Determinism Is Not Enough: Making Parallel Programs Reliable with Stable Multithreading

Junfeng Yang, Heming Cui, Jingyue Wu, Yang Tang, Gang Hu
Columbia University

ABSTRACT

Our accelerating computational demand and the rise of multicore hardware have made parallel programs, especially shared-memory multithreaded programs, increasingly pervasive and critical. Yet, these programs remain extremely difficult to write, test, analyze, debug, and verify. Conventional wisdom has attributed these difficulties to nondeterminism (i.e., repeated executions of the same program on the same input may show different behaviors), and researchers have recently dedicated much effort to bringing determinism into multithreading. In this article, we argue that determinism is not as useful as commonly perceived: it is neither sufficient nor necessary for reliability. We present our view on why multithreaded programs are difficult to get right, describe a promising approach we call stable multithreading to dramatically improve reliability, and summarize our last four years' research on building and applying stable multithreading systems.

1 Introduction

Reliable software has long been the dream of many researchers, practitioners, and users. In the last decade or so, several research and engineering breakthroughs have greatly improved the reliability of sequential programs (or the sequential aspect of parallel programs). Successful examples include Coverity's source code analyzer [6], Microsoft's Static Driver Verifier [3], Valgrind memory checker [17], and certified operating systems and compilers [20].

However, the same level of success has not yet propagated to parallel programs. These programs are notoriously difficult to write, test, analyze, debug, and verify, much harder than the sequential versions. Experts consider reliable parallelism "something of a black art" [8] and one of the grand challenges in computing [1, 18]. Unsurprisingly, widespread parallel programs are plagued with insidious concurrency bugs [15], such as data races (concurrent accesses to the same memory location with at least one write) and deadlocks (threads circularly waiting for resources). Some of the worst of these bugs have killed people in the Therac 25 incidents and caused the 2003 Northeast blackout. Our study also reveals that these bugs may be exploited by attackers to violate confidentiality, integrity, and availability of critical systems [24].

In recent years, two technology trends have made the challenge of reliable parallelism more urgent. The first is the rise of multicore hardware. The speed of a single processor core is limited by fundamental physical constraints, forcing processors into multicore designs. Thus, developers must resort to parallel code for best performance on multicore processors. The second is our accelerating computational demand. Scientific computing, video and image processing, financial simulation, "big data" analytics, web search, and online social networking are all massive computations and employ various kinds of parallel programs for performance.

If reliable software is an overarching challenge of computer science, reliable parallelism is surely the keystone. To make parallel programs reliable, researchers have devoted decades of effort, producing numerous ideas and systems, ranging from new hardware, programming languages, programming models, to tools that detect, diagnose, avoid, or fix concurrency bugs. As usual, new hardware, languages, or models take years, if not forever, to adopt. Tools are helpful, but they tend to attack derived problems, not the root cause.

Over the past four years, we have been attacking fundamental, open problems in making shared-memory multithreaded programs reliable. These programs express concurrency using threads, essentially lightweight, sequential processes that share memory. We target these programs because they are the most widespread type of parallel programs with mature support from hardware, operating systems, libraries, and programming languages. They will likely remain prevalent in the foreseeable future.

Unlike sequential programs, repeated executions of the same multithreaded program on the same input may yield different (e.g., correct vs. buggy) behaviors, depending on how the threads interleave. Conventional wisdom has long blamed this *nondeterminism* for the challenges in reliable multithreading [13]: threads are nondeterministic by default, and it is the (tricky) job of developers to account for this nondeterminism. Nondeterminism has direct implications on reliability. For instance, it makes testing less effective: a program may run correctly on an input in the testing lab because the interleavings tested happen to be correct, but executions on the same exact input may still fail in the field when the program hits a buggy, untested interleaving.

To eliminate this nondeterminism, several groups of researchers including us have dedicated much effort to building deterministic multithreading (DMT) systems [2, 4, 5, 7, 12, 14, 19]. These systems force a multithreaded program to always execute the same thread interleaving, or *schedule*, on the same input, thus always resulting in the same behavior. By mapping each input to only one schedule, DMT brings determinism, a key property of sequential computing, into multithreading.

However, we argue that nondeterminism is responsible for only a small piece of the puzzle and that determinism, the cure to nondeterminism, is not as useful as commonly perceived: it is neither sufficient nor necessary for reliability. It is not sufficient because a perfectly deterministic system can map each input to an arbitrary schedule, so that small input perturbations lead to vastly different schedules, artificially reducing the program's robustness and stability. It is not necessary because a nondeterministic system with a small set of schedules for all inputs can be made reliable by exhaustively checking all schedules. (See §2 for more discussion.)

We believe what makes multithreading hard is rather quantitative: multithreaded programs have *too many* schedules. The number of schedules for each input is already enormous because the parallel threads may interleave in many ways, depending on such factors as hardware timing and operating system scheduling. Aggregated over all inputs, the number is even greater. Finding a few schedules that trigger concurrency errors out of all enormously many schedules (so developers can prevent them) is like finding needles in a haystack. Although DMT reduces schedules for each input, it may map each input to a different schedule, so the total set of schedules for all inputs remains enormous.

We attacked this root cause by asking: are *all* the enormously many schedules necessary? Our study reveals that *many real-world programs can use a small set of schedules to efficiently process a wide range of inputs* [10]. Leveraging this insight, we envision a new approach we call *stable multithreading (StableMT)* that reuses each schedule on a wide range of inputs, mapping all inputs to a dramatically reduced set of schedules. By vastly shrinking the haystack, it makes the needles much easier to find. By mapping many inputs to the same schedule, it stabilizes program behaviors against small input perturbations. StableMT and DMT are not mutually exclusive: a system can be both deterministic and stable.

To realize our vision of StableMT, we have built three systems: TERN [10] and PEREGRINE [11], two compiler and runtime implementations of StableMT; and a program analysis framework that leverages StableMT to achieve high coverage and precision unmatched by its counterparts [22]. These systems address three complementary key challenges, two of which have been long open in related areas. Specifically, TERN addresses the challenge of *how to compute highly reusable schedules*. The more reusable the schedules, the fewer of them are needed. Unfortunately, computing reusable schedules is undecidable at compile time and costly at runtime. PEREGRINE addresses *how to efficiently make executions follow schedules and not deviate,* a decades-long challenge in the area of deterministic execution and replay. Our analysis framework addresses *how to effectively analyze multithreaded programs*, a well-known open problem in program analysis. Our implementations of these systems are mostly transparent to developers and fully compatible with existing hardware, operating systems, thread libraries, and programming languages, simplifying adoption.

Our initial results are promising. Evaluation on a diverse set of widespread multithreaded programs, including the `Apache` web server and the `MySQL` database, show that TERN and PEREGRINE dramatically reduce schedules. For instance, under typical setups, they reduce the number of schedules needed by parallel compression utility `PBZip2` down to *two* schedules for each different number of threads, regardless of the file contents. Their overhead is moderate, less than 15% for most programs. Our program analysis framework enables the construction of many program analyses with precision and coverage unmatched by their counterparts. For instance, a race detector we built found previously unknown bugs in extensively checked code with almost no false bug reports.

In the rest of this article, we first present our view on why multithreaded programs are hard to get right. We then describe our StableMT approach, its benefits, and the three StableMT systems we built. We finally present some results and conclude.

2 Why Are Multithreaded Programs So Hard to Get Right?

We start with preliminaries, then describe the challenges caused by nondeterminism and by too many schedules. We then explain why nondeterminism is a lesser cause than too many schedules.

2.1 Preliminaries: Inputs, Schedules, and Buggy Schedules

To ease discussion, we use *input* to broadly refer to the data a program reads from its execution environment, including not only the data read from files and sockets, but also command line arguments, return values of external functions such as `gettimeofday`, and any external data that can affect program execution. We use *schedule* to broadly refer to the (partially or totally) ordered set of communication operations in a multithreaded execution, including synchronizations (e.g., `lock` and `unlock` operations) and shared memory accesses (e.g., `load` and `store` instructions to shared memory). Of all the schedules, most run fine, but some trigger concurrency errors, causing program crashes, incorrect computations, deadlocked executions, and other failures. Consider the toy program below:

```
// thread 1       // thread 2
lock(l);          lock(l);
*p = ...;         p = NULL;
unlock(l);        unlock(l);
```

The schedule in which thread 2 gets the lock before thread 1 causes a dereference-of-NULL failure. Consider another example. The toy program below has data races on `balance`:

```
// thread 1                 // thread 2
// deposit 100              // withdraw 100
t = balance + 100;
                            balance = balance − 100;
balance = t;
```

The schedule with the statements executed in the order shown corrupts `balance`. We call the schedules that trigger concurrency errors *buggy schedules*. Strictly speaking, the errors are in the programs, triggered by a combination of inputs and schedules. However, typical concurrency errors, such as most errors appeared in previous studies [15, 24], depend much more on the schedules than the inputs (e.g., once the schedule is fixed, the bug occurs for all inputs allowed by the schedule). Thus, recent research on testing multithreaded programs (e.g., [16]) is focused on effectively testing schedules to find the buggy ones.

2.2 Challenges Caused by Nondeterminism

A multithreaded program is nondeterministic because even with the same program and input, different schedules may still lead to different behaviors. For instance, the two toy programs in the previous subsection do not always run into the bugs. Except for the schedules described, the other schedules lead to correct executions.

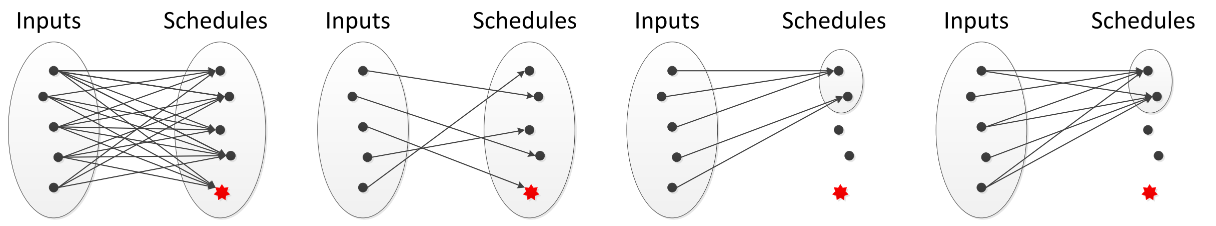
This nondeterminism raises many challenges, especially in testing and debugging. Suppose an input can execute under n schedules. Testing $n-1$ schedules is not enough for complete reliability because the single untested schedule may still be buggy. An execution in the field may hit this untested schedule and fail. Debugging is challenging, too. To reproduce a field failure for diagnosis, the exact input alone is not enough. Developers must also manage to reconstruct the buggy schedule out of n possibilities.

Figure 1a depicts the traditional multithreading approach. Conceptually, it is a many-to-many mapping, where one input may execute under many schedules because of nondeterminism, and many inputs may execute under one schedule because a schedule fixes the order of the communication operations but allows the local computations to operate on any input data.

2.3 Challenges Caused by Too Many Schedules

A typical multithreaded program has an enormous number of schedules. For a single input, the number of schedules is asymptotically exponential in the schedule length. For instance, given m threads each competing for a lock k times, each order of lock acquisitions forms a schedule, easily yielding $\frac{(mk)!}{(k!)^m} \geq (m!)^k$ total schedules—a number exponential in both m and k. Aggregated over all inputs, the number of schedules is even greater.

Finding a few buggy schedules in these exponentially many schedules raises a series of "needle-in-a-haystack" challenges. For instance, to write correct multithreaded programs, developers must carefully synchronize their code to weed out the buggy schedules. As usual, humans err when they must scrutinize many possibilities to locate corner cases. Various forms of testing tools suffer, too. Stress testing is the common method for (indirectly) testing sched-



(a) *Traditional.* (b) *Deterministic.* (c) *Stable (deterministic).* (d) *Stable (nondeterministic).*

Figure 1: Different multithreading approaches. Red stars represent buggy schedules. Traditional multithreading (a) is a conceptual many-to-many mapping where one input may execute under many schedules because of nondeterminism, and many inputs may execute under one schedule because a schedule fixes the order of the communication operations but allows the local computations to operate on any input data. DMT (b) may map each input to an arbitrary schedule, reducing programs' robustness on input perturbations. StableMT (c and d) reduces the total set of schedules for all inputs (represented by the shrunk ellipses), increasing robustness and improving reliability. StableMT and DMT are orthogonal: a StableMT system can be deterministic (c) or nondeterministic (d).

ules, but it often redundantly tests the same schedules while missing others. Recent tools (e.g., [16]) systematically test schedules for bugs, but we seriously lack resources to cover more than a tiny fraction of all exponentially many schedules.

2.4 Determinism Is Not As Useful as Commonly Perceived

To address the challenges raised by nondeterminism, researchers including us have dedicated much effort and built several systems that force a multithreaded program to always run the same schedule on the same input, bringing determinism to multithreading. This determinism does have value for reliability. For instance, one testing execution now validates all future executions on the same input. Reproducing a concurrency error now requires only the input.

In contrast to this effort, little has been done to solve the challenges caused by too many schedules. We believe the community has charged nondeterminism more than its share of the guilt and overlooked the main culprit—a rather quantitative cause that multithreaded programs simply have too many schedules. We argue that, although determinism has value, its value is smaller than commonly perceived. It is neither sufficient nor necessary for reliability.

Determinism $\not\Rightarrow$ reliability. Determinism is a narrow property: same input + same program = same behavior. It has no jurisdiction if the input or program changes however slightly. Yet, we often expect a program to be robust or stable against slight program changes or input perturbations. For instance, adding a debug `printf` should in principle not make the bug disappear. Similarly, a single bit flip of a file should usually not cause a compression utility to crash. Unfortunately, determinism does not provide this stability and, if naïvely implemented, even undermines it.

To illustrate, consider the system depicted in Figure 1b which maps each input to an arbitrary schedule. This mapping is perfectly deterministic, but it destabilizes program behaviors on multiple inputs. A single bit flip may force a program to discard a correct schedule and adventure into a vastly different, buggy schedule.

This instability is counterintuitive at least, and raises new reliability challenges. For instance, testing one input provides little assurance on very similar inputs, despite that the differences in input do not invalidate the tested schedule. Debugging now requires every bit of the bug-inducing input, including not only the data a user typed, but also environment variables, shared libraries, etc. A different user name? Error report doesn't include credit card numbers? The bug may never be reproduced, regardless of how many times developers retry, because the schedule chosen by the deterministic system for the altered input happens to be correct. Note that even a correct sequential program may show very different behaviors for small input changes across boundary conditions, but these conditions are typically infrequent and the different behaviors are intended by developers. In contrast, the instability introduced by the system in Figure 1b is artificial and on all inputs.

Besides inputs, naïvely implemented determinism can destabilize program behaviors on minor code changes, so adding a debug `printf` causes the bug to deterministically disappear. Another problem is that the number of all possible schedules remains enormous, so the coverage of schedule testing tools remains low.

In practice, to mitigate these problems, researchers have to augment determinism with other techniques. To support debug `printf`, some propose to temporarily revert to nondeterministic execution [12]. DMP [12], CoreDet [4], and Kendo [19] change schedules only if the inputs change low-level instructions executed. Although better than mapping each input to an arbitrary schedule, they still allow small input perturbations to destabilize schedules unnecessarily when the perturbations change the low-level instructions executed (e.g., one extra `load` executed), observed in our experiments [10]. Our TERN and PEREGRINE systems and others' DTHREADS [14] built subsequently to TERN combine DMT with StableMT (elaborated next section) to frequently reuse schedules on a wide range of inputs for stability.

Reliability $\not\Rightarrow$ determinism. Determinism is a binary property: if an input maps to $n > 1$ schedules, executions on this input may be nondeterministic, however small n is. Yet, a nondeterministic system with a small set of total schedules can be made reliable easily. Consider an extreme case, the nondeterministic system depicted in Figure 1d which maps all inputs to at most two schedules. In this system, the challenges caused by nondeterminism (§2.2) are easy to solve. For instance, to reproduce a field failure given an input, developers can easily afford to search for one out of only two schedules. To offer an analogy, a coin toss is nondeterministic, but humans have no problem understanding and doing it because there are only two possible outcomes.

3 Shrinking the Haystack with Stable Multithreading

Motivated by the limitations of determinism and the challenges caused by exponentially many schedules, we investigated a central research question: *are all the exponentially many schedules necessary*? A schedule is necessary if it is the only one that can (1) process specific inputs or (2) yield good performance under specific scenarios. Removing unnecessary schedules from the haystack would make the needles easier to find.

We investigated this question on a diverse set of popular multithreaded programs, ranging from server programs such as `Apache`,

Program	Purpose	Constraints on inputs sharing schedules
`Apache`	Web server	For a group of typical HTTP GET requests, same cache status
`PBZip2`	Compression	Same number of threads
`aget`	File download	Same number of threads, similar file sizes
`barnes`	N-body simulation	Same number of threads, same values of two configuration variables
`fft`	Fast Fourier transform	Same number of threads
`lu-contig`	Matrix decomposition	Same number of threads, similar sizes of matrices and blocks
`blackscholes`	Option pricing	Same number of threads, number of options no less than number of threads
`swaptions`	Swaption pricing	Same number of threads, number of swaptions no less than number of threads

Table 1: *Constraints on inputs sharing the same equivalent class of schedules*. For each program, one schedule out of the class suffices to process any input satisfying the constraints in the third column under typical setups (e.g., no system call failures or signals). We describe how to compute such constraints in §4.

to desktop utilities such as parallel compression utility PBZip2, to parallel implementations of computation-intensive algorithms such as fast Fourier transformation. These programs use diverse synchronization primitives such as locks, semaphores, condition variables, and barriers. Our investigation reveals the following two insights.

First, for many programs, a wide range of inputs share the same equivalent class of schedules. Thus, one schedule out of the class suffices to process the entire input range. Intuitively, an input often contains two types of data: (1) metadata that controls the communication of the execution, such as the number of threads to spawn; and (2) computational data that the threads locally compute on. A schedule requires the input metadata to have certain values, but it allows the computational data to vary. That is, it can process any input that has the same metadata. For instance, consider the aforementioned PBZip2 which splits an input file among multiple threads, each compressing one file block. The communication, i.e., which thread gets which file block, is independent of the thread-local compression. Under a typical setup (e.g., no `read` failures or signals), for each different number of threads set by a user, PBZip2 can use two schedules (one if the file can be evenly divided by the number of threads and another otherwise) to compress any file, regardless of the file data.

This loose coupling of inputs and schedules is not unique to PBZip2; many other programs also exhibit this property. Table 1 shows a sample of our findings. The programs shown include three real-world programs, Apache, PBZip2, and aget (a parallel file download utility) and five implementations of computation-intensive algorithms from two widely used benchmark suites, Stanford's SPLASH2 and Princeton's PARSEC. (We describe how to compute the constraints that a schedule places on the inputs in §4.)

Second, the overhead of enforcing a schedule on different inputs is low. Presumably, the exponentially many schedules allow the runtime system to react to various timing factors and select an efficient schedule. However, results from the StableMT systems we built invalidated this presumption. With carefully designed schedule representations (§4.2), our systems incurred less than 15% overhead enforcing schedules on different inputs for most evaluated programs (§6). We believe this moderate overhead is worth the gains in reliability.

Leveraging these insights, we have invented *stable multithreading (StableMT)*, a new multithreading approach that reuses each schedule on a wide range of inputs, mapping all inputs to a dramatically reduced set of schedules. By vastly shrinking the haystack, it addresses all the needle-in-a-haystack challenges at once. In addition, StableMT stabilizes program behaviors on inputs that map to the same schedule and minor program changes that do not affect the schedules, providing robustness and stability anticipated by developers and users.

StableMT and DMT are orthogonal. StableMT aims to reduce the set of schedules for *all* inputs, whereas DMT aims to reduce the schedules for *each* input (down to one). A StableMT system may be either deterministic or nondeterministic. Figure 1c and Figure 1d depict two StableMT systems: the many-to-one mapping in Figure 1c is deterministic, while the many-to-few mapping in Figure 1d is nondeterministic. A many-to-few mapping improves performance because the runtime system can choose an efficient schedule out of a few for an input based on current timing factors, but it increases the efforts and resources needed for reliability. Fortunately, the choices of schedules are only a few (e.g., a small constant such as two), so the challenges caused by nondeterminism are easy to solve.

3.1 Benefits

By vastly reducing the set of schedules, StableMT brings numerous reliability benefits to multithreading. We describe several:

Testing. StableMT automatically increases the coverage of schedule testing tools, with coverage defined as the ratio of tested schedules over all schedules. For instance, consider PBZip2 again which needs only two schedules for each different number of threads under typical setups. Testing 32 schedules effectively covers from 1 to 16 threads. Given that (1) PBZip2 achieves peak performance when the number of threads is identical or close to the number of cores and (2) a typical machine has up to 16 cores, 32 tested schedules can practically cover most schedules executed in the field.

Debugging. Reproducing a bug now does not require the exact input, as long as the original and the altered inputs map to the same schedule. It does not require the exact program either, as long as the changes to the program do not affect the schedule. Users may remove private information such as credit card numbers from their bug reports. Developers may reproduce the bugs in different environments or add `printf` statements.

Analyzing and verifying programs. Static analysis can now focus only on the set of schedules enforced in the field, gaining precision. Dynamic analysis enjoys the same benefits as testing. Model checking can now check drastically fewer schedules, mitigating the so-called "state explosion" problem [9]. Interactive theorem proving becomes easier, too, because verifiers need to prove theorems only on the set of schedules enforced in the field. We will describe these benefits in more detail in §5.

Avoiding errors at runtime. Programs can also adaptively learn correct schedules in the field, then reuse them on future inputs to avoid unknown, potentially buggy schedules. We will describe this benefit in more detail in §4.1.

3.2 Caveats

StableMT is certainly not for every multithreaded program. It works well with programs whose schedules are loosely coupled with inputs, but there are also other programs. For instance, a program may decide to spawn threads or invoke synchronizations based on intricate conditions involving many bits in the input. The parallel `grep`-like utility `pfscan` is an example. It searches for 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. To increase the input range each schedule covers, developers can exclude the operations on this lock from the schedule using annotations.

StableMT provides robustness and stability on small input and program perturbations when they do not affect schedules. However, there is still room to improve. For instance, when developers change their programs by adding synchronizations, it may be more efficient to update previously computed schedules rather than to recompute from scratch. We leave this idea for future work.

4 Building Stable Multithreading Systems

Although the vision of stable multithreading is appealing, realizing it faces numerous challenges. Three main challenges are:

- How can we compute the schedules to map inputs to? The schedules must be feasible so executions reusing them do not get stuck. They should also be highly reusable.
- How can we enforce schedules deterministically and efficiently? "Deterministically" so executions that reuse a schedule cannot deviate even if there are data races, and "efficiently" so overhead does not offset reliability benefits. This challenge is also a decades-long challenge in the area of deterministic execution and replay.
- How can we handle multithreaded server programs? They often run for a long time and react to each client request as it arrives, making their schedules very specific to a stream of requests and difficult to reuse.

Over the past four years, we have been tackling these challenges and building StableMT systems, which resulted in two StableMT prototypes, TERN [10] and PEREGRINE [11], that frequently reuse schedules with low overhead. This section describes our solutions to these challenges. Our solutions are by no means the only ones; subsequent to TERN, others have also built a system that stabilizes schedules for general multithreaded programs [14].

4.1 Computing Schedules

Crucial to implementing StableMT is how to compute the set of schedules for processing inputs. At the bare minimum, a schedule must be feasible when enforced on an input, so the execution does not get stuck or deviate from the schedule. Ideally, the set of schedules should also be small for reliability. One possible idea is to precompute schedules using static source code analysis, but the halting problem makes it undecidable to statically compute schedules guaranteed to work dynamically. Another possibility is to compute schedules on the fly while a program is running, but the computations may be complex and their overhead high.

Instead, we compute schedules by recording them from past executions; the recorded schedules can then be reused on future inputs to stabilize program behaviors. TERN, our system implementing this idea, works as follows. At runtime, it maintains a persistent cache of schedules recorded from past executions. When an input arrives, TERN searches the cache for a schedule compatible with the input. If it finds one, it simply runs the program while enforcing the schedule. Otherwise, it runs the program as is while recording a new schedule from the execution, and saves the schedule into the cache for future reuse.

The TERN approach to computing schedules has several benefits. First, by reusing schedules shown to work, TERN may avoid potential errors in unknown schedules, improving reliability. A real-world analogy is the natural tendencies in humans and animals to follow familiar routes to avoid possible hazards along unknown routes. Migrant birds, for example, often migrate along fixed flyways. Why don't our multithreading systems learn from them and reuse familiar schedules? (The name TERN comes from the Arctic Tern, a bird species that migrates the farthest among all animals.)

```
main(int argc, char *argv[]) {
  int i, nthread = atoi(argv[1]);
  for(i=0; i<nthread; ++i)
    pthread_create(worker); // create worker threads
  for(i=0; i<nthread; ++i)
    worklist.add(read_block(i)); // add block to work list
  // Error: missing pthread_join() operations
  worklist.clear(); // clear work list
  ...
}
worker() { // worker threads for compressing file blocks
  block = worklist.get(); // get a file block from work list
  compress(block);
}
compress(block_t block) {
  if(block.data[0] == block.data[1])
    ...
}
```

Figure 2: *An example program based on parallel compression utility PBZip2.* It spawns `nthread` worker threads, splits a file among the threads, and compresses the file blocks in parallel.

Second, TERN explicitly stores schedules, so developers and users can flexibly choose what schedules to record and when. For instance, developers can populate a cache of correct schedules during testing and then deploy the cache together with their program, improving testing effectiveness and avoiding the overhead to record schedules on user machines. Moreover, they can run their favorite checking tools on the schedules to detect a variety of errors, and choose to keep only the correct schedules in the cache.

Lastly, TERN is efficient because it can amortize the cost of computing schedules. Specifically, recording and checking a schedule is more expensive than reusing a schedule, but, fortunately, TERN does it only once for each schedule and then reuses the schedule on many inputs, amortizing the cost.

A key challenge facing TERN is to check that an input is compatible with a schedule before executing the input under the schedule. Otherwise, if TERN tries to enforce a schedule, for instance, of two threads on an input that requires four, the execution would not follow the schedule. This challenge turns out to be the most difficult one we must solve in building TERN. Our final solution leverages several advanced program analysis techniques, including two new ones we invent. We refer interested readers to our papers [10, 11] for details, and only describe the high level idea here.

When recording a schedule, TERN tracks how the synchronizations in the schedule depend on the input. It captures these dependencies into a relaxed, quickly checkable set of constraints called the *precondition* of the schedule. It then reuses the schedule on all inputs satisfying the precondition, avoiding the runtime cost of recomputing schedules.

A naïve way to compute the precondition is to collect constraints from all input-dependent branches in an execution. For instance, if a branch instruction inspects input variable `X` and goes down the true branch, we add a constraint that `X` must be nonzero to the precondition. A precondition computed this way is sufficient, but it contains many unnecessary constraints concerning only thread-local computations. Since an over-constraining precondition decreases schedule-reuse rates, TERN removes these unnecessary constraints from the precondition.

We illustrate how TERN works using a simple program based on the aforementioned parallel compression utility `PBZip2`. Figure 2 shows this example. Its input includes all command line arguments in `argv` and the input file data. To compress a file, it spawns

```
// main                       // worker 1               // worker 2
4: pthread_create(worker);
4: pthread_create(worker);
6: worklist.add();
                          12: worklist.get();
6: worklist.add();
                                                   12: worklist.get();
8: worklist.clear();
```

Figure 3: *A synchronization schedule of the example program.* Each synchronization is labeled with its line number in Figure 2.

```
3: 0 < nthread ? true
3: 1 < nthread ? true
3: 2 < nthread ? false
5: 0 < nthread ? true
5: 1 < nthread ? true
5: 2 < nthread ? false
16: ... // constraints collected from compress()
```

Figure 4: *All input constraints collected for the schedule.* Each constraint is labeled with its line number in Figure 2. Constraints collected from function `compress` are later removed by TERN because they have no effects on the schedule. The remaining constraints simplify to $nthread = 2$.

`nthread` worker threads, splits the file accordingly, and compresses the file blocks in parallel by calling function `compress`. To coordinate the worker threads, it uses a synchronized work list. (Here we use work-list synchronization for clarity; in practice, TERN handles Pthread synchronizations.) The example actually has a bug: it is missing `pthread_join` operations at line 7, so the work list may be used by function `worker` after it is cleared at line 8, causing potential program crashes. This bug is based on a real bug in `PBZip2`.

We first illustrate how TERN records a schedule and its precondition. Suppose we run this example with two threads, and TERN records a schedule as shown in Figure 3, which avoids the use-after-free bug. (Other schedules are also possible.) To compute the precondition of the schedule, TERN first records the outcomes of all executed branch statements that depend on input data. Figure 4 shows the set of constraints collected. It then applies advanced program analyses to remove the constraints that concern only local computations and have no effects on the schedule, including all constraints collected from function `compress`. The remaining ones simplify to $nthread = 2$, which forms the precondition of the schedule. TERN stores the schedule and precondition into the schedule cache.

We now illustrate how TERN reuses a schedule. Suppose we want to compress a completely different file also with two threads. TERN will detect that `nthread` satisfies $nthread = 2$, so it will reuse the schedule in Figure 3 to compress the file, regardless of the file data. This execution is reliable because the schedule avoids the use-after-free bug. It is also efficient because the schedule orders only synchronizations and allows the `compress` operations to run in parallel. Suppose we run this program again with four threads. TERN will detect that the input does not satisfy the precondition $nthread = 2$, so it will record a new schedule and precondition.

4.2 Efficiently Enforcing Schedules

Prior work enforces schedules at two different granularities: shared memory accesses or synchronizations, forcing users to trade off efficiency and determinism. Specifically, memory access schedules make data races deterministic but are prohibitively inefficient (e.g., 1.2X-6X as slow as traditional multithreading [4]); synchronization schedules are much more efficient (e.g., average 16% slowdown [19]) because they are coarse grained, but they cannot make programs with data races deterministic, such as our second toy program in §2 and many real-world multithreaded programs [15, 23].

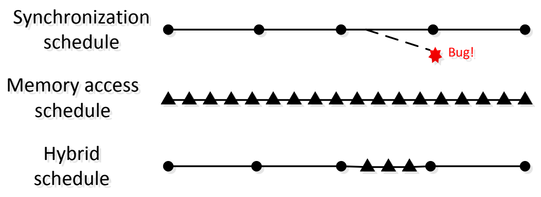


Figure 5: *Hybrid schedule idea.* Circles represent synchronizations, and triangles memory accesses. A synchronization schedule is efficient because it is coarse-grained, but it is not deterministic because data races may still cause executions to deviate from the schedule and fail. A memory access schedule is deterministic, but it is slow because it is fine-grained. A hybrid schedule combines the best of both by scheduling memory access only for the racy portion of an execution and synchronizations otherwise.

This determinism vs. performance challenge has been open for decades in the areas of deterministic execution and replay. Because of this challenge, TERN, our first StableMT system, enforces only synchronization schedules.

To address this challenge, we have built PEREGRINE, our second StableMT system [11]. The insight in PEREGRINE is that although many programs have races, the races tend to occur only within small portions of an execution, and the majority of the execution is still race-free. Intuitively, if a program is full of data races, most of them would have been caught during testing. Empirically, we analyzed the executions of seven real programs with races, and found that, despite millions of memory accesses, only up to 10 data races were detected per execution.

Since races occur rarely, we can schedule synchronizations for the race-free portions of an execution, and resort to scheduling memory accesses only for the "racy" portions, combining both the efficiency of synchronization schedules and the determinism of memory access schedules. These hybrid schedules are almost as coarse-grained as synchronization schedules, so they can also be frequently reused. Figure 5 illustrates this idea.

How can we predict where data races may occur before an execution actually starts? One possible idea is to use static analysis to detect data races at compile time. However, static race detectors are notoriously imprecise: a majority of their reports tend to be false reports, not true data races. Scheduling many memory accesses in the false reports would severely slow down the execution.

PEREGRINE leverages the record-and-reuse approach in TERN to predict races: a recorded execution can effectively foretell what may happen for executions reusing the same schedule. Specifically, when recording a synchronization schedule, PEREGRINE records a detailed memory access trace. From the trace, it detects data races that occurred (with respect to the schedule), and adds the memory accesses involved in the races to the schedule. Now, this hybrid schedule can be efficiently and deterministically enforced, solving the aforementioned open challenge. To reuse the schedule on other inputs, PEREGRINE provides new precondition computation algorithms to guarantee that executions reusing the schedule will not run into any new data races. To enforce an order on memory accesses, PEREGRINE modifies a live program at runtime using a safe, efficient instrumentation framework we built [21].

4.3 Handling Server Programs

Server programs present three challenges for StableMT. First, they may run continuously, making their schedules effectively infinite and too specific to reuse. Second, they often process inputs, i.e., client requests, as soon as the requests arrive. Each request may arrive at a random moment, causing a different schedule. Third, since requests do not arrive at the same time, PEREGRINE cannot

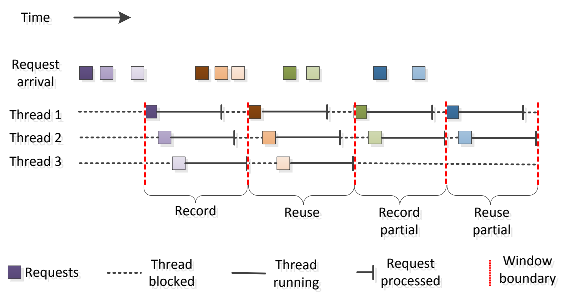


Figure 6: *Recording and reusing schedules for a server program with three threads.* The continuous execution stream is broken down into windows of requests, and PEREGRINE records and reuses schedules across windows.

check them against the precondition of a schedule upfront.

Our observation is that server programs tend to return to the same quiescent states, so PEREGRINE can use these states to split a continuous request stream down to *windows* of requests, as illustrated in Figure 6. Specifically, PEREGRINE buffers requests as they arrive until it gathers enough requests to keep all worker threads busy. It then runs the worker threads to process the requests, while buffering newly arrived requests to avoid interference between windows. If PEREGRINE cannot gather enough requests before a predefined timeout, it proceeds with the partial window to reduce response time. By breaking a request stream into windows, PEREGRINE can record and reuse schedules across windows, stabilizing server programs. Server quiescent states may evolve. For instance, a web server may cache requests in memory. Developers can annotate the functions that query cache, and PEREGRINE treats the return values as inputs and selects proper schedules. Windowing reduces concurrency, but the cost is moderate based on our experiments.

5 Applying Stable Multithreading For Better Program Analysis

As discussed in §3, StableMT can be applied in many ways to improve reliability. In this section, we describe a program analysis framework we have built atop PEREGRINE to effectively analyze multithreaded programs, an open challenge in program analysis.

At the core of this open challenge lies the tradeoff between precision and coverage. Of the two common types of program analysis, static analysis, which analyzes source code without running it, covers all schedules but with poor precision (e.g., many false error reports). The reason is that it must over-approximate the enormous number of schedules, and thus it may analyze a much larger set of schedules, including those impossible to occur at runtime. Not surprisingly, it may detect many "bugs" in the impossible schedules. Dynamic analysis, which runs code and analyzes the executions, precisely identifies bugs because it sees the code's precise runtime effects. However, it has poor coverage because of the exponentially many schedules.

Fortunately, StableMT shrinks the set of possible schedules, enabling a new program analysis approach that gets the best of both static analysis and dynamic analysis. Figure 7 illustrates the high level idea of this approach. It statically analyzes a parallel program over only a small set of schedules at compile time, then dynamically enforces these schedules at runtime. By focusing on only a small set of schedules, we vastly improve the precision of static analysis and reduce false error reports; by enforcing the analyzed schedules dynamically, we guarantee high coverage. Dynamic analysis benefits, too, because it enjoys automatically increased coverage defined as

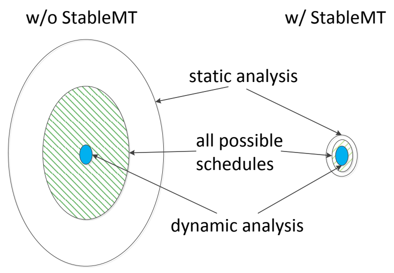


Figure 7: *Program analysis with and without StableMT.* Without StableMT, static analysis tends to analyze many more schedules than all possible schedules; dynamic analysis tends to analyze a tiny fraction of all possible schedules. StableMT shrinks the set of schedules, automatically improving both static analysis and dynamic analysis.

the ratio of checked schedules over all schedules.

A key challenge in implementing this approach is how to statically analyze a program with respect to a schedule. A static tool typically invokes many analyses to compute the final results. To modify this tool for improved precision, a naïve method is to modify every analysis involved, but this method would be quite labor-intensive and error-prone. It may also be fragile: if a crucial analysis is unaware of the schedule, it may easily pollute the final results.

To solve this challenge, we have created a new program analysis framework and algorithms to *specialize* a program according to a schedule. The resultant program has simpler control and data flow than the original program, and can be analyzed with stock analyses, such as constant folding and dead code elimination, for vastly improved precision. In addition, our framework provides a precise *def-use* analysis that computes how values are defined and used in a program. Its results are much more precise than those of regular def-use analyses, because it reports only facts that may occur when the given schedule is enforced at runtime. This precision can be the foundation of many powerful tools such as race detectors.

We illustrate the high-level idea of our framework reusing the example in Figure 2. Suppose we want to build a static race detector that flags when different threads write the same shared memory location concurrently. Although different worker threads do access disjoint file blocks, existing static analysis may not be able to determine this fact. For instance, since `nthread`, the number of threads, is determined at runtime, static analysis often has to approximate these dynamic threads as one or two abstract thread instances. It may thus collapse different threads' accesses to distinct `block` as the same access, emitting false race reports.

Fortunately, such difficult problems are greatly simplified by StableMT. Suppose whenever `nthread` is 2, we always enforce the schedule shown in Figure 3. Since the number of threads is fixed, our framework rewrites the example program to replace `nthread` with 2. It then unrolls the loops and clones function `worker` to give each worker thread its own copy of `worker`, so that distinguishing different worker threads becomes automatic.

Our framework enables the construction of many high coverage and highly precise analyses. For instance, the static race detector we built found seven previously unknown, harmful races in programs extensively checked by previous tools. It emits extremely few false reports, none for 10 out of 18 programs, a huge reduction compared to other static race detectors.

6 Evaluation

In this section, we describe the main results of PEREGRINE, our latest StableMT system. We focus on two evaluation questions:

§6.1: Can PEREGRINE frequently reuse schedules? The higher the

Program-Workload	Reuse Rates (%)	Schedules
Apache-trace	90.3%	100
MySQL-simple	94.0%	50
MySQL-tx	44.2%	109
PBZip2-usr	96.2%	90

Table 2: *Schedule reuse rates under four workloads.* Column **Schedules** indicates the number of schedules in the schedule cache.

reuse rate is, the more stable program behaviors become, and the more efficient PEREGRINE is.

§6.2: Can PEREGRINE efficiently enforce schedules? A low overhead is crucial for programs that frequently reuse schedules.

We choose a diverse set of 18 programs as our evaluation benchmarks. These programs are either widely used real-world parallel programs, such as `Apache` and `PBZip2`, or parallel implementations of computation-intensive algorithms in standard benchmark suites.

6.1 Stability

To evaluate PEREGRINE's stability, i.e., how frequently it can reuse schedules, we compare the preconditions it computes to the best possible preconditions derived from manual inspection. (Some of the manually derived preconditions are shown in Table 1.) For half of the 18 programs, the preconditions it computes are as good as or close to the best preconditions, allowing frequent reuses. For the other programs, the preconditions are more restrictive.

We also evaluate stability by measuring the schedule reuse rates under given workloads. Table 2 shows the results, obtained from TERN and replicable in PEREGRINE. The four workloads are either real workloads collected by us or synthetic workloads used by the developers themselves [10]. For three out of the four workloads, TERN reuses a small number of schedules to process over 90% of the workloads. For `MySQL`-tx, TERN has a lower reuse rate largely because the workload is too random to reuse schedules. Nonetheless, it still processes 44.2% of the workload.

6.2 Efficiency

The overhead of enforcing schedules is crucial for programs that frequently reuse schedules. Figure 8 shows this overhead for both TERN and PEREGRINE. Each bar represents the execution time with TERN or PEREGRINE normalized to traditional multithreading, averaged over 500 runs. For `Apache`, we show the throughput (TPUT) and response time (RESP).

We make two observations about this figure. First, for most programs, the overhead is less than 15%, demonstrating that StableMT can be efficient. For two programs (`water-nsquared` and `cholesky`), the overhead is relatively large because they do a large number of mutex operations within tight loops. However, this overhead is still below 50%, and much lower than the 1.1X-10X overhead of a prior DMT system [4]. Some programs enjoy a speedup because our systems safely skip some blocking operations [10, 11].

Second, PEREGRINE is only slightly slower than TERN, demonstrating that full determinism can be efficient. (Recall that TERN schedules only synchronizations, whereas PEREGRINE additionally schedules memory accesses to make data races deterministic.)

7 Conclusion and Future Work

Through conceiving, building, applying, and evaluating StableMT systems, we have demonstrated that StableMT is feasible; it can stabilize program behaviors for better reliability, work both efficiently and deterministically, and greatly improve precision of static analysis. We believe StableMT offers new promises to solve the grand parallel programming challenge. However, TERN and PEREGRINE

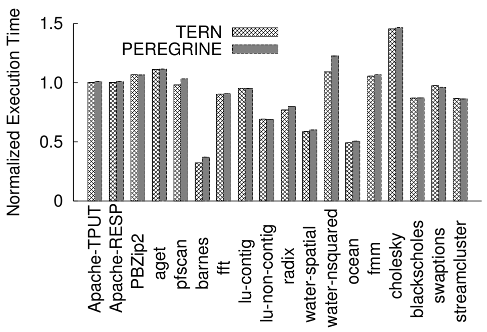


Figure 8: *Normalized execution time when reusing schedules.* A bar with value greater (smaller) than 1 indicates a slowdown (speedup) compared to traditional multithreading. The overhead is smaller than 15% for most programs, and up to 50% for two. Five programs run faster because TERN or PEREGRINE safely skips some blocking operations.

are still research prototypes, not yet ready for wide adoption. Moreover, the ideas we have explored are just the first few in this direction of StableMT; the bulk of work still lies ahead:

- At the system level, can we build efficient, lightweight StableMT systems that work automatically with all multithreaded programs? TERN and PEREGRINE require recording executions and analyzing source code, which can be heavyweight. As the number of cores increases, can we build StableMT systems that scale to hundreds of cores?

- At the application level, we have only scratched the surface: improving program analysis is just one possible application. There are many others, such as improving testing coverage, verifying a program with respect to a small set of dynamically enforced schedules, and optimizing thread scheduling and placement based on a schedule because it effectively predicts the future. Moreover, the idea of stabilizing schedules may apply to other parallel programming methods such as MPI, OpenMP, and Cilk-like tasks.

- At the conceptual level, can we reinvent parallel programming to greatly reduce the set of schedules? For instance, a multithreading system may disallow schedules by default, and only allow those that developers explicitly write code to enable. Since developers are already of different calibers, we may let only the best programmers decide what schedules to use, reducing the likelihood of programming errors.

We invite readers to join us in exploring this fertile and exciting direction of stable multithreading and reliable parallelism.

Acknowledgments

We thank everyone who helped with the work [10, 11, 22, 25] summarized here. In particular, John Gallagher built the alias analysis in PEREGRINE, Chia-Che Tsai conducted the evaluation of TERN, and Huayang Guo helped evaluate PEREGRINE. We thank Al Aho, Remzi Arpaci-Dusseau, Tom Bergan, Emery Berger, Luis Ceze, Xi Chen, Jason Flinn, Roxana Geambasu, Gail Kaiser, Jim Larus, Jinyang Li, Jiri Simsa, Ying Xu, and the anonymous reviewers for their many helpful comments. This work was supported in part by NSF, AFRL, ONR, an NFS Career award, an AFOSR YIP Award, and a Sloan Research Fellowship.

References

[1] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, Oct. 2009.

[2] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient system-enforced deterministic parallelism. *Commun. ACM*, 55(5):111–119, May 2012.

[3] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proceedings of the Eighth International SPIN Workshop on Model Checking of Software (SPIN '01)*, pages 103–122, May 2001.

[4] 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.

[5] 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.

[6] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM*, 53:66–75, Feb. 2010.

[7] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel java. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '09)*, pages 97–116, Oct. 2009.

[8] B. Cantrill and J. Bonwick. Real-world concurrency. *Commun. ACM*, 51(11): 34–39, Nov. 2008.

[9] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[10] 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.

[11] H. Cui, J. Wu, J. Gallagher, H. Guo, and J. Yang. Efficient deterministic multithreading through schedule relaxation. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, pages 337–351, Oct. 2011.

[12] 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.

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

[14] 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)*, pages 327–336, Oct. 2011.

[15] 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.

[16] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the Eighth Symposium on Operating Systems Design and Implementation (OSDI '08)*, pages 267–280, Dec. 2008.

[17] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI '07)*, pages 89–100, June 2007.

[18] C. O'Hanlon. A conversation with John Hennessy and David Patterson. *Queue*, 4(10):14–22, Dec. 2006.

[19] 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.

[20] Z. Shao. Certified software. *Commun. ACM*, 53(12):56–66, Dec. 2010.

[21] 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.

[22] J. Wu, Y. Tang, G. Hu, H. Cui, and J. Yang. Sound and precise analysis of parallel programs through schedule specialization. In *Proceedings of the ACM SIGPLAN 2012 Conference on Programming Language Design and Implementation (PLDI '12)*, pages 205–216, June 2012.

[23] 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.

[24] J. Yang, A. Cui, S. Stolfo, and S. Sethumadhavan. Concurrency attacks. In *the Fourth USENIX Workshop on Hot Topics in Parallelism (HOTPAR '12)*, June 2012.

[25] J. Yang, H. Cui, and J. Wu. Determinism is overrated: What really makes multithreaded programs hard to get right and what can be done about it? In *the Fifth USENIX Workshop on Hot Topics in Parallelism (HOTPAR '13)*, June 2013.