4	17	10	0	0	10 s
radix*	8	5	1	64	14.8	74	65	0	0	53 s
pfscan*	3	30	2	1209	110.7	3321	203+*unk.*	203	1	343 s
pfscan*	5	50+	2	3792	404.8	12774+	*unk.*	87+	1	dnf

Table 1. Overall results. This is the fully-optimized algorithm. Applications marked with * use *"join on all threads"* (§7.1.2).

		# epoch markers		naïve §3.3			no §4.1.1			no §4.1.3			no §4.1.4		
App	# Thr	w/ §3.1	no §3.1	$\|\mathcal{E}\|$	$\|\Sigma\|$	time	$\|\mathcal{E}\|$	$\|\Sigma\|$	time	$\|\mathcal{E}\|$	$\|\Sigma\|$	time	$\|\mathcal{E}\|$	$\|\Sigma\|$	time
blackscholes	9	0	2	—	—	—	1	***3***	6 s	—	—	—	—	—	—
lu	8	0	2	2	***486+***	***dnf***	***1+***	∞	***dnf***	***5***	***9***	***22 s***	***4***	***7***	***18 s***
radix	8	2	4	***2+***	***131+***	***dnf***	5	***86***	53 s	***42***	***622***	***619 s***	—	—	—
pfscan	3	3	7	***159***	***11272***	***799 s***	30	***3356***	***350 s***	***50***	***4772***	***405 s***	—	—	—

Table 2. Cost of removing optimizations. Compare ***bold*** values in columns 5–16 with columns 3, 7, and 11 in Table 1. We mark columns with a dash (—) when the corresponding optimization has no effect. The time limit for *dnf* is 2 hours.

two schedules for a given thread count. The high analysis runtime is due to the presence of long sequences of conditionals that use division and modulo arithmetic in a way that our SMT solver (STP) finds pathologically challenging.

lu (from SPLASH2) synchronizes using a dynamic number of barriers. Our algorithm divides `lu`'s schedules into two epochs: one that begins at program entry (E_1), and one that begins within `lu`'s main parallel loop (E_2). We need just one schedule for epoch E_1 and two schedules for epoch E_2. In epoch E_2, one schedule traverses one lock-step iteration of the parallel loop, and the other exits the program.

`lu` introduces two program analysis challenges. First, we must infer that all threads execute the main parallel loop in lockstep. Failure to infer this fact results in infeasible schedules, as shown in column 6 of Table 2—a naïve `MakeStateForEpoch` does not make this inference.

Second, `lu` makes calls of the form `pthread_join(t[i])` that are deceptively difficult to analyze. We are unable to uniquely identify each `t[i]`, so we must analyze three paths per call: one in which `t[i]` is an invalid thread id, another in which `t[i]` refers to a thread that has already exited, and another in which `t[i]` refers to a thread that has not yet exited. In total, to join with N threads, we analyze 3^N paths, even though just *one* of those paths is feasible. We avoid this difficulty by replacing calls to `pthread_join` with a high-level *"join on all threads"* operation that is easy to analyze. Each application marked with an asterisk in Table 1 was modified to use this operation in place of `pthread_join`.

7.1.3 Case Study: Barriers and Semaphores

radix (from SPLASH2) is barrier-synchronized like `lu`, but with the addition of two parallel phases that use semaphores to coordinate a tree-based reduction. These semaphores present the major difficulty—as shown in Table 1, we explore a number of infeasible schedules.

The following example demonstrates the problem:

```
1  Thread 1           Thread 2
2  for (...) {        for (...) {
3    epochMarker()      epochMarker()
4    sem_wait(&s)       sem_post(&s)
5    ...                ...
```

`MakeStateForEpoch` cannot prove that `s.count==0` at the beginning of the epoch. (In the actual code, this is difficult because each `&s` is selected from an array.) As a result, we explore an infeasible schedule in which T_1 does *not* block at line 4 because it assumes that `s.count>0`. This schedule incorrectly synchronizes T_1 and T_2, which can lead to the (incorrect) conclusion that the program contains data races.

It is actually quite easy to *prove* that the above schedule is infeasible. Our insight is to exploit Σ, which pairs each schedule with an input constraint `I`. For the above schedule, `I` is `s.count>0`. Let E be the epoch containing that schedule. Our job is to show that each schedule $S_i \in \Sigma$ that terminates at epoch E always terminates with $\neg$`I` resolving to true. This is easy to show using rely-guarantee reasoning: we add $\neg$`(s.count>0)` as an *assertion* to the end of each S_i; we add $\neg$`(s.count>0)` as an *assumption* to the beginning of epoch E; and then we symbolically execute each epoch in $\mathcal{E}$ to ensure that the assertions are always satisfied. Note that the assumption is necessary because epoch E contains a schedule that "loops back" to itself.

It would be possible to automatically discharge the necessary verification conditions. We have not implemented this feature, but we have applied this approach to `radix` by manually annotating the program with assumption and assertion annotations to drive the verification procedure. We verified that the 65 schedules listed as "infeasible" in Table 1 are truly infeasible.

Why were we able to prove infeasibility so easily both in the above example and for `radix`'s 65 infeasible schedules? The reason is that each input constraint `I` happens to be a function of synchronization state *only*. If some `I` was instead a function of arbitrary program state, we would need to consider all *paths* that terminate at epoch E, rather than just all *schedules*. The interesting novelty in our proof is that, since we had a small number of schedules to consider, we could reason about each schedule in isolation by reusing *sequential* rely-guarantee reasoning techniques.

7.1.4 Case Study: Task Queues and Locks

pfscan uses task parallelism with one producer thread and multiple worker threads, and it uses locks to guard shared data. The queue is implemented with locks and condition variables. `pfscan` has the following high-level structure:

```
1  Producer           Consumers
2  for (f in files)   while (dequeue(&f))
3    enqueue(f)         scanfile(f)
```

`scanfile` implements string matching. We had to abstract one loop (in `scanfile`) using the technique described in §4.2. This loop computes the next matching substring—its live-ins include a string buffer and a current position, and its live-out is the position of the next match. With 5 threads, were unable to enumerate a complete set of input-covering schedules within a two hour time limit.

Interestingly, the prefix schedules optimization (no §4.1.1 in Table 2) does not help `pfscan` much at all. The reason is that `pfscan` acquires a lock on almost every failure path to perform logging. In fact, the majority of schedules enumerated for `pfscan` are needed to handle these failure paths: with 3 threads, at least one thread executed a failure path on 2092 out of 3321 total schedules. Since all failure paths acquire the *same* lock, they could conceivably be merged into one schedule—this is an interesting direction for future research.

For `pfscan`, our algorithm produces a set Σ that includes deadlocking schedules. These deadlocks are all infeasible. The deadlocks include two scenarios: (1) the producer believes the queue is full while the consumers have already exited, and (2) the consumers believe the producer has exited without first setting the "done" flag. We enumerate these false deadlocks because our implementation of `MakeStateForEpoch` is not powerful enough to produce constraints that precisely relate the queue's `capacity`, `count`, and `done` fields.

We also explore redundant schedules that arise from code duplication. Recall from §4.1.4 that compilers transform while loops to an if-then-do-while form. This transformation duplicates the `dequeue` call made by each consumer thread in line 2 of the above code snippet. If this transformation could be disabled, we would explore 7 fewer epochs and 326 fewer schedules for 3 threads, and at least 11 fewer epochs and 479 fewer schedules for 5 threads.

			DTree Size		Norm. Exec Time		
App	IPE	LI	max	avg	2thr	4thr	8thr
blackscholes	all	0	0	0	1.0	1.0	1.0
fft	all	0	1	1	1.0	1.0	1.0
lu	200M	1	4	1.75	1.0	1.0	1.0
radix	1B	4	6	2.95	1.0	1.05	1.05
pfscan	6K	7	24	2.2	1.6	—	—

Table 3. Runtime system characterization: IPE is *avg. instructions per epoch*; LI is *# local variables instrumented*.

7.2 Optimizations Characterization

Table 2 shows that an effective `MakeStateForEpoch` is *vital*. An effective `MakeStateForEpoch` not only reduces the number of explored paths, but it also improves analysis runtime by providing a more tightly constrained initial state that our symbolic engine has n easier time reasoning about. To see this effect, compare the rate of schedule enumeration in columns 6–7 of Table 2 (4 schedules per minute for `lu`) with the rate of enumeration in Table 1 (16 schedules per minute for `lu`).

The prefix schedules optimization (§4.1.1) is also essential. Without this optimization enabled, our algorithm explores an essentially unbounded number of failure paths in `lu` and never escaped the first epoch. (The number of failure paths could be bounded by applying input abstraction to two loops, but the key point is that `lu` does *not* require input abstraction when the prefix schedules optimization is enabled.)

The optimization to avoid code duplication (§4.1.4) improved `lu` only. However, as we observed in §7.1.4, we explored redundant schedules in `pfscan` as a result of a form of code duplication that this optimization could not eliminate.

7.3 Runtime System Characterization

Table 3 characterizes our runtime system in three ways, as described below. All numbers in Table 3 are based on executions of the final *instrumented* program which is linked with our custom runtime system. These executions are constrained to the input-covering schedules summarized in Table 1, and, except where otherwise mentioned, the "num threads" was set to 2 for `pfscan` and 4 for everything else.

In Table 3, Column 2 reports the average number of instructions executed per thread in a single epoch (IPE). We counted instructions by instrumenting LLVM bytecode, so actual the number of x86 instructions executed may differ slightly. As discussed in §5, IPE should ideally be large enough to span most ordering bugs. Our average IPE is well over the window of 3K instructions that was suggested by Lucia et al. in [29]. Although the averages are generally much higher than 3K, we noticed some variability. For example, IPE for `radix` fluctuated between 30K instructions and 1B instructions, depending on the parallel phase.

Columns 3–5 characterize the amount of work performed to compute a new schedule. Column 3 states the number of local variables instrumented to maintain shadow copies, and columns 4 and 5 state the maximum and average number of arithmetic and boolean operators used by schedule selector decision trees (recall §6.2).

Columns 4–6 characterize our runtime overheads. Each column states the execution time of the final instrumented program (linked with our runtime system) normalized to nondeterministic execution with the same number of threads. A value of 2.0 means "twice as long." For benchmark applications, we used standard benchmark workloads, and for `pfscan`, we performed a search in a directory containing 50 files. For barrier-synchronized applications, overhead is minimal—the programs are already designed to execute in a bulk-synchronous fashion. For `pfscan`, we were unable to measure how well our runtime scales with increasing threads, since we were unable to enumerate input-covering schedules for more than 2 threads within our time limit.

8. Discussion of Guarantees

Given a program P, our schedule enumeration algorithm outputs a set of bounded epochs $\mathcal{E}$ along with a set of input-covering schedules Σ_E for each epoch $E \in \mathcal{E}$. Our schedule enumeration algorithm and runtime system combine to provide the following guarantees, which we state without proof:

Property 1 (Completeness of Σ_{E_0})**.** *Suppose execution begins from program entry with initial memory state M_0, where M_0 contains nothing except the program's inputs. If Σ_{E_0} is the set of input-covering schedules for E_0, the epoch at program entry, then for all valid M_0, there must exist a pair $(I, S) \in \Sigma_{E_0}$ such that M_0 satisfies constraint I.*

Property 2 (Soundness and Completeness of $\mathcal{E}$ and all Σ_E)**.** *Suppose execution begins from a program context corresponding to some epoch $E \in \mathcal{E}$, and suppose the initial memory state is M.*

Then, for all pairs $(I, S) \in \Sigma_E$ where M satisfies constraint I, if our runtime system forces execution to follow S, then either: (a) execution will encounter a data race; or (b) execution will follow schedule S without deviation. In case (b), schedule S must terminate at program exit, at a deadlock, or at some subsequent epoch $E' \in \mathcal{E}$. If schedule S terminates at epoch E', then execution must arrive at E' with a memory state M' such that there exists a pair $(I', S') \in \Sigma_{E'}$ where M' satisfies constraint I'.

Properties 1 and 2 establish that our system is both sound and complete for race-free programs. By *sound*, we mean that for any epoch $E \in \mathcal{E}$ and any pair $(I, S) \in \Sigma_E$, it must be possible for execution to follow schedule S when given an appropriate initial memory state. By *complete*, we mean that, for all possible program inputs, execution will proceed through a (possibly nonterminating) sequence of epochs $E_0, E_1, E_2, \cdots$, where each E_i exists in $\mathcal{E}$, and as execution arrives at each epoch E_i, there must exist a schedule $S_i \in \Sigma_{E_i}$ such that execution can be constrained to S_i within that epoch. Soundness is established by Property 2, and completeness is established by Property 1 combined with inductive application of Property 2.

The important consequence of Properties 1 and 2 is that verification tools can reason soundly and completely even when they consider only those schedules contained in Σ. Of course, these properties hold *only when* the program's execution is constrained by our runtime system—when execution does *not* use our runtime system, Σ under-approximates the set of schedules that might be followed and our verification guarantees are voided. This is why we intend to use our runtime system in *all* executions of a given program.

Additionally, our system is subject to two categories of limitations that we summarize below:

Fundamental Assumptions. As stated in §1.1, our approach *fundamentally* assumes, first, that programs are data race free, and second, that programs have a bounded number of live threads at any moment. When the first assumption is broken, our schedule enumeration algorithm is unsound and execution can diverge from the expected schedule at runtime. When the second assumption is broken, our schedule enumeration algorithm will not terminate.

Limitations of our Implementation. As stated in §6.1, our implementation has limited support for async signals, C++ libraries, and floating point arithmetic. As stated in footnote 1 in §3.1, our implementation does not support recursive functions that synchronize. Properties 1 and 2 do not hold for programs that exceed these limitations. However, these limitations are specific to our implementation and are not fundamental to our approach.

Full proofs of Properties 1 and 2 are beyond the scope of this paper. Full proofs would require a model of execution, a model of the runtime constraint system, and either assuming correct or proving correct our slicing algorithm (based on precondition slicing, which was described without a formal proof of correctness [10]), our underlying symbolic execution engine [7], and our underlying SMT solver [15].

9. Related Work

Schedule Memoization. The most closely related work is *schedule memoization* from TERN [11] and PEREGRINE [12]. As already mentioned, those systems provide best-effort schedule memoization only, while our system enumerates a complete input-covering set. Our notion of epochs is related to TERN's idea of *windowing*, which handles a specific kind of unboundedness—event loops in server programs.

TERN's windowing requires programmer annotations, while we introduce epoch boundaries automatically.

PEREGRINE uses a similar slicing algorithm to approximate weakest preconditions. A key technical difference is that PEREGRINE's algorithm does not assume data race freedom. Instead, PEREGRINE uses a static may-race analysis to find memory access pairs that may-race and then adds a happens-before edge to the schedule for each such pair. Adopting this approach in our setting would result in a more complex analysis and more symbolic path explosion due to the need to consider many possible may-race pairs. Note that path explosion is not a problem in PEREGRINE's setting, where slicing is used to compute an input constraint for tested paths *only*, but not to select more symbolic paths.

Finally, TERN allows developers to explore multiple schedules for the same input and then discard those schedules that trigger bugs. This is a form of *automatic bug avoidance* that could be adopted by our system as well.

Deterministic Execution. See our comparison in §1.2. A further comparison can be found in the critique by Yang *et al.* [41], who observe that, while determinism reduces the number of schedules to one schedule per input, determinism does not necessarily reduce the *total* number of schedules by a significant amount. Yang *et al.* argue—and we agree—that reducing the *total* number of schedules provides a more significant benefit to testing and verification tools.

Synchronization Analysis. Static analyses have been developed to summarize synchronization behavior. These include may-happen-in-parallel analysis [33] and barrier matching [1, 42]—see Rinard's survey for a summary [37]. These analyses are cheap to compute but they construct schedules approximately, while our analysis is expensive to compute but constructs schedules precisely; thus, these approaches occupy two opposite ends of the performance/precision tradeoff.

Symbolic Execution. We faced three main technical challenges: infeasible paths (§3.3), redundant schedules (§4.1), and unbounded loops (§2 and §4.2). Each challenge represents a specific instance of symbolic execution's *path explosion* problem, which has been studied extensively. For example, our optimizations to avoid schedule redundancies are reminiscent of various partial-order reductions developed for model checking [14, 20], though that prior work identifies schedule redundancies given a fixed input, while we identify redundancies across inputs (cf. §4.1.1).

Further, in §4.2 we observed that some form of input abstraction is necessary to achieve good scalability of symbolic execution. Other authors have made the same observation, most notably Anand *et al.* [2] and Godefroid [17]. Anand *et al.* [2] propose using manually-written abstractions (as we do), and they propose a methodology for writing those abstractions. Relatedly, abstractions can be created using programmer-written contracts [38] or via automatically constructed summaries [16, 18].

Finally, one can view our use of bounded epochs as a form of *path merging* [19, 21], where epoch boundaries represent path merge-points. Classical path merging algorithms execute all paths connecting control-flow points A and B, then merge the resulting states at B. This is made feasible by keeping the distance between A and B short. Our algorithm does not execute all paths within each epoch, making it more challenging to construct initial states for middle-of-program epochs, so we rely on other techniques [4].

Program Verification. A large body of prior work has focused on the verification of multithreaded programs. Much of this work has used *preemption bounding*—a schedule is preemption bounded to depth k if the schedule includes no more than k preemptions. Musuvathi *et al.* used this idea in a model checker for multithreaded programs [31, 32]. Qadeer and Wu showed how to reduce a multithreaded program into a sequential program given a fixed k [36], and subsequent authors have defined more advanced reductions [22, 23, 25]. Although this approach grows exponentially more costly as k increases, it has been shown empirically that many concurrency bugs can be found with $k \leq 2$, making this approach practical. For example, Qadeer and Wu used this technique to find data race bugs in Windows device drivers.

The idea to analyze a multithreaded program by first reducing it to an equivalent sequential program is shared by the technique of *schedule specialization* [39] that we summarized in §1.1. Unfortunately, all of the above reductions are *incomplete* in practice, in the sense that they do not analyze *all* possible schedules. Specifically, preemption bounding is incomplete unless $k \approx \infty$, and schedule specialization is incomplete unless all schedules are available. By constraining execution to a small set of input-covering schedules, our approach can make these promising reductions *complete*. Notably, the approach to schedule specialization taken by Wu *et al.* [39] can be *directly* applied to our proposed system by producing a specialized program for each schedule in Σ.

10. Conclusions

This paper opens two new research directions. First, we have introduced the *input-covering schedules* problem. We showed how to make this problem more tractable by partitioning execution into bounded epochs. We designed and implemented an algorithm for enumerating input-covering schedules. An empirical evaluation demonstrates that it is possible to enumerate a complete set of input-covering schedules for at least some realistic programs.

Second, by constraining execution to a set of input-covering schedules, we open the door for new verification techniques that exploit the fact that, in such a constrained environment, all thread schedules are known *a priori*. We took a small first step in this direction by designing and implementing a simple deadlock checker.

The source code for our implementation will be made available at `http://sampa.cs.washington.edu/`.

Acknowledgements

We thank the anonymous reviewers for their valuable comments. We also thank the members of the Sampa and PLSE groups at UW for their feedback and fruitful discussions. This work was supported in part by the National Science Foundation CAREER award 0846004, a Google Ph.D. Fellowship awarded to Tom Bergan, and a gift from Microsoft.

References

[1] A. Aiken and D. Gay. Barrier Inference. In *POPL*, 1998.

[2] S. Anand, C. S. Păsăreanu, and W. Visser. Symbolic Execution with Abstract Subsumption Checking. In *SPIN*, 2006.

[3] T. Bergan, J. Devietti, N. Hunt, and L. Ceze. The Deterministic Execution Hammer: How Well Does it Actually Pound Nails? In *Workshop on Determinism and Correctness in Parallel Programming (WoDet)*, 2011.

[4] T. Bergan, D. Grossman, and L. Ceze. Symbolic Execution of Multithreaded Programs from Arbitrary Program Contexts. Technical Report UW-CSE-13-08-01, Univ. of Washington.

[5] T. Bergan, N. Hunt, L. Ceze, and S. Gribble. Deterministic Process Groups in dOS. In *OSDI*, 2010.

[6] M. D. Bond and K. S. McKinley. Probabilistic Calling Context. In *OOPSLA*, 2007.

[7] S. Bucur, V. Ureche, C. Zamfir, and G. Candea. Parallel Symbolic Execution for Automated Real-World Software Testing. In *EuroSys*, 2011.

[8] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A Randomized Scheduler with Probabilistic Guarantees of Finding Bugs. In *ASPLOS*, 2010.

[9] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI*, 2008.

[10] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado. Bouncer: Securing Software by Blocking Bad Input. In *SOSP*, 2007.

[11] H. Cui, J. Wu, C. che Tsai, and J. Yang. Stable Deterministic Multithreading Through Schedule Memoization. In *OSDI*, 2010.

[12] H. Cui, J. Wu, J. Gallagher, H. Guo, and J. Yang. Efficient Deterministic Multithreading through Schedule Relaxation. In *SOSP*, 2011.

[13] L. Effinger-Dean, H.-J. Boehm, P. Joisha, and D. Chakrabarti. Extended Sequential Reasoning for Data-Race-Free Programs. In *Workshop on Memory Systems Performance and Correctness (MSPC)*, 2011.

[14] C. Flanagan and P. Godefroid. Dynamic Partial-Order Reduction for Model Checking Software. In *POPL*, 2005.

[15] V. Ganesh and D. L. Dill. A Decision Procedure for Bit-vectors and Arrays. In *CAV*, 2007.

[16] P. Godefroid. Compositional Dynamic Test Generation. In *POPL*, 2007.

[17] P. Godefroid. Higher-Order Test Generation. In *PLDI*, 2011.

[18] P. Godefroid and D. Luchaup. Automatic Partial Loop Summarization in Dynamic Test Generation. In *ISSTA*, 2011.

[19] T. Hansen, P. Schachte, and H. Sondergaard. State Joining and Splitting for the Symbolic Execution of Binaries. In *Intl. Conf. on Runtime Verification (RV)*, 2009.

[20] V. Kahlon, C. Wang, and A. Gupta. Monotonic Partial Order Reduction: An Optimal Symbolic Partial Order Reduction Technique. In *CAV*, 2007.

[21] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea. Efficient State Merging in Symbolic Execution. In *PLDI*, 2012.

[22] S. La Torre, P. Madhusudan, and G. Parlato. Context-Bounded Analysis of Concurrent Queue Systems. In *TACAS*, 2008.

[23] S. La Torre, P. Madhusudan, and G. Parlato. Reducing Context-Bounded Concurrent Reachability to Sequential Reachability. In *CAV*, 2009.

[24] O. Laadan, N. Viennot, and J. Nieh. Transparent, Lightweight Application Execution Replay on Commodity Multiprocessor Operating Systems. In *SIGMETRICS*, 2010.

[25] A. Lal and T. Reps. Reducing Concurrent Analysis Under a Context Bound to Sequential Analysis. In *CAV*, 2008.

[26] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7), July 1978.

[27] C. Lattner. *Macroscopic Data Structure Analysis and Optimization*. PhD thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, May 2005.

[28] Y. A. Liu and S. D. Stoller. From Recursion to Iteration: What are the Optimizations? In *PEPM*, 1999.

[29] B. Lucia, L. Ceze, and K. Strauss. ColorSafe: Architectural Support for Debugging and Dynamically Avoiding Multi-Variable Atomicity Violations. In *ISCA*, 2010.

[30] B. Lucia, J. Devietti, K. Strauss, and L. Ceze. Atom-Aid: Detecting and Surviving Atomicity Violations. In *ISCA*, 2008.

[31] M. Musuvathi and S. Qadeer. Iterative Context Bounding for Systematic Testing of Multithreaded Programs. In *PLDI*, 2007.

[32] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and Reproducing Heisenbugs in Concurrent Programs. In *OSDI*, 2008.

[33] G. Naumovich, G. S. Avrunin, and L. A. Clarke. An Efficient Algorithm for Computing MHP Information for Concurrent Java Programs. In *FSE*, 1999.

[34] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: Efficient Deterministic Multithreading in Software. In *ASPLOS*, 2009.

[35] S. Park, S. Lu, and Y. Zhou. CTrigger: Exposing Atomicity Violation Bugs from their Hiding Places. In *ASPLOS*, 2009.

[36] S. Qadeer and D. Wu. KISS: Keep It Simple and Sequential. In *PLDI*, 2005.

[37] M. Rinard. Analysis of Multithreaded Programs. In *Static Analysis Symposium (SAS)*, 2001.

[38] S. Tobin-Hochstadt and D. Van Horn. Higher-Order Symbolic Execution via Contracts. In *OOPSLA*, 2012.

[39] J. Wu, Y. Tang, G. Hu, H. Cui, and J. Yang. Sound and Precise Analysis of Parallel Programs through Schedule Specialization. In *PLDI*, 2012.

[40] M. Xu, M. Hill, and R. Bodik. A Regulated Transitive Reduction for Longer Memory Race Recording. In *ASPLOS*, 2006.

[41] J. Yang, H. Cui, and J. Wu. Determinism Is Overrated: What Really Makes Multithreaded Programs Hard to Get Right and What Can Be Done About It. In *HotPar*, 2013.

[42] Y. Zhang and E. Duesterwald. Barrier Matching for Programs With Textually Unaligned Barriers. In *PPoPP*, 2007.