Efficient Deterministic Multithreading through Schedule Relaxation

Heming Cui, Jingyue Wu, John Gallagher, Huayang Guo, Junfeng Yang
{heming, jingyue, jmg, huayang, junfeng}@cs.columbia.edu
Department of Computer Science
Columbia University

ABSTRACT

Deterministic multithreading (DMT) eliminates many pernicious software problems caused by nondeterminism. It works by constraining a program to repeat the same thread interleavings, or *schedules*, when given same input. Despite much recent research, it remains an open challenge to build *both deterministic and efficient* DMT systems for general programs on commodity hardware. To deterministically resolve a data race, a DMT system must enforce a deterministic schedule of shared memory accesses, or *mem-schedule*, which can incur prohibitive overhead. By using schedules consisting only of synchronization operations, or *sync-schedule*, this overhead can be avoided. However, a sync-schedule is deterministic only for race-free programs, but most programs have races.

Our key insight is that races tend to occur only within minor portions of an execution, and a dominant majority of the execution is still race-free. Thus, we can resort to a mem-schedule only for the "racy" portions and enforce a sync-schedule otherwise, combining the efficiency of sync-schedules and the determinism of mem-schedules. We call these combined schedules *hybrid schedules*.

Based on this insight, we have built PEREGRINE, an efficient deterministic multithreading system. When a program first runs on an input, PEREGRINE records an execution trace. It then *relaxes* this trace into a hybrid schedule and reuses the schedule on future compatible inputs efficiently and deterministically. PEREGRINE further improves efficiency with two new techniques: *determinism-preserving slicing* to generalize a schedule to more inputs while preserving determinism, and *schedule-guided simplification* to precisely analyze a program according to a specific schedule. Our evaluation on a diverse set of programs shows that PEREGRINE is deterministic and efficient, and can frequently reuse schedules for half of the evaluated programs.

Categories and Subject Descriptors:

D.4.5 [**Operating Systems**]: Threads, Reliability D.2.4 [**Software Engineering**]: Software/Program Verification;

General Terms:

Algorithms, Design, Reliability, Performance

Keywords:

Deterministic Multithreading, Program Slicing, Program Simplification, Symbolic Execution

SOSP '11, October 23-26, 2011, Cascais, Portugal.

1 Introduction

Different runs of a multithreaded program may show different behaviors, depending on how the threads interleave. This *nondeterminism* makes it difficult to write, test, and debug multithreaded programs. For instance, testing becomes less assuring because the schedules tested may not be the ones run in the field. Similarly, debugging can be a nightmare because developers may have to reproduce the exact buggy schedules. These difficulties have resulted in many "heisenbugs" in widespread multithreaded programs [39].

Recently, researchers have pioneered a technique called *deterministic multithreading (DMT)* [9, 10, 12, 19, 20, 42]. DMT systems ensure that the same input is always processed with the same deterministic schedule, thus eliminating heisenbugs and problems due to nondeterminism. Unfortunately, despite these efforts, an open challenge [11] well recognized by the DMT community remains: how to build *both deterministic and efficient* DMT systems for general multithreaded programs on commodity multiprocessors. Existing DMT systems either incur prohibitive overhead, or are not fully deterministic if there are data races.

Specifically, existing DMT systems enforce two forms of schedules: (1) a *mem-schedule* is a deterministic schedule of shared memory accesses [9, 10, 20], such as `load/store` instructions, and (2) a *sync-schedule* is a deterministic order of synchronization operations [19, 42], such as `lock()/unlock()`. Enforcing a mem-schedule is truly deterministic even for programs with data races, but may incur prohibitive overhead (*e.g.*, roughly 1.2X-6X [9]). Enforcing a sync-schedule is efficient (*e.g.*, average 16% slowdown [42]) because most code does not control synchronization and can still run in parallel, but a sync-schedule is only deterministic for race-free programs, when, in fact, most real programs have races, harmful or benign [39, 54]. The dilemma is, then, to pick either determinism or efficiency but not both.

Our key insight is that although most programs have races, these races tend to occur only within minor portions of an execution, and the majority of the execution is still race-free. Thus, we can resort to a mem-schedule only for the "racy" portions of an execution and enforce a sync-schedule otherwise, combining both the efficiency of sync-schedules and the determinism of mem-schedules. We call these combined schedules *hybrid schedules*.

Based on this insight, we have built PEREGRINE, an efficient deterministic multithreading system to address the aforementioned open challenge. When a program first runs on an input, PEREGRINE records a detailed execution trace including memory accesses in case the execution runs into races. PEREGRINE then *relaxes* this detailed trace into a hybrid schedule, including (1) a total order of synchronization operations and (2) a set of execution order constraints to deterministically resolve each occurred race. When the same input is provided again, PEREGRINE can reuse this schedule deterministically and efficiently.

Reusing a schedule only when the program input matches exactly is too limiting. Fortunately, the schedules PEREGRINE computes are often "coarse-grained" and reusable on a broad range of inputs. Indeed, our previous work has shown that a small number of sync-schedules can often cover over 90% of the workloads for real programs such as `Apache` [19]. The higher the reuse rates, the more efficient PEREGRINE is. Moreover, by reusing schedules, PEREGRINE makes program behaviors more *stable* across different inputs, so that slight input changes do not lead to vastly different schedules [19] and thus "*input-heisenbugs*" where slight input changes cause concurrency bugs to appear or disappear.

Before reusing a schedule on an input, PEREGRINE must check that the input satisfies the *preconditions* of the schedule, so that (1) the schedule is feasible, *i.e.*, the execution on the input will reach all events in the same deterministic order as in the schedule and (2) the execution will not introduce new races. (New races may occur if they are *input-dependent*; see §4.1.) A naïve approach is to collect preconditions from all input-dependent branches in an execution trace. For instance, if a branch instruction inspects input variable `X` and goes down the true branch, we collect a precondition that `X` must be nonzero. Preconditions collected via this approach ensures that an execution on an input satisfying the preconditions will always follow the path of the recorded execution in all threads. However, many of these branches concern thread-local computations and do not affect the program's ability to follow the schedule. Including them in the preconditions thus unnecessarily decreases schedule-reuse rates.

How can PEREGRINE compute sufficient preconditions to avoid new races and ensure that a sched-

ule is feasible? How can PEREGRINE filter out unnecessary branches to increase schedule-reuse rates? Our previous work [19] requires developers to grovel through the code and mark the input affecting schedules; even so, it does not guarantee full determinism if there are data races.

PEREGRINE addresses these challenges with two new program analysis techniques. First, given an execution trace and a hybrid schedule, it computes sufficient preconditions using *determinism-preserving slicing*, a new precondition slicing [18] technique designed for multithreaded programs. Precondition slicing takes an execution trace and a *target* instruction in the trace, and computes a *trace slice* that captures the instructions required for the execution to reach the target with equivalent operand values. Intuitively, these instructions include "branches whose outcome matters" to reach the target and "mutations that affect the outcome of those branches" [18]. This trace slice typically has much fewer branches than the original execution trace, so that we can compute more relaxed preconditions. However, previous work [18] does not compute correct trace slices for multithreaded programs or handle multiple targets; our slicing technique correctly handles both cases.

Our slicing technique often needs to determine whether two pointer variables may point to the same object. *Alias analysis* is the standard static technique to answer these queries. Unfortunately, one of the best alias analyses [52] yields overly imprecise results for 30% of the evaluated programs, forcing PEREGRINE to reuse schedules only when the input matches almost exactly. The reason is that standard alias analysis has to be conservative and assume all possible executions, yet PEREGRINE cares about alias results according only to the executions that reuse a specific schedule. To improve precision, PEREGRINE uses *schedule-guided simplification* to first simplify a program according to a schedule, then runs standard alias analysis on the simplified program to get more precise results. For instance, if the schedule dictates eight threads, PEREGRINE can clone the corresponding thread function eight times, so that alias analysis can separate the results for each thread, instead of imprecisely merging results for all threads.

We have built a prototype of PEREGRINE that runs in user-space. It automatically tracks `main()` arguments, data read from files and sockets, and values returned by `random()`-variants as input. It handles long-running servers by splitting their executions into *windows* and reusing schedules across windows [19]. The hybrid schedules it computes are fully deterministic for programs that (1) have no nondeterminism sources beyond thread scheduling, data races, and inputs tracked by PEREGRINE and (2) adhere to the assumptions of the tools PEREGRINE uses.

We evaluated PEREGRINE on a diverse set of 18 programs, including the `Apache` web server [6]; three desktop programs, such as `PBZip2` [3], a parallel compression utility; implementations of 12 computation-intensive algorithms in the popular SPLASH2 and PARSEC benchmark suites; and `racey` [29], a benchmark with numerous intentional races for evaluating deterministic execution and replay systems. Our results show that PEREGRINE is both deterministic and efficient (executions reusing schedules range from 68.7% faster to 46.6% slower than nondeterministic executions); it can frequently reuse schedules for half of the programs (*e.g.*, two schedules cover all possible inputs to `PBZip2` compression as long as the number of threads is the same); both its slicing and simplification techniques are crucial for increasing schedule-reuse rates, and have reasonable overhead when run offline; its recording overhead is relatively high, but can be reduced using existing techniques [13]; and it requires no manual efforts except a few annotations for handling server programs and for improving precision.

Our main contributions are the schedule-relaxation approach and PEREGRINE, an efficient DMT system. Additional contributions include the ideas of hybrid schedules, determinism-preserving slicing, and schedule-guided simplification. To our knowledge, our slicing technique is the first to compute correct (non-trivial) preconditions for multithreaded programs. We believe these ideas apply beyond PEREGRINE (§2.2).

The remainder of this paper is organized as follows. We first present a detailed overview of PEREGRINE (§2). We then describe its core ideas: hybrid schedules (§3), determinism-preserving slicing (§4), and schedule-guided simplification (§5). We then present implementation issues (§6) and evaluation (§7). We finally discuss related work (§8) and conclude (§9).

2 PEREGRINE Overview

Figure 1 shows the architecture of PEREGRINE. It has four main components: the instrumentor, recorder, analyzer, and replayer. The *instrumentor* is an LLVM [2] compiler plugin that

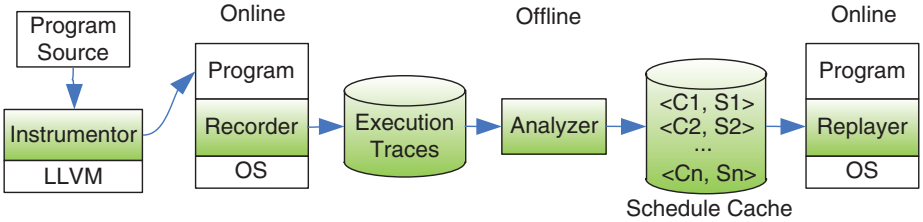

Figure 1: PEREGRINE *Architecture: components and data structures are shaded (and in green).*

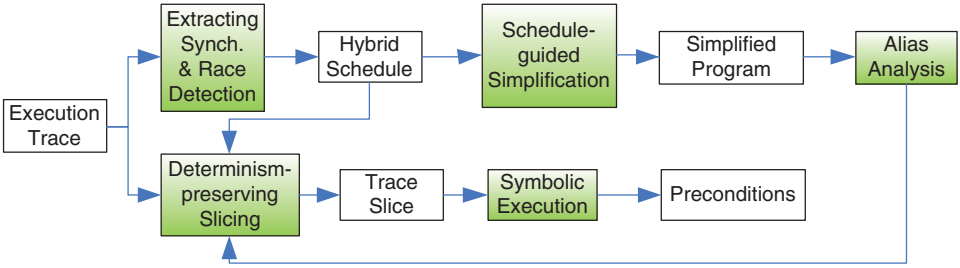

Figure 2: *Analyses performed by the analyzer.*

prepares a program for use with PEREGRINE. It instruments synchronization operations such as `pthread_mutex_lock()`, which the recorder and replayer control at runtime. It marks the `main()` arguments, data read from `read()`, `fscanf()`, and `recv()`, and values returned by `random()`-variants as inputs. We chose LLVM [2] as our instrumentation framework for its compatibility with GCC and easy-to-analyze intermediate representation (IR). However, our approach is general and should apply beyond LLVM. For clarity, we will present our examples and algorithms at the source level, instead of the LLVM IR level.

The *recorder* is similar to existing systems that deterministically record executions [13, 22, 33]. Our current recorder is implemented as an LLVM interpreter. When a program runs, the recorder saves the LLVM instructions interpreted for each thread into a central log file. The recorder does not record external input data, such as data read from a file, because our analysis does not need this information. To schedule synchronization operations issued by different threads, the recorder can use a variety of DMT algorithms [19].

The *analyzer* is a stand-alone program that computes (1) a hybrid schedule S and (2) the preconditions C required for reusing the schedule on future inputs. It does so using a series of analyses, shown in Figure 2. To compute a hybrid schedule, the analyzer first extracts a total order of synchronization operations from the execution trace. It then detects data races according to this synchronization order, and computes additional *execution order* constraints to deterministically resolve the detected races. To compute the preconditions of a schedule, the analyzer first *simplifies* the program according to the schedule, so that alias analysis can compute more precise results. It then slices the execution trace into a trace slice with instructions required to avoid new races and reach all events in the schedule. It then uses *symbolic execution* [31] to collect preconditions from the input-dependent branches in the slice. The trace slice is typically much smaller than the execution trace, so that the analyzer can compute relaxed preconditions, allowing frequent reuses of the schedule. The analyzer finally stores $\langle C, S \rangle$ into the schedule cache, which conceptually holds a set of such tuples. (The actual representation is tree-based for fast lookup [19].)

The *replayer* is a lightweight user-space scheduler for reusing schedules. When an input arrives, it searches the schedule cache for a $\langle C, S \rangle$ tuple such that the input satisfies the preconditions C. If it finds such a tuple, it simply runs the program enforcing schedule S efficiently and deterministically. Otherwise, it forwards the input to the recorder.

In the remainder of this section, we first use an example to illustrate how PEREGRINE works,

```
      int size; // total size of data
      int nthread; // total number of threads
      unsigned long result = 0;

      int main(int argc, char *argv[]) {
L1:       nthread = atoi(argv[1]);
L2:       size = atoi(argv[2]);
L3:       assert(nthread>0 && size>=nthread);
L4:       for(int i=1; i<nthread; ++i)
L5:           pthread_create(..., worker, NULL);
L6:       worker(NULL);
          // NOTE: missing pthread_join()
L7:       if(atoi(argv[3]) == 1)
L8:           result += ...; // race with line L15
L9:       printf("result = %lu\n", result); // race with line L15
          ...
      }
      void *worker(void *arg) {
L10:      char *data = malloc(size/nthread);
L11:      read(..., data, size/nthread);
L12:      for(int i=0; i<size/nthread; ++i)
L13:          data[i] = ...; // compute using data
L14:      pthread_mutex_lock(&mutex);
L15:      result += ...; // race with lines L8 and L9
L16:      pthread_mutex_unlock(&mutex);
          ...
      }
```

Figure 3: *Running example.* It uses the common divide-and-conquer idiom to split work among multiple threads. It contains write-write (lines L8 and L15) and read-write (lines L9 and L15) races on `result` because of missing `pthread_join()`.

highlighting the operation of the analyzer (§2.1). We then describe PEREGRINE's deployment and usage scenarios (§2.2) and assumptions (§2.3).

2.1 An Example

Figure 3 shows our running example, a simple multithreaded program based on the real ones used in our evaluation. It first parses the command line arguments into `nthread` (line L1) and `size` (L2), then spawns `nthread` threads including the main thread (L4–L5) and processes `size/nthread` bytes of data in each thread. The thread function `worker()` allocates a local buffer (L10), reads data from a file (L11), processes the data (L12–L13), and sums the results into the shared variable `result` (L14–L16). The `main()` function may further update `result` depending on `argv[3]` (L7–L8), and finally prints out `result` (L9). This example has read-write and write-write races on `result` due to missing `pthread_join()`. This error pattern matches some of the real errors in the evaluated programs such as `PBZip2`.

Instrumentor. To run this program with PEREGRINE, we first compile it into LLVM IR and instrument it with the instrumentor. The instrumentor replaces the synchronization operations (lines L5, L14, and L16) with PEREGRINE-provided wrappers controlled by the recorder and replayer at runtime. It also inserts code to mark the contents of `argv[i]` and the data from `read()` (line L11) as input.

Recorder: execution trace. When we run the instrumented program with arguments "`2 2 0`" to spawn two threads and process two bytes of data, suppose that the recorder records the execution trace in Figure 4. (This figure also shows the hybrid schedule and preconditions PEREGRINE computes, explained later in this subsection.) This trace is just one possible trace depending on the scheduling algorithm the recorder uses.

Analyzer: hybrid schedule. Given the execution trace, the analyzer starts by computing a hybrid schedule. It first extracts a sync-schedule consisting of the operations tagged with (1), (2), ..., (8) in Figure 4. It then detects races in the trace according to this sync-schedule, and finds the race

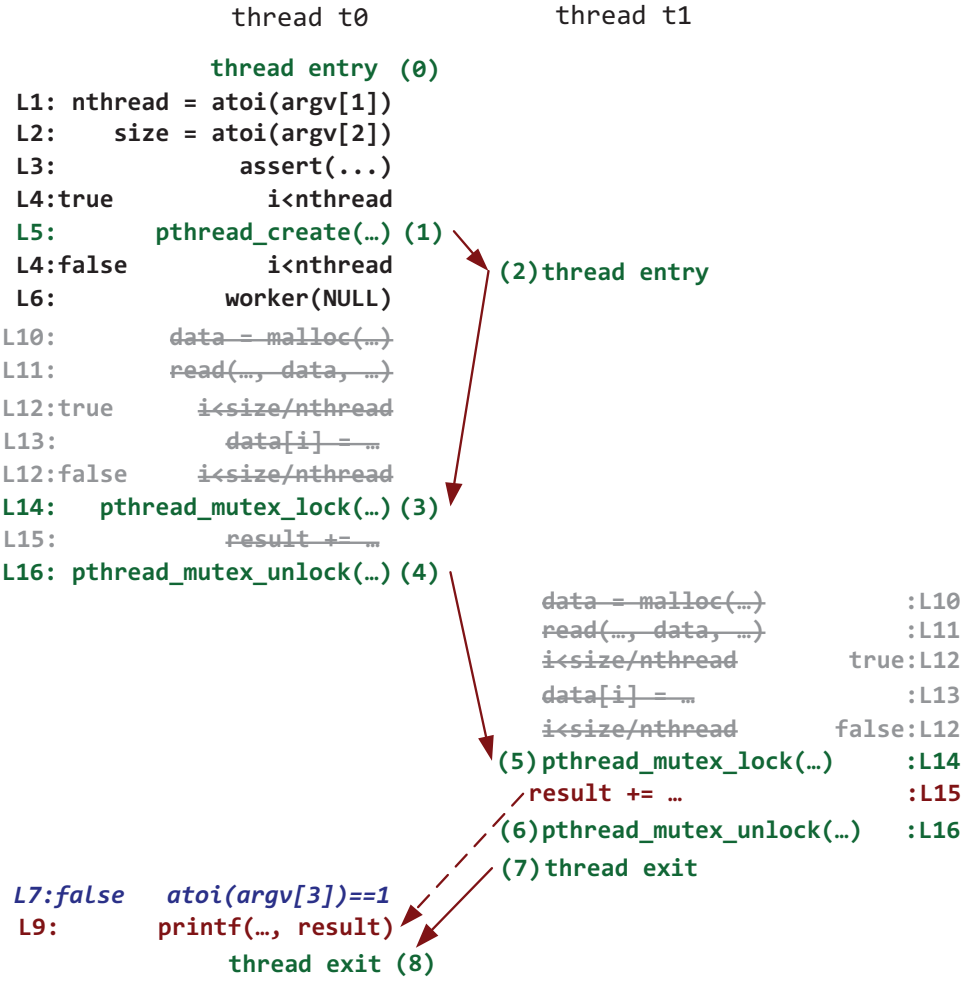

Figure 4: *Execution trace, hybrid schedule, and trace slice.* An execution trace of the program in Figure 3 on arguments "2 2 0" is shown. Each executed instruction is tagged with its static line number Li. Branch instructions are also tagged with their outcome (true or false). Synchronization operations (green), including thread entry and exit, are tagged with their relative positions in the synchronization order. They form a sync-schedule whose order constraints are shown with solid arrows. L15 of thread t_1 and L9 of thread t_0 race on `result`, and this race is deterministically resolved by enforcing an execution order constraint shown by the dotted arrow. Together, these order constraints form a hybrid schedule. Instruction L7 of t_0 (italic and blue) is included in the trace slice to avoid new races, while L6, L4:false, L4:true, L3, L2, and L1 of t_0 are included due to intra-thread dependencies. Crossed-out (gray) instructions are elided from the slice.

on `result` between L15 of thread t_1 and L9 of t_0. It then computes an execution order constraint to deterministically resolve this race, shown as the dotted arrow in Figure 4. The sync-schedule and execution order constraint together form the hybrid schedule. Although this hybrid schedule constrains the order of synchronization and the last two accesses to `result`, it can still be efficiently reused because the core computation done by `worker` can still run in parallel.

Analyzer: simplified program. To improve analysis precision, the analyzer simplifies the program according to the hybrid schedule. For instance, based on the number of `pthread_create()` operations in the schedule, the analyzer clones function `worker()` to give each thread a copy, so that the alias analysis separates different threads and determines that the two instances of L13 in t_0 and

$$(atoi\_argv_1 = 2) \wedge (atoi\_argv_2 \geq atoi\_argv_1) \wedge (atoi\_argv_3 \neq 1)$$

Figure 5: *Preconditions computed from the trace slice in Figure 4.* Variable $atoi\_argv_i$ represents the return of `atoi(arg[i])`.

t_1 access different `malloc`'ed locations and never race.

Analyzer: trace slice. The analyzer uses determinism-preserving slicing to reduce the execution trace into a trace slice, so that it can compute relaxed preconditions. The final trace slice consists of the instructions not crossed out in Figure 4. The analyzer computes this trace slice using inter-thread and intra-thread steps. In the inter-thread step, it adds instructions required to avoid new races into the slice. Specifically, for t_0 it adds the false branch of L7, or L7:false, because if the true branch is taken, a new race between L8 of t_0 and L15 of t_1 occurs. It ignores branches of line L12 because alias analysis already determines that L13 of t_0 and L13 of t_1 never race.

In the intra-thread step, the analyzer adds instructions required to reach all instructions identified in the inter-thread step (L7:false of t_0 in this example) and all events in the hybrid schedule. It does so by traversing the execution trace backwards and tracking control- and data-dependencies. In this example, it removes L15, L13, L12, L11, and L10 because no instructions currently in the trace slice depend on them. It adds L6 because without this call, the execution will not reach instructions L14 and L16 of thread t_0. It adds L4:false because if the true branch is taken, the execution of t_0 will reach one more `pthread_create()`, instead of L14, `pthread_mutex_lock()`, of t_0. It adds L4:true because this branch is required to reach L5, the `pthread_create()` call. It similarly adds L3, L2, and L1 because later instructions in the trace slice depend on them.

Analyzer: preconditions. After slicing, all branches from L12 are gone. The analyzer joins the remaining branches together as the preconditions, using a version of KLEE [15] augmented with thread support [19]. Specifically, the analyzer marks input data as *symbolic*, and then uses KLEE to track how this symbolic data is propagated and observed by the instructions in the trace slice. (Our PEREGRINE prototype runs symbolic execution within the recorder for simplicity; see §6.1.) If a branch instruction inspects symbolic data and proceeds down the true branch, the analyzer adds the precondition that the symbolic data makes the branch condition true. The analyzer uses symbolic summaries [18] to succinctly generalize common library functions. For instance, it considers the return of `atoi(arg)` symbolic if `arg` is symbolic.

Figure 5 shows the preconditions the analyzer computes from the trace slice in Figure 4. These preconditions illustrate two key benefits of PEREGRINE. First, they are sufficient to ensure deterministic reuses of the schedule. Second, they only loosely constrain the data size ($atoi\_argv_2$) and do not constrain the data contents (from `read()`), allowing frequent schedule-reuses. The reason is that L10–L13 are all sliced out. One way to leverage this benefit is to populate a schedule cache with small workloads to reduce analysis time, and then reuse the schedules on large workloads.

Replayer. Suppose we run this program again on different arguments "`2 1000 8`." The replayer checks the new arguments against the preconditions in Figure 5 using KLEE's constraint checker, and finds that these arguments satisfy the preconditions, despite the much larger data size. It can therefore reuse the hybrid schedule in Figure 4 on this new input by enforcing the same order of synchronization operations and accesses to `result`.

2.2 Deployment and Usage Scenarios

PEREGRINE runs in user-space and requires no special hardware, presenting few challenges for deployment. To populate a schedule cache, a user can record execution traces from real workloads; or a developer can run (small) representative workloads to pre-compute schedules before deployment. PEREGRINE efficiently makes the behaviors of multithreaded programs more repeatable, even across a range of inputs. We envision that users can use this repeatability in at least four ways.

Concurrency error avoidance. PEREGRINE can reuse well-tested schedules collected from the testing lab or the field, reducing the risk of running into untested, buggy schedules. Currently PEREGRINE detects and avoids only data races. However, combined with the right error detectors, PEREGRINE can be easily extended to detect and avoid other types of concurrency errors.

Record and replay. Existing deterministic record-replay systems tend to incur high CPU and storage overhead (*e.g.*, 15X slowdown [13] and 11.7 GB/day storage [22]). A record-replay system

on top of PEREGRINE may drastically reduce this overhead: for inputs that hit the schedule cache, we do not have to log any schedule.

Replication. To keep replicas of a multithreaded program consistent, a replication tool often records the thread schedules at one replica and replays them at others. This technique is essentially *online* replay [35]. It may thus incur high CPU, storage, and bandwidth overhead. With PEREGRINE, replicas can maintain a consistent schedule cache. If an input hits the schedule cache, all replicas will automatically select the same deterministic schedule, incurring zero bandwidth overhead.

Schedule-diversification. Replication can tolerate hardware or network failures, but the replicas may still run into the same concurrency error because they all use the same schedules. Fortunately, many programs are already "mostly-deterministic" as they either compute the same correct result or encounter heisenbugs. We can thus run PEREGRINE to deterministically diversify the schedules at different replicas (*e.g.*, using different scheduling algorithms or schedule caches) to tolerate *unknown* concurrency errors,

Applications of individual techniques. The individual ideas in PEREGRINE can also benefit other research efforts. For instance, hybrid schedules can make the sync-schedule approach deterministic without recording executions, by coupling it with a sound static race detector. Determinism-preserving slicing can (1) compute input filters to block bad inputs [18] causing concurrency errors and (2) randomize an input causing a concurrency error for use with anonymous bug reporting [16]. Schedule-guided simplification can transparently improve the precision of many existing static analyses: simply run them on the simplified programs. This improved precision may be leveraged to accurately detect errors or even verify the correctness of a program according to a set of schedules. Indeed, from a verification perspective, our simplification technique helps *verify* that executions reusing schedules have *no* new races.

2.3 Assumptions

At a design level, we anticipate the schedule-relaxation approach to work well for many programs/workloads as long as (1) they can benefit from repeatability, (2) their schedules can be frequently reused, (3) their races are rare, and (4) their nondeterminism comes from the sources tracked by PEREGRINE. This approach is certainly not designed for every multithreaded program. For instance, like other DMT systems, PEREGRINE should not be used for parallel simulators that desire nondeterminism for statistical confidence. For programs/workloads that rarely reuse schedules, PEREGRINE may be unable to amortize the cost of recording and analyzing execution traces. For programs full of races, enforcing hybrid schedules may be as slow as mem-schedules. PEREGRINE addresses nondeterminism due to thread scheduling and data races. It mitigates input nondeterminism by reusing schedules on different inputs. It currently considers command line arguments, data read from a file or a socket, and the values returned by `random()`-variants as inputs. PEREGRINE ensures that schedule-reuses are fully deterministic if a program contains only these nondeterminism sources, an assumption met by typical programs. If a program is nondeterministic due to other sources, such as functions that query physical time (*e.g.*, `gettimeofday()`), pointer addresses returned by `malloc()`, and nondeterminism in the kernel or external libraries, PEREGRINE relies on developers to annotate these sources.

The underlying techniques that PEREGRINE leverages make assumptions as well. PEREGRINE computes preconditions from a trace slice using the symbolic execution engine KLEE, which does not handle floating point operations; though recent work [17] has made advances in symbolic execution of floating point programs. (Note that floating point operations not in trace slices are not an issue.) We explicitly designed PEREGRINE's slicing technique to compute sufficient preconditions, but these preconditions may still include unnecessary ones, because computing the *weakest* (most relaxed) preconditions in general is undecidable [4]. The alias analysis PEREGRINE uses makes a few assumptions about the analyzed programs [8]; a "sounder" alias analysis [28] would remove these assumptions. These analyses may all get expensive for large programs. For server programs, PEREGRINE borrows the windowing idea from our previous work [19]; it is thus similarly limited (§6.3).

At an implementation level, PEREGRINE uses the LLVM framework, thus requiring that a program is in either source (so we can compile using LLVM) or LLVM IR format. PEREGRINE ignores inline x86 assembly or calls to external functions it does not know. For soundness, developers have to lift

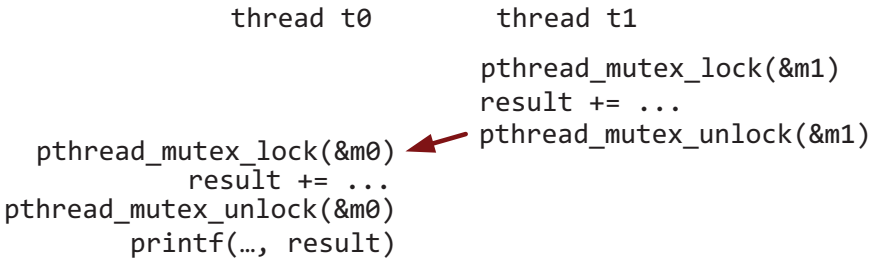

Figure 6: *No* PEREGRINE *race with respect to this schedule.*

x86 assembly to LLVM IR and provide summaries for external functions. (The external function problem is alleviated because KLEE comes with a `Libc` implementation.) Currently PEREGRINE works only with a single process, but previous work [10] has demonstrated how DMT systems can be extended to multiple processes.

3 Hybrid Schedules

This section describes how PEREGRINE computes (§3.1) and enforces (§3.2) hybrid schedules.

3.1 Computing Hybrid Schedules

To compute a hybrid schedule, PEREGRINE first extracts a total order of synchronization operations from an execution trace. Currently, it considers 28 `pthread` operations, such as `pthread_mutex_lock()` and `pthread_cond_wait()`. It also considers the entry and exit of a thread as synchronization operations so that it can order these events together with other synchronization operations. These operations are sufficient to run the programs evaluated, and more can be easily added. PEREGRINE uses a total, instead of a partial, order because previous work has shown that a total order is already efficient [19, 42].

For determinism, PEREGRINE must detect races that occurred during the recorded execution and compute execution order constraints to deterministically resolve the races. An off-the-shelf race detector would flag too many races because it considers the original synchronization constraints of the program, whereas PEREGRINE wants to detect races according to a sync-schedule [44, 45]. To illustrate, consider Figure 6, a modified sync-schedule based on the one in Figure 4. Suppose the two threads acquire different mutex variables, and thread t_1 acquires and releases its mutex before t_0. Typical lockset-based [47] or happens-before-based [34] race detectors would flag a race on `result`, but our race detector does not: the sync-schedule in the figure deterministically resolves the order of accesses to `result`. Sync-schedules anecdotally reduced the number of possible races greatly, in one extreme case, from more than a million to four [44].

Mechanically, PEREGRINE detects occurred races using a happens-before-based algorithm. It flags two memory accesses as a race iff (1) they access the same memory location and at least one is a `store` and (2) they are *concurrent*. To determine whether two accesses are concurrent, typical happens-before-based detectors use vector clocks [40] to track logically when the accesses occur. Since PEREGRINE already enforces a total synchronization order, it uses a simpler and more memory-efficient logical clock representation.

Specifically, given two adjacent synchronization operations within one thread with relative positions m and n in the sync-schedule, PEREGRINE uses $[m, n)$ as the logical clock of all instructions executed by the thread between the two synchronization operations. For instance, in Figure 4, all instructions run by thread t_0 between the `pthread_mutex_unlock()` operation and the thread exit have clock $[4, 8)$. PEREGRINE considers two accesses with clocks $[m_0, n_0)$ and $[m_1, n_1)$ concurrent if the two clock ranges overlap, *i.e.*, $m_0 < n_1 \wedge m_1 < n_0$. For instance, $[4, 8)$ and $[5, 6)$ are concurrent.

To deterministically resolve a race, PEREGRINE enforces an execution order constraint $inst_1 \rightarrow inst_2$ where $inst_1$ and $inst_2$ are the two dynamic instruction instances involved in the race. PEREGRINE identifies a dynamic instruction instance by $\langle sid, tid, nbr \rangle$ where sid refers to the

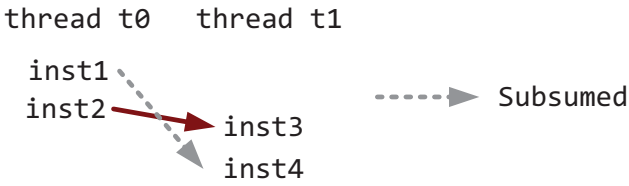

Figure 7: *Example subsumed execution order constraint.*

unique ID of a static instruction in the executable file; tid refers to the internal thread ID maintained by PEREGRINE, which always starts from zero and increments deterministically upon each `pthread_create()`; and nbr refers to the number of control-transfer instructions (branch, call, and return) locally executed within the thread from the last synchronization to instruction $inst_i$. For instance, PEREGRINE represents the execution order constraint in Figure 4 as $\langle L15, t_1, 0\rangle \rightarrow \langle L9, t_0, 2\rangle$, where the branch count 2 includes the return from `worker` and the branch L7 of thread t_0. We must distinguish different dynamic instances of a static instruction because some of these dynamic instances may be involved in races while others are not. We do so by counting branches because if an instruction is executed twice, there must be a control-transfer between the two instances [22]. We count branches starting from the last synchronization operation because the partial schedule preceding this operation is already made deterministic.

If one execution order constraint subsumes another, PEREGRINE does not add the subsumed one to the schedule. Figure 7 shows a subsumed constraint example. Algorithmically, PEREGRINE considers an execution order constraint $inst_1 \rightarrow inst_4$ subsumed by $inst_2 \rightarrow inst_3$ if (1) $inst_1$ and $inst_2$ have the same logical clock (so they must be executed by the same thread) and $inst_2$ occurs no earlier than $inst_1$ in the recorded execution trace; (2) $inst_3$ and $inst_4$ have the same logical clock and $inst_3$ occurs no later than $inst_4$ in the trace. This algorithm ignores transitive order constraints, so it may miss some subsumed constraints. For instance, it does not consider $inst_1 \rightarrow inst_4$ subsumed if we replace constraint $inst_2 \rightarrow inst_3$ with $inst_2 \rightarrow inst_{other}$ and $inst_{other} \rightarrow inst_3$ where $inst_{other}$ is executed by a third thread.

3.2 Enforcing Hybrid Schedules

To enforce a synchronization order, PEREGRINE uses a technique called *semaphore relay* [19] that orders synchronization operations with per-thread semaphores. At runtime, a synchronization wrapper (recall that PEREGRINE instruments synchronization operations for runtime control) waits on the semaphore of the current thread. Once it is woken up, it proceeds with the actual synchronization operation, then wakes up the next thread according to the synchronization order. For programs that frequently do synchronization operations, the overhead of semaphore may be large because it may cause a thread to block. Thus, PEREGRINE also provides a spin-wait version of semaphore relay called *flag relay*. This technique turns out to be very fast for many programs evaluated (§7.2).

To enforce an execution order constraint, PEREGRINE uses program instrumentation, avoiding the need for special hardware, such as the often imprecise hardware branch counters [22]. Specifically, given a dynamic instruction instance $\langle sid, tid, nbr\rangle$, PEREGRINE instruments the static instruction sid with a semaphore `up()` or `down()` operation. It also instruments the branch instructions counted in nbr so that when each of these branch instructions runs, a per-thread branch counter is incremented. PEREGRINE activates the inserted semaphore operation for thread tid only when the thread's branch counter matches nbr. To avoid interference and unnecessary contention when there are multiple order constraints, PEREGRINE assigns a unique semaphore to each constraint.

PEREGRINE instruments a program by leveraging a fast instrumentation framework we previously built [53]. It keeps two versions of each basic block: a normally compiled, fast version, and a slow backup padded with calls to a `slot()` function before each instruction. As shown in Figure 8, the `slot()` function interprets the actions (semaphore `up/down`) to be taken at each instruction. To instrument an instruction, PEREGRINE simply updates the actions for that instruction. This instrumentation may be expensive, but fortunately, PEREGRINE leaves it off most of the time and turns it on only at the last synchronization operation before an inserted semaphore operation.

```
void slot(int sid) { // sid is static instruction id
  if(instruction sid is branch)
    nbr[self()] ++; // increment per-thread branch counter
  // get semaphore operations for current thread at instruction sid
  my_actions = actions[sid][self()];
  for action in my_actions
    if nbr[self()] == action.nbr // check branch counter
      actions.do(); // perform up() or down()
}
```

Figure 8: *Instrumentation to enforce execution order constraints.*

PEREGRINE turns on/off this instrumentation by switching a per-thread flag. Upon each function entry, PEREGRINE inserts code to check this flag and determine whether to run the normal or slow version of the basic blocks. PEREGRINE also inserts this check after each function returns in case the callee has switched the per-thread flag. The overhead of these checks tend to be small because the flags are rarely switched and hardware branch predication works well in this case [53].

One potential issue with branch-counting is that PEREGRINE has to "fix" the partial path from the last synchronization to the dynamic instruction instance involved in a race so that the branch-counts match between the recorded execution and all executions reusing the extracted hybrid schedule, potentially reducing schedule-reuse rates. Fortunately, races are rare, so this issue has not reduced PEREGRINE's schedule-reuse rates based on our evaluation.

4 Determinism-Preserving Slicing

PEREGRINE uses determinism-preserving slicing to (1) compute sufficient preconditions to avoid new races and ensure that a schedule is feasible, and (2) filter many unnecessary preconditions to increase schedule-reuse rates. It does so using inter- and intra-thread steps. In the inter-thread step (§4.1), it detects and avoids *input-dependent* races that do not occur in the execution trace, but may occur if we reuse the schedule on a different input. In the intra-thread step (§4.1), the analyzer computes a *path slice* per thread by including instructions that may affect the events in the schedule or the instructions identified in the inter-thread step.

4.1 Inter-thread Step

In the inter-thread step, PEREGRINE detects and avoids input-dependent races with respect to a hybrid schedule. An example input-dependent race is the one between lines L8 and L15 in Figure 3, which occurs when `atoi(argv[3])` returns 1 causing the true branch of L7 to be taken. Figure 9 shows two more types of input-dependent races.

To detect such races, PEREGRINE starts by refining the logical clocks computed based on the sync-schedule (§3.1) with execution order constraints because it will also enforce these constraints. PEREGRINE then iterates through all pairs of concurrent *regions*, where a region is a set of instructions with an identical logical clock. For each pair, it detects input-dependent races, and adds the racy instructions to a list of *slicing targets* used by the intra-thread step.

Figure 10 shows the algorithm to detect input-dependent races for two concurrent regions. The algorithm iterates through each pair of instructions respectively from the two regions, and handles three types of input-dependent races. First, if neither instruction is a branch instruction, it queries alias analysis to determine whether the instructions *may* race. If so, it adds both instructions to `slicing_targets` and adds additional preconditions to ensure that the pointers dereferenced are different, so that reusing the schedule on a different input does not cause the may-race to become a real race. Figure 9(a) shows a race of this type.

Second, if exactly one of the instructions is a branch instruction, the algorithm checks whether the

```
// thread t1        // thread t2
a[input1]++;        a[input2] = 0;
```

(a)

```
// thread t1        // thread t2
if(input1==0)       if(input2==0)
    a++;                a = 0;
```

(b)

Figure 9: *Input-dependent races.* Race (a) occurs when `input1` and `input2` are the same; Race (b) occurs when both true branches are taken.

```
// detect input-dependent races, and add involved dynamic instruction
// instances to slicing_targets used by the inter-thread step. r1 and
// r2 are two concurrent regions
void detect_input_dependent_races(r1, r2) {
  // iterate through all instruction pairs in r1, r2
  for (i1, i2) in (r1, r2) {
    if (neither i1 nor i2 is a branch instruction) {
      if(mayrace(i1, i2)) {
        slicing_targets.add(i1); // add i1 to slicing targets
        slicing_targets.add(i2); // add i2 to slicing targets
      }
    } else if (exactly one of i1, i2 is a branch instruction) {
      br = branch instruction in i1, i2;
      inst = the other instruction in i1, i2;
      nottaken = the not taken branch of br in the execution trace;
      if(mayrace_br(br, nottaken, inst)) {
        // add the taken branch of br to slicing targets
        taken = the taken branch of br in trace;
        slicing_targets.add_br(br, taken);
      }
    } else { // both i1, i2 are branches
      nottaken1 = the not taken branch of i1 in trace;
      nottaken2 = the not taken branch of i2 in trace;
      if(mayrace_br_br(i1, nottaken1, i2, nottaken2) {
        taken1 = the taken branch of i1 in trace;
        slicing_targets.add_br(i1, taken1);
      }
    }
  }
}
// return true if instructions i1 and i2 may race
bool mayrace(i1, i2) {
  // query alias analysis
  return mayalias(i1, i2) && ((i1 is a store) || (i2 is a store));
}
// return true if the not-taken branch of br may race with inst
bool mayrace_br(br, nottaken, inst) {
  for i in (instructions in the nottaken branch of br) {
    if(mayrace(i, inst))
      return true;
  }
  return false;
}
// return true if the not-taken branch of br1 may race with the
// not-taken branch of br2
bool mayrace_br_br(br1, nottaken1, br2, nottaken2) {
  for inst in (instructions in the nottaken2 branch of br2) {
    if(mayrace_br(br1, nottaken1, inst))
      return true;
  }
  return false;
}
```

Figure 10: *Input-dependent race detection algorithm.*

instructions contained in the not-taken branch[1] of this instruction may race with the other instruction. It must check the not-taken branch because a new execution may well take the not-taken branch and cause a race. To avoid such a race, PEREGRINE adds the taken branch into the trace slice so that executions reusing the schedule always go down the taken branch. For instance, to avoid the input-dependent race between lines L8 and L15 in Figure 3, PEREGRINE includes the false branch of L7 in the trace slice.

[1] PEREGRINE computes instructions contained in a not-taken branch using an interprocedural *post-dominator analysis* [4].

Third, if both instructions are branch instructions, the algorithm checks whether the not-taken branches of the instructions may race, and if so, it adds either taken branch to `slicing_targets`. For instance, to avoid the race in Figure 9(b), PEREGRINE includes one of the false branches in the trace slice.

For efficiency, PEREGRINE avoids iterating through all pairs of instructions from two concurrent regions because instructions in one region often repeatedly access the same memory locations. Thus, PEREGRINE computes memory locations read or written by all instructions in one region, then checks whether instructions in the other region also read or write these memory locations. These locations are static operands, not dynamic addresses [14], so that PEREGRINE can aggressively cache them per static function or branch. The complexity of our algorithm thus drops from $O(MN)$ to $O(M + N)$ where M and N are the numbers of memory instructions in the two regions respectively.

4.2 Intra-thread Step

In the intra-thread step, PEREGRINE leverages a previous algorithm [18] to compute a per-thread path slice, by including instructions required for the thread to reach the `slicing_targets` identified in the inter-thread step and the events in the hybrid schedule. To do so, PEREGRINE first prepares a per-thread ordered target list by splitting `slicing_targets` and events in the hybrid schedule and sorting them based on their order in the execution trace.

PEREGRINE then traverses the execution trace backwards to compute path slices. When it sees a target, it adds the target to the path slice of the corresponding thread, and starts to track the control- and data-dependencies of this target.[2] PEREGRINE adds a branch instruction to the path slice if taking the opposite branch may cause the thread not to reach any instruction in the current (partial) path slice; L3 in Figure 4 is added for this reason. It adds a non-branch instruction to the path slice if the result of this instruction may be used by instructions in the current path slice; L1 in Figure 4 is added for this reason.

A "`load p`" instruction may depend on an earlier "`store q`" if `p` and `q` may alias even though `p` and `q` may not be the same in the execution trace, because an execution on a different input may cause `p` and `q` to be the same. Thus, PEREGRINE queries alias analysis to compute such *may*-dependencies and include the depended-upon instructions in the trace slice.

Our main modification to [18] is to slice toward multiple ordered targets. To illustrate this need, consider branch L4:false of t_0 in Figure 4. PEREGRINE must add this branch to thread t_0's slice, because otherwise, the thread would reach another `pthread_create()`, a different synchronization operation than the `pthread_mutex_lock()` operation in the schedule.

The choice of LLVM IR has considerably simplified our slicing implementation. First, LLVM IR limits memory access to only two instructions, `load` and `store`, so that our algorithms need consider only these instructions. Second, LLVM IR uses an unlimited number of virtual registers, so that our analysis does not get poisoned by stack spilling instructions. Third, each virtual register is defined exactly once, and multiple definitions to a variable are merged using a special instruction. This representation (*static single assignment*) simplifies control- and data-dependency tracking. Lastly, the type information LLVM IR preserves helps improving the precision of the alias analysis.

5 Schedule-Guided Simplification

In both the inter- and intra-thread steps of determinism-preserving slicing, PEREGRINE frequently queries alias analysis. The inter-thread step needs alias information to determine whether two instructions may race (`mayalias()` in Figure 10). The intra-thread step needs alias information to track potential dependencies.

We thus integrated `bddbddb` [51, 52], one of the best alias analyses, into PEREGRINE by creating an LLVM frontend to collect program facts into the format `bddbddb` expects. However, our initial evaluation showed that `bddbddb` sometimes yielded overly imprecise results, causing PEREGRINE to prune few branches, reducing schedule-reuse rates (§7.3). The cause of the imprecision is that

[2]For readers familiar with precondition slicing, PEREGRINE does not always track data-dependencies for the operands of a target. For instance, consider instruction L9 of thread t_0 in Figure 4. PEREGRINE's goal is to deterministically resolve the race involving L9 of t_0, but it allows the value of `result` to be different. Thus, PEREGRINE does not track dependencies for the value of `result`; L15 of t_0 is elided from the slice for this reason.

standard alias analysis is purely static, and has to be conservative and assume all possible executions. However, PEREGRINE requires alias results only for the executions that may reuse a schedule, thus suffering from unnecessary imprecision of standard alias analysis.

To illustrate, consider the example in Figure 3. Since the number of threads is determined at runtime, static analysis has to abstract this unknown number of dynamic thread instances, often coalescing results for multiple threads into one. When PEREGRINE slices the trace in Figure 4, `bddbddb` reports that the accesses to `data` (L13 instances) in different threads may alias. PEREGRINE thus has to add them to the trace slice to avoid new races (§4.1). Since L13 depends on L12, L11, and L10, PEREGRINE has to add them to the trace slice, too. Eventually, an imprecise alias result snowballs into a slice as large as the trace itself. The preconditions from this slice constrains the data size to be exactly 2, so PEREGRINE cannot reuse the hybrid schedule in Figure 4 on other data sizes.

To improve precision, PEREGRINE uses schedule-guided simplification to simplify a program according to a schedule, so that alias analysis is less likely to get confused. Specifically, PEREGRINE performs three main simplifications:

1. It clones the functions as needed. For instance, it gives each thread in a schedule a copy of the thread function.
2. It unrolls a loop when it can determine the loop bound based on a schedule. For instance, from the number of the `pthread_create()` operations in a schedule, it can determine how many times the loop at lines L4–L5 in Figure 3 executes.
3. It removes branches that contradict the schedule. Loop unrolling can be viewed as a special case of this simplification.

PEREGRINE does all three simplifications using one algorithm. From a high level, this algorithm iterates through the events in a schedule. For each pair of adjacent events, it checks whether they are "at the same level," *i.e.*, within the same function and loop iteration. If so, PEREGRINE does not clone anything; otherwise, PEREGRINE clones the mismatched portion of instructions between the events. (To find these instructions, PEREGRINE uses an interprocedural reachability analysis by traversing the control flow graph of the program.) Once these simplifications are applied, PEREGRINE can further simplify the program by running stock LLVM transformations such as constant folding. It then feeds the simplified program to `bddbddb`, which can now distinguish different thread instances (*thread-sensitivity* in programing language terms) and precisely reports that L13 of t_0 and L13 of t_1 are not aliases, enabling PEREGRINE to compute the small trace slice in Figure 4.

By simplifying a program, PEREGRINE can automatically improve the precision of not only alias analysis, but also other analyses. We have implemented *range analysis* [46] to improve the precision of alias analysis on programs that divide a global array into disjoint partitions, then process each partition within a thread. The accesses to these disjoint partitions from different threads do not alias, but `bddbddb` often collapses the elements of an array into one or two abstract locations, and reports the accesses as aliases. Range analysis can solve this problem by tracking the lower and upper bounds of the integers and pointers. With range analysis, PEREGRINE answers alias queries as follows. Given two pointers `(p+i)` and `(q+i)`, it first queries `bddbddb` whether `p` and `q` may alias. If so, it queries the more expensive range analysis whether `p+i` and `q+j` may be equal. It considers the pointers as aliases only when both queries are true. Note that our simplification technique is again key to precision because standard range analysis would merge ranges of different threads into one.

While schedule-guided simplification improves precision, PEREGRINE has to run alias analysis for each schedule, instead of once for the program. This analysis time is reasonable as PEREGRINE's analyzer runs offline. Nonetheless, the simplified programs PEREGRINE computes for different schedules are largely the same, so a potential optimization is to *incrementally* analyze a program, which we leave for future work.

6 Implementation Issues

6.1 Recording an Execution

To record an execution trace, PEREGRINE can use one of the existing deterministic record-replay systems [13, 22, 33] provided that PEREGRINE can extract an instruction trace. For simplicity, we have built a crude recorder on top of the LLVM interpreter in KLEE. When an program calls the PEREGRINE-provided wrapper to `pthread_create(..., func, args)`, the recorder spawns a thread

to run `func(args)` within an interpreter instance. These interpreter instances log each instruction interpreted into a central file. For simplicity, PEREGRINE does symbolic execution during recording because it already runs KLEE when recording an execution and pays the high overhead of interpretation. A faster recorder would enable PEREGRINE to symbolically execute only the trace slices instead of the typically larger execution traces. Since deterministic record-replay is a well studied topic, we have not focused our efforts on optimizing the recorder.

6.2 Handling Blocking System Calls

Blocking system calls are natural scheduling points, so PEREGRINE includes them in the schedules [19]. It currently considers eight blocking system calls, such as `sleep()`, `accept()`, and `read()`. For each blocking system call, the recorder logs when the call is issued and when the call is returned. When PEREGRINE computes a schedule, it includes these blocking system call and return operations. When reusing a schedule, PEREGRINE attempts to enforce the same call and return order. This method works well for blocking system calls that access local state, such as `sleep()` or `read()` on local file descriptors. However, other blocking system calls receive input from the external world, which may or may not arrive each time a schedule is reused. Fortunately, programs that use these operations tend to be server programs, and PEREGRINE handles this class of programs differently.

6.3 Handling Server Programs

Server programs present two challenges for PEREGRINE. First, they are more prone to timing nondeterminism than batch programs because their inputs (client requests) arrive nondeterministically. Second, they often run continuously, making their schedules too specific to reuse.

PEREGRINE addresses these challenges with the *windowing* idea from our previous work [19]. The insight is that server programs tend to return to the same quiescent states. Thus, instead of processing requests as they arrive, PEREGRINE breaks a continuous request stream down into windows of requests. Within each window, it admits requests only at fixed points in the current schedule. If no requests arrive at an admission point for a predefined timeout, PEREGRINE simply proceeds with the partial window. While a window is running, PEREGRINE buffers newly arrived requests so that they do not interfere with the running window. With windowing, PEREGRINE can record and reuse schedules across windows.

PEREGRINE requires developers to annotate points at which request processing begins and ends. It also assumes that after a server processes all current requests, it returns to the same quiescent state. That is, the input from the requests does not propagate further after the requests are processed. The same assumption applies to the data read from local files. For server programs not meeting this assumption, developers can manually annotate the functions that observe the changed server state, so that PEREGRINE can consider the return values of these functions as input. For instance, since `Apache` caches client requests, we made it work with PEREGRINE by annotating the return of `cache_find()` as input.

One limitation of applying our PEREGRINE prototype to server programs is that our current implementation of schedule-guided simplification does not work well with thread pooling. To give each thread a copy of the corresponding thread function, PEREGRINE identifies `pthread_create(...,func,...)` operations in a program and clones function `func`. Server programs that use thread pooling tend to create worker threads to run generic thread functions during program initialization, then repeatedly use the threads to process client requests. Cloning these generic thread functions thus helps little with precision. One method to solve this problem is to clone the relevant functions for processing client requests. We have not implemented this method because the programs we evaluated include only one server program, `Apache`, on which slicing already performs reasonably well without simplification (§7.3).

6.4 Skipping Wait Operations

When reusing a schedule, PEREGRINE enforces a total order of synchronization operations, which subsumes the execution order enforced by the original synchronization operations. Thus, for speed, PEREGRINE can actually skip the original synchronization operations as in [19]. PEREGRINE currently skips sleep-related operations such as `sleep()` and wait-related operations such as `pthread_barrier_wait()`. These operations often unconditionally block the calling thread, in-

Program	Race Description
`Apache`	Reference count decrement and check against 0 are not atomic, resulting in a program crash.
`PBZip2`	Variable `fifo` is used by one thread after being freed by another thread, resulting in a program crash.
`barnes`	Variable `tracktime` is read by one thread before assigned the correct value by another thread.
`fft`	`initdonetime` and `finishtime` are read by one thread before assigned the correct values by another thread.
`lu-non-contig`	Variable `rf` is read by one thread before assigned the correct value by another thread.
`streamcluster`	PARSEC has a custom barrier implementation that synchronizes using a shared integer flag `is_arrival_phase`.
`racey`	Numerous intentional races caused by multiple threads reading and writing global arrays `sig` and `m` without synchronization.

Table 1: *Programs used for evaluating* PEREGRINE*'s determinism.*

curring context switch overhead, yet this blocking is unnecessary as PEREGRINE already enforces a correct execution order. Our evaluation shows that skipping blocking operations significantly speeds up executions.

6.5 Manual Annotations

PEREGRINE works automatically for most of the programs we evaluated. However, as discussed in §6.3, it requires manual annotations for server programs. In addition, if a program has nondeterminism sources beyond what PEREGRINE automatically tracks, developers should annotate these sources with `input(void* addr, size_t nbyte)` to mark `nbyte` of data starting from `addr` as input, so that PEREGRINE can track this data.

Developers can also supply optional annotations to improve PEREGRINE's precision in four ways. First, for better alias results, developers can add custom memory allocators and `memcpy`-like functions to a configuration file of PEREGRINE. Second, they can help PEREGRINE better track ranges by adding `assert()` statements. For instance, a function in the FFT implementation we evaluated uses bit-flip operations to transform an array index into another, yet both indexes have the same range. The range analysis we implemented cannot precisely track these bit-flip operations, so it assumes the resultant index is unbounded. Developers can fix this problem by annotating the range of the index with an assertion "`assert(index<bound)`." Third, they can provide *symbolic summaries* to help PEREGRINE compute more relaxed constraints. For instance, consider Figure 5 and a typical implementation of `atoi()` that iterates through all characters in the input string and checks whether each character is a digit. Without a summary of `atoi()`, PEREGRINE would symbolically execute the body of `atoi()`. The preconditions it computes for `argv[3]` would be $(argv_{3,0} \neq 49) \wedge (argv_{3,1} < 48 \vee argv_{3,1} > 57)$, where $argv_{3,i}$ is the ith byte of `argv[3]` and 48, 49, and 57 are ASCII codes of '0', '1', and '9'. These preconditions thus unnecessarily constrain `argv[3]` to have a valid length of one. Another example is string search. When a program calls `strstr()`, it often concerns whether there exists a match, not specifically where the match occurs. Without a symbolic summary of `strstr()`, the preconditions from `strstr()` would constrain the exact location where the match occurs. Similarly, if a trace slice contains complex code such as a decryption function, users can provide a summary of this function to mark the decrypted data as symbolic when the argument is symbolic. Note that complex code not included in trace slices, such as the `read()` in Figure 3, is not an issue.

7 Evaluation

Our PEREGRINE implementation consists of 29,582 lines of C++ code, including 1,338 lines for the recorder; 2,277 lines for the replayer; and 25,967 lines for the analyzer. The analyzer further splits into 7,845 lines for determinism-preserving slicing, 12,332 lines for schedule-guided simplification, and 5,790 lines for our LLVM frontend to `bddbddb`.

We evaluated our PEREGRINE implementation on a diverse set of 18 programs, including `Apache`, a popular web server; `PBZip2`, a parallel compression utility; `aget`, a parallel `wget`-like utility; `pfscan`, a parallel `grep`-like utility; parallel implementations of 13 computation-intensive algorithms, 10 in SPLASH2 and 3 in PARSEC; and `racey`, a benchmark specifically designed to exercise

Program	Races	Order Constraints
Apache	0	0
PBZip2	4	3
barnes	5	1
fft	10	4
lu-non-contig	10	7
streamcluster	0	0
racey	167974	9963

Table 2: *Hybrid schedule statistics.* Column **Races** shows the number of races detected according the corresponding sync-schedule, and Column **Order Constraints** shows the number of execution order constraints PEREGRINE adds to the final hybrid schedule. The latter can be smaller than the former because PEREGRINE prunes subsumed execution order constraints (§3). PEREGRINE detected no races for `Apache` and `streamcluster` because the corresponding sync-schedules are sufficient to resolve the races deterministically; it thus adds no order constraints for these programs.

deterministic execution and replay systems [29]. All SPLASH2 benchmarks were included except one that we cannot compile, one that our current prototype cannot handle due to an implementation bug, and one that does not run correctly in 64-bit environment. The chosen PARSEC benchmarks (`blackscholes`, `swaptions` and `streamcluster`) include the ones that (1) we can compile, (2) use threads, and (3) use no x86 inline assemblies. These programs were widely used in previous studies (*e.g.*, [12, 39, 54]).

Our evaluation machine was a 2.67 GHz dual-socket quad-core Intel Xeon machine with 24 GB memory running Linux 2.6.35. When evaluating PEREGRINE on `Apache` and `aget`, we ran the evaluated program on this machine and the corresponding client or server on another to avoid contention between the programs. These machines were connected via 1Gbps LAN. We compiled all programs to machine code using `llvm-gcc -O2` and the LLVM compiler `llc`. We used eight worker threads for all experiments.

Unless otherwise specified, we used the following workloads in our experiments. For `Apache`, we used `ApacheBench` [1] to repeatedly download a 100 KB webpage. For `PBZip2`, we compressed a 10 MB randomly generated text file. For `aget`, we downloaded a 77 MB file (`Linux-3.0.1.tar.bz2`). For `pfscan`, we scanned the keyword `return` from 100 randomly chosen files in GCC. For SPLASH2 and PARSEC programs, we ran workloads which typically completed in 1-100 ms.

In the remainder of this section, we focus on four questions:

§7.1: Is PEREGRINE deterministic if there are data races? Determinism is one of the strengths of PEREGRINE over the sync-schedule approach.

§7.2: Is PEREGRINE fast? For typical multithreaded programs that have rare data races, PEREGRINE should be roughly as fast as the sync-schedule approach. Efficiency is one of the strengths of PEREGRINE over the mem-schedule approach.

§7.3: Is PEREGRINE stable? That is, can it frequently reuse schedules? The higher the reuse rate, the more repeatable program behaviors become and the more PEREGRINE can amortize the cost of computing hybrid schedules.

§7.4: Can PEREGRINE significantly reduce manual annotation overhead? Recall that our previous work [19] required developers to manually annotate the input affecting schedules.

7.1 Determinism

We evaluated PEREGRINE's determinism by checking whether PEREGRINE could deterministically resolve races. Table 1 lists the seven racy programs used in this experiment. We selected the first five because they were frequently used in previous studies [37, 39, 43, 44] and we could reproduce their races on our evaluation machine. We selected the integer flag race in PARSEC to test whether PEREGRINE can handle ad hoc synchronization [54]. We selected `racey` to stress test PEREGRINE: each run of `racey` may have thousands of races, and if any of these races is resolved differently, `racey`'s final output changes with high probability [29].

For each program with races, we recorded an execution trace and computed a hybrid schedule from the trace. Table 2 shows for each program (1) the number of dynamic races detected according to the sync-schedule and (2) the number of execution order constraints in the hybrid schedule. The

Program	Deterministic?	
	sync-schedule	hybrid schedule
Apache	✔	✔
PBZip2	✖	✔
barnes	✖	✔
fft	✖	✔
lu-non-contig	✖	✔
streamcluster	✔	✔
racey	✖	✔

Table 3: *Determinism of sync-schedules v.s. hybrid schedules.*

reduction from the former to the latter shows how effectively PEREGRINE can prune redundant order constraints (§3). In particular, PEREGRINE prunes 94% of the constraints for `racey`. For `Apache` and `streamcluster`, their races are already resolved deterministically by their sync-schedules, so PEREGRINE adds no execution order constraints.

To verify that the hybrid schedules PEREGRINE computed are deterministic, we first manually inspected the order constraints PEREGRINE added for each program except `racey` (because it has too many races for manual verification). Our inspection results show that these constraints are sufficient to resolve the corresponding races. We then re-ran each program including `racey` 1000 times while enforcing the hybrid schedule and injecting delays; and verified that each run reused the schedule and computed equivalent results. (We determined result equivalence by checking either the output or whether the program crashed.)

We also compared the determinism of PEREGRINE to our previous work [19] which only enforces sync-schedules. Specifically, we reran the seven programs with races 50 times enforcing only the sync-schedules and injecting delays, and checked whether the reuse runs computed equivalent results as the recorded run. As shown in Table 3, sync-schedules are unsurprisingly deterministic for `Apache` and `streamcluster`, because no races are detected according to the corresponding sync-schedules. However, they are not deterministic for the other five programs, illustrating one advantage of PEREGRINE over the sync-schedule approach.

7.2 Efficiency

Replayer overhead. The most performance-critical component is the replayer because it operates within a deployed program. Figure 11 shows the execution times when reusing hybrid schedules; these times are normalized to the nondeterministic execution time. (The next paragraph compares these times to those of sync-schedules.) For `Apache`, we show the throughput (TPUT) and response time (RESP). All numbers reported were averaged over 500 runs. PEREGRINE has relatively high overhead on `water-nsquared` (22.6%) and `cholesky` (46.6%) because these programs do a large number of mutex operations within tight loops. Still, this overhead is lower than the reported 1.2X-6X overhead of a mem-schedule DMT system [9]. Moreover, PEREGRINE speeds up `barnes`, `lu-non-contig`, `radix`, `water-spatial`, and `ocean` (by up to 68.7%) because it safely skips synchronization and sleep operations (§6.4). For the other programs, PEREGRINE's overhead or speedup is within 15%. (Note that increasing the page or file sizes of the workload tends to reduce PEREGRINE's relative overhead because the network and disk latencies dwarf PEREGRINE's.)

For comparison, Figure 11 shows the normalized execution time when enforcing just the sync-schedules. This overhead is comparable to our previous work [19]. For all programs except `water-nsquared`, the overhead of enforcing hybrid schedules is only slightly larger (at most 5.4%) than that of enforcing sync-schedules. This slight increase comes from two sources: (1) PEREGRINE has to enforce execution order constraints to resolve races deterministically for `PBZip2`, `barnes`, `fft`, and `lu-non-contig`; and (2) the instrumentation framework PEREGRINE uses also incurs overhead (§3.2). The overhead for `water-nsquared` increases by 13.4% because it calls functions more frequently than the other benchmarks, and our instrumentation framework inserts code at each function entry and return (§3.2).

Figure 12 shows the speedup of flag relay (§3.2) and skipping blocking operations (§6.4). Besides `water-nsquared` and `cholesky`, a second group of programs, including `barnes`, `lu-non-contig`, `radix`, `water-spatial`, and `ocean`, also perform many synchronization operations, so flag relay speeds up both groups of programs significantly. Moreover, among the synchronization operations

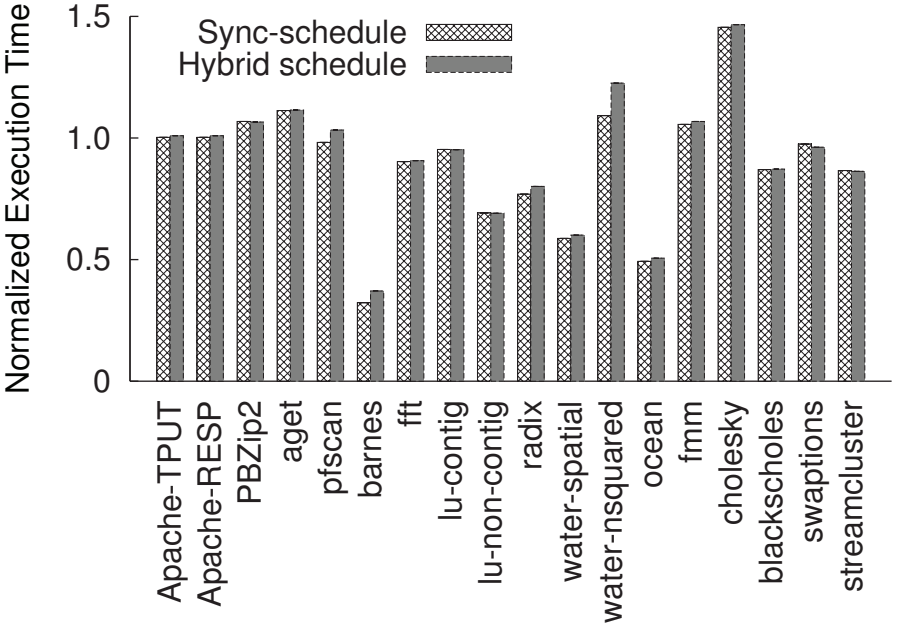

Figure 11: *Normalized execution time when reusing sync-schedules v.s. hybrid schedules.* A time value greater than 1 indicates a slowdown compared to a nondeterministic execution without PEREGRINE. We did not include `racey` because it was not designed for performance benchmarking.

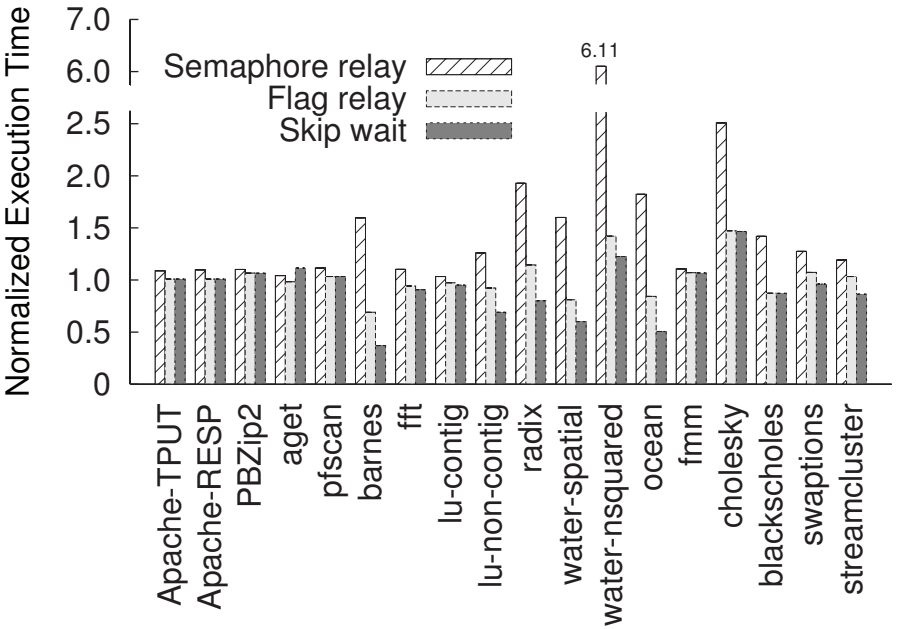

Figure 12: *Speedup of optimization techniques.* Note that Y axis is broken.

Program	Trace	Det	Sli	Sim	Sym
Apache	449	0.4	885.32	n/a	5.8
PBZip2	2,227	0.1	587.9	317.8	19.7
aget	233	0.4	78.8	60.1	13.2
pfscan	46,602	1.1	1,601.4	2,047.9	1,136.6
barnes	324	0.2	300.5	481.5	56.9
fft	39	0.0	2.1	3,661.7	0.4
lu-contig	44,799	19.9	1,271.5	124.9	1,126.7
lu-non-contig	41,302	21.2	1,999.8	14,243.8	1,201.0
radix	3,110	1.5	46.2	96.4	182.9
water-spatial	7,508	1.0	1,407.0	9,628.1	120.6
water-nsquared	12,381	1.7	962.3	1,841.4	215.7
ocean	55,247	26.4	2,259.3	5,902.8	2,062.1
fmm	13,772	8.3	260.5	1,107.5	151.3
cholesky	47,200	28.8	3,102.9	6,350.1	685.5
blackscholes	62,024	16.5	539.9	542.9	3,284.8
swaptions	1,366	0.0	23.2	87.3	1.2
streamcluster	259	0.1	1.4	1.9	4.9

Table 4: *Analysis time.* **Trace** shows the number of thousand LLVM instructions in the execution trace of the evaluated programs, the main factor affecting the execution time of PEREGRINE's various analysis techniques, including race detection (**Det**), slicing (**Sli**), simplification and alias analysis (**Sim**), and symbolic execution (**Sym**). The execution time is measured in seconds. The `Apache` trace is collected from one window of eight requests. `Apache` uses thread pooling which our simplification technique currently does not handle well (§6.3); nonetheless, slicing without simplification works reasonably well for `Apache` already (§7.3).

done by the second group of programs, many are `pthread_barrier_wait()` operations, so PEREGRINE further speeds up these programs by skipping these wait operations.

Analyzer and recorder overhead. Table 4 shows the execution time of PEREGRINE's various program analyses. The execution time largely depends on the size of the execution trace. All analyses typically finish within a few hours. For `PBZip2` and `fft`, we used small workloads (compressing 1 KB file and transforming a 256X256 matrix) to reduce analysis time and to illustrate that the schedules learned from small workloads can be efficiently reused on large workloads. The simplification and alias analysis time of `fft` is large compared to its slicing time because it performs many multiplications on array indexes, slowing down our range analysis. Although `lu-non-contig` and `lu-contig` implement the same scientific algorithm, their data access patterns are very different (§7.3), causing PEREGRINE to spend more time analyzing `lu-non-contig` than `lu-contig`.

As discussed in §6.1, PEREGRINE currently runs KLEE to record executions. Column Sym is also the overhead of PEREGRINE's recorder. This crude, unoptimized recorder can incur large slowdown compared to the normal execution of a program. However, this slowdown can be reduced to around 10X using existing record-replay techniques [13, 33]. Indeed, we have experimented with a

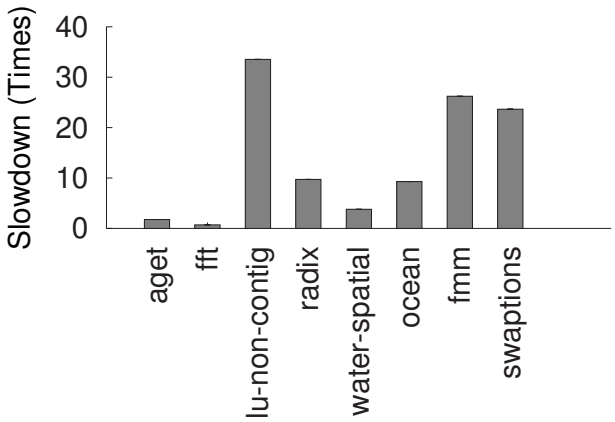

Figure 13: *Overhead of recording* `load` *and* `store` *instructions.*

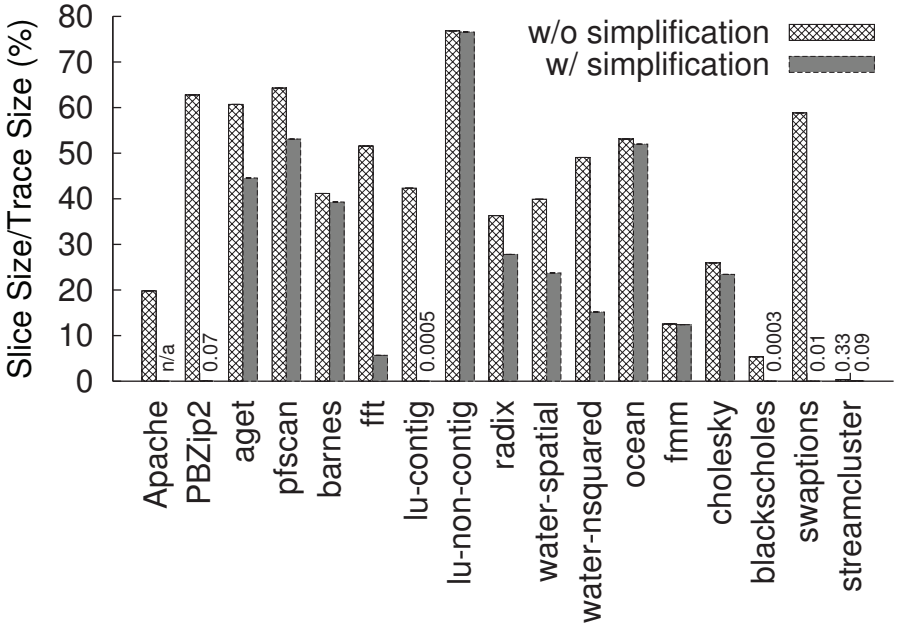

Figure 14: *Slicing ratio after applying determinism-preserving slicing alone (§4) and after further applying schedule-guided simplification (§5).*

preliminary version of a new recorder that records an execution by instrumenting `load` and `store` instructions and saving them into per-thread logs [13]. Figure 13 shows that this new recorder incurs roughly 2-35X slowdown on eight programs, comparable to existing record-replay systems. Due to time constraints, we have not integrated this new recorder with PEREGRINE.

7.3 Stability

Stability measures how frequently PEREGRINE can reuse schedules. The more frequently PEREGRINE reuses schedules, the more efficient it is, and the more repeatable a program running on top of PEREGRINE becomes. While PEREGRINE achieves determinism and efficiency through hybrid schedules, it may have to pay the cost of slightly reduced reuse rates compared to a manual approach [19].

A key factor determining PEREGRINE's schedule-reuse rates is how effectively it can slice out irrelevant instructions from the execution traces. Figure 14 shows the ratio of the slice size over the trace size for PEREGRINE's determinism-preserving slicing technique, with and without schedule-guided simplification. The slicing technique alone reduces the trace size by over 50% for all programs except `PBZip2`, `aget`, `pfscan`, `fft`, `lu-non-contig`, `ocean`, and `swaptions`. The slicing technique combined with scheduled-guide simplification vastly reduces the trace size for `PBZip2`, `aget`, `fft`, `lu-contig`, and `swaptions`.

Recall that PEREGRINE computes the preconditions of a schedule from the input-dependent branches in a trace slice. The fewer branches included in the slice, the more general the preconditions PEREGRINE computes tend to be. We further measured the number of such branches in the trace slices. Table 5 shows the results, together with a upper bound determined by the total number of input-dependent branches in the execution trace, and a lower bound determined by only including branches required to reach the recorded synchronization operations. This lower bound may not be tight as we ignored data dependency. For `barnes`, `fft`, `blackscholes`, `swaptions`, and `streamcluster`, slicing with simplification (Column "Slicing+Sim") achieves the best possible reduction. For `PBZip2`, `aget`, `pfscan`, and `lu-contig`, the number of input-dependent branches in the trace slice is close to the lower bound. In the remaining programs, `Apache`, `fmm`, and `cholesky`

Program	UB	PEREGRINE		LB
		Slicing	Slicing+Sim	
Apache	4,522	624	n/a	56
PBZip2	913	865	101	94
aget	20,826	18,859	9,514	9,491
pfscan	1,062,047	992,524	992,520	992,501
barnes	92	52	52	52
fft	2,266	1,568	17	17
lu-contig	2,823,379	2,337,431	131	128
lu-non-contig	2,962,621	2,877,877	2,876,364	128
radix	175,679	98,750	89,732	75
water-spatial	98,054	77,567	76,763	233
water-nsquared	89,348	76,786	76,242	1,843
ocean	2,605,185	2,364,538	2,361,256	400
fmm	299,816	57,670	56,532	1,642
cholesky	7,459	1,627	1,627	1,233
blackscholes	421,909	409,618	10	10
swaptions	35,584	35,005	21	21
streamcluster	20,851	75	42	42

Table 5: *Effectiveness of program analysis techniques.* **UB** shows the total number of input-dependent branches in the corresponding execution trace, an upper bound on the number included in the trace slice. **Slicing** and **Slicing+Sim** show the number of input-dependent branches in the slice after applying determinism-preserving slicing alone (§4) and after further applying schedule-guided simplification (§5). **LB** shows a lower bound on the number of input-dependent branches, determined by only including branches required to reach the recorded synchronization operations. This lower bound may not be tight as we ignored data dependency when computing it.

also enjoy large reduction, while the other five programs do not. This table also shows that schedule-guided simplification is key to reduce the number of input-dependent branches for `PBZip2`, `fft`, `lu-contig`, `blackscholes`, and `swaptions`, and to reach the lower bound for `blackscholes`, `swaptions`, and `streamcluster`.

We manually examined the preconditions PEREGRINE computed from the input-dependent branches for these programs. We category these programs below.

Best case: `PBZip2`, `fft`, `lu-contig`, `blackscholes`, `swaptions`, and `streamcluster`. PEREGRINE computes the weakest (*i.e.*, most relaxed) preconditions for these programs. The preconditions often allow PEREGRINE to reuse one or two schedules for each number of threads, putting no or few constraints on the data processed. Schedule-guided simplification is crucial for these programs; without simplification, the preconditions would fix the data size and contents.

Slicing limitation: `Apache` and `aget`. The preconditions PEREGRINE computes for `Apache` fix the URL length; they also constrain the page size to be within an 8 KB-aligned range if the page is not cached. The preconditions PEREGRINE computes for `aget` fix the positions of "/" in the URL and narrow down the file size to be within an 8 KB-aligned range. These preconditions thus unnecessarily reduce the schedule-reuse rates. Nonetheless, they can still match many different inputs, because they do not constrain the page or file contents.

Symbolic execution limitation: `barnes`. `barnes` reads in two floating point numbers from a file, and their values affect schedules. Since PEREGRINE cannot symbolically execute floating point instructions, it currently does not collect preconditions from them.

Alias limitation: `lu-non-contig`, `radix`, `water-spatial`, `water-nsquared`, `ocean`, and `cholesky`. Even with simplification, PEREGRINE's alias analysis sometimes reports may-alias for pointers accessed in different threads, causing PEREGRINE to include more instructions than necessary in the slices and compute preconditions that fix the input data. For instance, each thread in `lu-non-contig` accesses disjoint regions in a global array, but the accesses from one thread are *not* continuous, confusing PEREGRINE's alias analysis. (In contrast, each thread in `lu-contig` accesses a contiguous array partition.)

Programs that rarely reuse schedules: `pfscan` and `fmm`. For instance, `pfscan` searches a keyword in a set of files using multiple threads, and for each match, it grabs a lock to increment a counter. A schedule computed on one set of files is unlikely to suit another.

Program	LOC	PEREGRINE	TERN
Apache	464 K	24	6
PBZip2	7,371	1	3
aget	834	0	n/a
pfscan	776	0	n/a
barnes	1,954	0	9
fft	1,403	1	4
lu-contig	991	0	n/a
lu-non-contig	1,265	0	3
radix	661	0	4
water-spatial	1,573	0	9
water-nsquared	1,188	0	10
ocean	6,494	0	5
fmm	3,208	0	9
cholesky	3,683	0	4
blackscholes	1,275	0	n/a
swaptions	1,110	0	n/a
streamcluster	1,963	0	n/a
racey	124	0	n/a

Table 6: *Source annotation requirements of* PEREGRINE *v.s.* TERN. **PEREGRINE** represents the number of annotations added for PEREGRINE, and **TERN** counts annotations added for TERN. Programs not included in the TERN evaluation are labeled n/a. LOC of PBZip2 also includes the lines of code of the compression library libbz2.

7.4 Ease of Use

Table 6 shows the annotations (§6.5) we added to make the evaluated programs work with PEREGRINE. For most programs, PEREGRINE works out of the box. Apache uses its own library functions for common tasks such as memory allocation, so we annotated 21 such functions. We added two annotations to mark the boundaries of client request processing and one to expose the hidden state in Apache (§6.3). PBZip2 decompression uses a custom search function (memstr) to scan through the input file for block boundaries. We added one annotation for this function to relax the preconditions PEREGRINE computes. (PEREGRINE works automatically with PBZip2 compression.) We added one assertion to annotate the range of a variable in fft (§6.5).

For comparison, Table 6 also shows the annotation overhead of our previous DMT system TERN [19]. For all programs except Apache, PEREGRINE has fewer number of annotations than TERN. Although the number of annotations that TERN has is also small, adding these annotations may require developers to manually reconstruct the control- and data-dependencies between instructions.

In order to make the evaluated programs work with PEREGRINE, we had to fix several bugs in them. For aget, we fixed an off-by-one write in revstr() which prevented us from tracking constraints for the problematic write, and a missing check on the return value of pwrite() which prevented us from computing precise ranges. We fixed similar missing checks in swaptions, streamcluster, and radix. We did not count these modifications in Table 6 because they are real bug fixes. (This interesting side-effect illustrates the potential of PEREGRINE as an error detection tool: the precision gained from simplification enables PEREGRINE to detect real races in well-studied programs.)

8 Related Work

Deterministic execution. By reusing schedules, PEREGRINE mitigates input nondeterminism and makes program behaviors repeatable across inputs. This method is based on the *schedule-memoization* idea in our previous work TERN [19], but PEREGRINE largely eliminates manual annotations, and provides stronger determinism guarantees than TERN. To our knowledge, no other DMT systems mitigate input nondeterminism; some actually aggravate it, potentially creating "input-heisenbugs."

PEREGRINE and other DMT systems can be complementary: PEREGRINE can use an existing DMT algorithm when it runs a program on a new input so that it may compute the same schedules at different sites; existing DMT systems can speed up their pathological cases using the schedule-relaxation idea.

Determinator [7] advocates a new, radical programming model that converts all races, including races on memory and other shared resources, into exceptions, to achieve pervasive determinism. This programming model is not designed to be backward-compatible. dOS [10] provides similar pervasive determinism with backward compatibility, using a DMT algorithm first proposed in [20] to enforce mem-schedules. While PEREGRINE currently focuses on multithreaded programs, the ideas in PEREGRINE can be applied to other shared resources to provide pervasive determinism. PEREGRINE's hybrid schedule idea may help reduce dOS's overhead. Grace [12] makes multithreaded programs with fork-join parallelism behave like sequential programs. It detects memory access conflicts efficiently using hardware page protection. Unlike Grace, PEREGRINE aims to make general multithreaded programs, not just fork-join programs, repeatable.

Concurrent to our work, DTHREADS [36] is another efficient DMT system. It tracks memory modifications using hardware page protection and provides a protocol to deterministically commit these modifications. In contrast to DTHREADS, PEREGRINE is software-only and does not rely on page protection hardware which may be expensive and suffer from false sharing; PEREGRINE records and reuses schedules, thus it can handle programs with ad hoc synchronizations [54] and make program behaviors stable.

Program analysis. Program slicing [49] is a general technique to prune irrelevant statements from a program or trace. Recently, systems researchers have leveraged or invented slicing techniques to block malicious input [18], synthesize executions for better error diagnosis [57], infer source code paths from log messages for postmortem analysis [56], and identify critical inter-thread reads that may lead to concurrency errors [59]. Our determinism-preserving slicing technique produces a correct trace slice for multithreaded programs and supports multiple ordered targets. It thus has the potential to benefit existing systems that use slicing.

Our schedule-guided simplification technique shares similarity with SherLog [56] such as the removal of branches contradicting a schedule. However, SherLog starts from log messages and tries to compute an execution trace, whereas PEREGRINE starts with a schedule and an execution trace and computes a simplified yet runnable program. PEREGRINE can thus transparently improve the precision of many existing analyses: simply run them on the simplified program.

Replay and re-execution. Deterministic replay [5, 21, 22, 26, 27, 32, 33, 41, 44, 48, 50] aims to replay the exact recorded executions, whereas PEREGRINE "replays" schedules on different inputs. Some recent deterministic replay systems include Scribe, which tracks page ownership to enforce deterministic memory access [33]; Capo, which defines a novel software-hardware interface and a set of abstractions for efficient replay [41]; PRES and ODR, which systematically search for a complete execution based on a partial one [5, 44]; SMP-ReVirt, which uses page protection for recording the order of conflicting memory accesses [22]; and Respec [35], which uses online replay to keep multiple replicas of a multithreaded program in sync. Several systems [35, 44] share the same insight as PEREGRINE: although many programs have races, these races tend to occur infrequently.

PEREGRINE can help these systems reduce CPU, disk, or network bandwidth overhead, because for inputs that hit PEREGRINE's schedule cache, these systems do not have to record a schedule.

Retro [30] shares some similarity with PEREGRINE because it also supports "mutated" replay. When repairing a compromised system, Retro can replay legal actions while removing malicious ones using a novel dependency graph and *predicates* to detect when changes to an object need not be propagated further. PEREGRINE's determinism-preserving slicing algorithm may be used to automatically compute these predicates, so that Retro does not have to rely on programmer annotations.

Concurrency errors. The complexity in developing multithreaded programs has led to many concurrency errors [39]. Much work exists on concurrency error detection, diagnosis, and correction (*e.g.*, [23–25, 38, 43, 55, 58, 59]). PEREGRINE aims to make the executions of multithreaded programs repeatable, and is complementary to existing work on concurrency errors. PEREGRINE may use existing work to detect and fix the errors in the schedules it computes. Even for programs free of concurrency errors, PEREGRINE still provides value by making their behaviors repeatable.

9 Conclusion and Future Work

PEREGRINE is one of the first efficient and fully deterministic multithreading systems. Leveraging the insight that races are rare, PEREGRINE combines sync-schedules and mem-schedules into hy-

brid schedules, getting the benefits of both. PEREGRINE reuses schedules across different inputs, amortizing the cost of computing hybrid schedules and making program behaviors repeatable across inputs. It further improves efficiency using two new techniques: determinism-preserving slicing to generalize a schedule to more inputs while preserving determinism, and schedule-guided simplification to precisely analyze a program according to a dynamic schedule. Our evaluation on a diverse set of programs shows that PEREGRINE is both deterministic and efficient, and can frequently reuse schedules for half of the evaluated programs.

PEREGRINE's system and ideas have broad applications. Our immediate future work is to build applications on top of PEREGRINE, such as fast deterministic replay, replication, and diversification systems. We will also extend our approach to system-wide deterministic execution by computing inter-process communication schedules and preconditions. PEREGRINE enables precise program analysis according to a set of inputs and dynamic schedules. We will leverage this capability to accurately detect concurrency errors and verify concurrency-error-freedom for real programs.

Acknowledgement

We thank Cristian Cadar, Bryan Ford (our shepherd), Ying Xu, and the anonymous reviewers for their many helpful comments, which have substantially improved the content and presentation of this paper. We thank Dawson Engler, Yang Tang, and Gang Hu for proofreading. This work was supported in part by AFRL FA8650-10-C-7024 and FA8750-10-2-0253, and NSF grants CNS-1117805, CNS-1054906 (CAREER), CNS-1012633, and CNS-0905246.

References

[1] ab - Apache HTTP server benchmarking tool. `http://httpd.apache.org/docs/2.2/programs/ab.html`.

[2] The LLVM Compiler Framework. `http://llvm.org`.

[3] Parallel BZIP2 (PBZIP2). `http://compression.ca/pbzip2/`.

[4] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley, 2006.

[5] G. Altekar and I. Stoica. ODR: output-deterministic replay for multicore debugging. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, pages 193–206, Oct. 2009.

[6] Apache Web Server. `http://www.apache.org`.

[7] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient system-enforced deterministic parallelism. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI '10)*, Oct. 2010.

[8] D. Avots, M. Dalton, V. B. Livshits, and M. S. Lam. Improving software security with a C pointer analysis. In *Proceedings of the 27th International Conference on Software Engineering (ICSE '05)*, pages 332–341, May 2005.

[9] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. CoreDet: a compiler and runtime system for deterministic multithreaded execution. In *Fifteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '10)*, pages 53–64, Mar. 2010.

[10] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble. Deterministic process groups in dOS. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI '10)*, pages 1–16, Oct. 2010.

[11] T. Bergan, J. Devietti, N. Hunt, and L. Ceze. The deterministic execution hammer: how well does it actually pound nails? In *The 2nd Workshop on Determinism and Correctness in Parallel Programming (WODET '11)*, Mar. 2011.

[12] E. Berger, T. Yang, T. Liu, D. Krishnan, and A. Novark. Grace: safe and efficient concurrent programming. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '09)*, pages 81–96, Oct. 2009.

[13] S. Bhansali, W.-K. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinić, D. Mihocka, and J. Chau. Framework for instruction-level tracing and analysis of program executions. In *Proceedings of the 2nd International Conference on Virtual Execution Environments (VEE '06)*, pages 154–163, June 2006.

[14] P. Boonstoppel, C. Cadar, and D. Engler. RWset: attacking path explosion in constraint-based test generation. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, pages 351–366, Mar. 2008.

[15] C. Cadar, D. Dunbar, and D. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the Eighth Symposium on Operating Systems Design and*

Implementation (OSDI '08), pages 209–224, Dec. 2008.

[16] M. Castro, M. Costa, and J.-P. Martin. Better bug reporting with better privacy. In *Thirteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '08)*, pages 319–328, Mar. 2008.

[17] P. Collingbourne, C. Cadar, and P. H. Kelly. Symbolic crosschecking of floating-point and SIMD code. In *Proceedings of the 6th ACM European Conference on Computer Systems (EUROSYS '11)*, pages 315–328, Apr. 2011.

[18] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado. Bouncer: securing software by blocking bad input. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP '07)*, pages 117–130, Oct. 2007.

[19] H. Cui, J. Wu, C.-C. Tsai, and J. Yang. Stable deterministic multithreading through schedule memoization. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI '10)*, Oct. 2010.

[20] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: deterministic shared memory multiprocessing. In *Fourteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '09)*, pages 85–96, Mar. 2009.

[21] G. Dunlap, S. T. King, S. Cinar, M. Basrat, and P. Chen. ReVirt: enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI '02)*, pages 211–224, Dec. 2002.

[22] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen. Execution replay of multiprocessor virtual machines. In *Proceedings of the 4th International Conference on Virtual Execution Environments (VEE '08)*, pages 121–130, Mar. 2008.

[23] D. Engler and K. Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 237–252, Oct. 2003.

[24] P. Fonseca, C. Li, and R. Rodrigues. Finding complex concurrency bugs in large multi-threaded applications. In *Proceedings of the 6th ACM European Conference on Computer Systems (EUROSYS '11)*, pages 215–228, Apr. 2011.

[25] Q. Gao, W. Zhang, Z. Chen, M. Zheng, and F. Qin. 2ndStrike: towards manifesting hidden concurrency typestate bugs. In *Sixteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '11)*, pages 239–250, Mar. 2011.

[26] D. Geels, G. Altekar, P. Maniatis, T. Roscoe, and I. Stoica. Friday: global comprehension for distributed replay. In *Proceedings of the Fourth Symposium on Networked Systems Design and Implementation (NSDI '07)*, Apr. 2007.

[27] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang. R2: An application-level kernel for record and replay. In *Proceedings of the Eighth Symposium on Operating Systems Design and Implementation (OSDI '08)*, pages 193–208, Dec. 2008.

[28] B. Hackett and A. Aiken. How is aliasing used in systems software? In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '06/FSE-14)*, pages 69–80, Nov. 2006.

[29] M. D. Hill and M. Xu. Racey: A stress test for deterministic execution. `http://www.cs.wisc.edu/~markhill/racey.html`.

[30] T. Kim, X. Wang, N. Zeldovich, and M. F. Kaashoek. Intrusion recovery using selective re-execution. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI '10)*, pages 1–9, Oct. 2010.

[31] J. C. King. A new approach to program testing. In *Proceedings of the international conference on Reliable software*, pages 228–233, 1975.

[32] R. Konuru, H. Srinivasan, and J.-D. Choi. Deterministic replay of distributed Java applications. In *Proceedings of the 14th International Symposium on Parallel and Distributed Processing (IPDPS '00)*, pages 219–228, May 2000.

[33] O. Laadan, N. Viennot, and J. Nieh. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '10)*, pages 155–166, June 2010.

[34] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. ACM*, 21(7):558–565, 1978.

[35] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. M. Chen, and J. Flinn. Respec: efficient online multiprocessor replayvia speculation and external determinism. In *Fifteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '10)*, pages 77–90, Mar. 2010.

[36] T. Liu, C. Curtsinger, and E. D. Berger. DTHREADS: efficient deterministic multithreading. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, Oct. 2011.

[37] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: detecting atomicity violations via access interleaving invariants. In *Twelfth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '06)*, pages 37–48, Oct. 2006.

[38] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. Muvi: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. *SIGOPS Oper. Syst. Rev.*, 41(6):103–116, 2007.

[39] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Thirteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '08)*, pages 329–339, Mar. 2008.

[40] F. Mattern. Virtual time and global states of distributed systems. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226. 1988.

[41] P. Montesinos, M. Hicks, S. T. King, and J. Torrellas. Capo: a software-hardware interface for practical deterministic multiprocessor replay. In *Fourteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '09)*, pages 73–84, Mar. 2009.

[42] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: efficient deterministic multithreading in software. In *Fourteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '09)*, pages 97–108, Mar. 2009.

[43] S. Park, S. Lu, and Y. Zhou. CTrigger: exposing atomicity violation bugs from their hiding places. In *Fourteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '09)*, pages 25–36, Mar. 2009.

[44] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. PRES: probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, pages 177–192, Oct. 2009.

[45] M. Ronsse and K. De Bosschere. Recplay: a fully integrated practical record/replay system. *ACM Trans. Comput. Syst.*, 17(2):133–152, 1999.

[46] R. Rugina and M. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI '00)*, pages 182–195, June 2000.

[47] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multithreaded programming. *ACM Trans. Comput. Syst.*, pages 391–411, Nov. 1997.

[48] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *Proceedings of the USENIX Annual Technical Conference (USENIX '04)*, pages 29–44, June 2004.

[49] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages 3(3)*, pages 121–189, 1995.

[50] VMWare Virtual Lab Automation. `http://www.vmware.com/solutions/vla/`.

[51] J. Whaley. bddbddb Project. http://bddbddb.sourceforge.net. URL `http://bddbddb.sourceforge.net`.

[52] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI '04)*, pages 131–144, June 2004.

[53] J. Wu, H. Cui, and J. Yang. Bypassing races in live applications with execution filters. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI '10)*, Oct. 2010.

[54] W. Xiong, S. Park, J. Zhang, Y. Zhou, and Z. Ma. Ad hoc synchronization considered harmful. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI '10)*, Oct. 2010.

[55] Y. Yu, T. Rodeheffer, and W. Chen. RaceTrack: efficient detection of data race conditions via adaptive tracking. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 221–234, Oct. 2005.

[56] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy. SherLog: error diagnosis by connecting clues from run-time logs. In *Fifteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '10)*, pages 143–154, Mar. 2010.

[57] C. Zamfir and G. Candea. Execution synthesis: a technique for automated software debugging. In *Proceedings of the 5th ACM European Conference on Computer Systems (EUROSYS '10)*, pages 321–334, Apr. 2010.

[58] W. Zhang, C. Sun, and S. Lu. ConMem: detecting severe concurrency bugs through an effect-oriented approach. In *Fifteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '10)*, pages 179–192, Mar. 2010.

[59] W. Zhang, J. Lim, R. Olichandran, J. Scherpelz, G. Jin, S. Lu, and T. Reps. ConSeq: detecting concurrency bugs through sequential errors. In *Sixteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '11)*, pages 251–264, Mar. 2011.