

# Determinism Is Overrated: What Really Makes Multithreaded Programs Hard to Get Right and What Can Be Done about It?

Junfeng Yang, Heming Cui, Jingyue Wu  
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, and researchers have recently dedicated much effort to bringing determinism into multithreading. In this paper, 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

Multithreaded programs are notoriously difficult to write, test, analyze, debug, and verify, much harder than the sequential versions. These programs are the most widespread parallel programs, yet experts consider reliable parallelism “something of a black art” [6] and one of the grand challenges in computing [1, 14]. Unsurprisingly, widespread parallel programs are plagued with insidious concurrency bugs [12], such as data races and deadlocks. 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 [19].

In recent years, two technology trends have made the challenge of reliable multithreading 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 often employ multithreading for performance.

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 [10]: 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 [2–5, 9, 11, 15]. 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 believe nondeterminism is responsible for only a small piece of the puzzle. We argue 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* [7]. Leveraging this insight, we envision a new approach we call *stable multithreading (SMT)*<sup>1</sup> 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. SMT and DMT are not mutually exclusive: a system can be both deterministic and stable.

To demonstrate the feasibility of SMT, we have built two compiler and runtime systems, TERN [7] and PEREGRINE [8], to address complementary challenges in implementing SMT systems. To demonstrate key benefits of SMT, we have built a program analysis framework that leverages SMT to greatly improve the precision of static analysis [17]. These systems are designed to be compatible with existing hardware, operating systems, thread libraries, and programming languages to simplify adoption. They are also mostly transparent to developers.

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 positives.

In the rest of this paper, we present our view on why multithreading is hard to get right (§2); describe our SMT approach, its benefits (§3), and the three SMT systems we built (§4 and §5); and conclude (§6).

## 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.

<sup>1</sup>Not to be confused with simultaneous multithreading.

### 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 the ones appeared in previous studies [12, 19], depend much more on the schedules than the inputs (i.e., once the schedule is fixed, the bug occurs for all inputs allowed by the schedule). Thus, recent research on testing multithreaded programs (e.g., [13]) 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

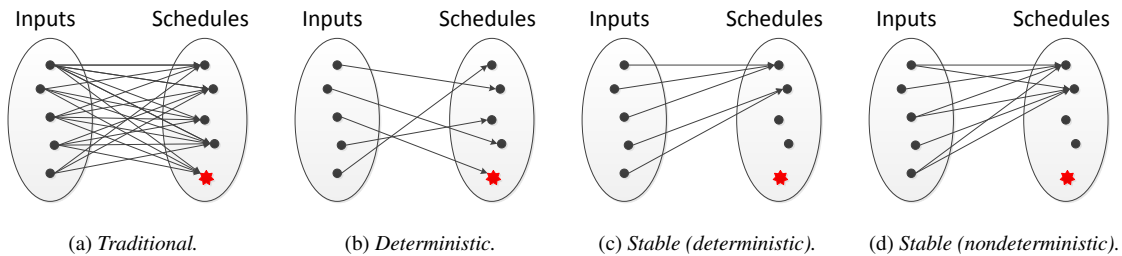


Figure 1: Different multithreading approaches. Red stars represent buggy schedules.

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  $\frac{(mk)!}{k!} \geq (m!)^k$  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 schedules, but it often redundantly tests the same schedules while missing others. Recent tools (e.g., [13]) 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 Overrated

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 needle-in-a-haystack 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 for reliability, 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. A single bit flip of a file should usually not cause a compression utility to crash. Unfortunately, determinism does not provide stability and, if naïvely implemented, even undermines it.

To illustrate, consider a worst-case system depicted in Figure 1b that 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 (ad)venture 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 avoid the worst case, researchers have to augment determinism with other techniques. To support debug `printf`, some propose to temporarily revert to nondeterministic execution [9]. DMP [9], CoreDet [3], and Kendo [15] 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, observed in our experiments [7]. Our TERN and PEREGRINE systems and the DTHREADS system built subsequently to TERN by others [11] combine DMT with SMT (elaborated next section) to frequently reuse schedules on a wide range of inputs for stability.

**Reliability  $\Rightarrow$  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 needle-in-a-haystack 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 pop-

ular multithreaded programs, ranging from server programs such as Apache, to desktop utilities such as parallel compression utility PBZip2, to parallel implementations of computation-intensive algorithms such as fast Fourier transformation. They use diverse synchronizations such as locks, semaphores, condition variables, and barriers. Our investigation reveals 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 SMT systems we built invalidated this presumption. With carefully designed schedule representations, our systems incurred less than 15% overhead enforcing schedules on different inputs for most evaluated programs [8]. We believe this moderate overhead is worth the gains in reliability. In general, users most likely prefer a slightly slower program that computes correct results than a faster program that frequently crashes.

Leveraging these insights, we have invented *stable multithreading (SMT)*, 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

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 listed, one schedule out of the class suffices to process any input satisfying the constraints shown in the third column, assuming typical setups (e.g., no system call failures or signals). We describe how to compute such constraints in §4.

the needle-in-a-haystack challenges at once. In addition, SMT 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.

SMT and DMT are orthogonal. SMT aims to reduce the set of schedules for *all* inputs, whereas DMT aims to reduce the schedules for *each* input (down to one). A SMT system may be either deterministic or nondeterministic. Figure 1c and Figure 1d depict two SMT 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), so the challenges caused by nondeterminism are easy to solve.

### 3.1 Benefits

By vastly reducing schedules, SMT brings numerous reliability benefits to multithreading. We describe several:

**Testing.** SMT 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 dif-

ferent 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. Interactive theorem proving becomes easier, too, because verifiers need prove theorems only on the set of schedules enforced in the field. We will discuss these benefits 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 discuss this benefit in §4.1.

### 3.2 Caveats

SMT 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 complex conditions involving most bits in the input. 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.

SMT 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 three main challenges:

- How can we compute the schedules to map inputs to? They 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 cannot

deviate from schedules despite data races, and “efficiently” so overhead does not offset the benefits.

- 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 SMT systems, which resulted in two SMT prototypes, TERN [7] and PEREGRINE [8], 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 [11].

#### 4.1 Computing Schedules

TERN, our first SMT system, computes schedules by recording them from past executions; the recorded schedules can then be reused on future inputs to stabilize program behaviors. It 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.

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.

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.

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. (The name TERN comes from the Arctic Tern, a bird species that migrates the farthest among all animals.)

Second, TERN explicitly stores schedules, so developers and users can flexibly choose what schedules to record and when. For instance, developers can populate

the cache during testing and 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 and 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.

#### 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 [3]); synchronization schedules are much more efficient (e.g., average 16% slowdown [15]) 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 [12, 18]. 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 SMT system, enforces only synchronization schedules.

To address this challenge, we have built PEREGRINE, our second SMT system [8]. 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 memory accesses only for the “racy” portions, combining both the efficiency of synchronization schedules and the determinism of memory access schedules.

#### 4.3 Handling Server Programs

Server programs present three challenges for SMT. 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 check them against the precondition of a schedule upfront.

Our observation is that server programs tend to re-

turn to the same quiescent states, so PEREGRINE can use these states to split a continuous request stream down to *windows* of requests. 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, stable multithreading 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 covers all schedules but with poor precision (e.g., many false positives). 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 precisely identifies bugs because it sees the code’s precise runtime effects. However, it has poor coverage because of the exponentially many schedules.

Fortunately, SMT shrinks the set of possible schedules, enabling a new program analysis approach that gets the best of both static analysis and dynamic analysis. Specifically, our framework 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 positives; by enforcing the analyzed schedules dynamically, we guarantee high coverage. Dynamic analysis benefits, too, because it enjoys automatically increased coverage defined as the ratio of checked schedules.

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 positives, none for 10 out of 18 programs, a huge reduction compared to other static race detectors.

## 6 Conclusion and Future Work

Through conceiving, building, applying, and evaluating SMT systems, we have demonstrated that SMT is feasible; it can stabilize program behaviors for better reliability, work both efficiently and deterministically, and greatly improve precision of static analysis. We believe SMT offers new promises to solve the grand parallel programming challenge. However, TERN and PEREGRINE are still research prototypes, not yet ready for wide adoption. Moreover, the ideas explored are just the first few in this direction of SMT; the bulk of work lies ahead:

- At the system level, can we build efficient, lightweight SMT 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 SMT systems that scale to hundreds of cores?
- At the application level, we have only scratched the surface: improving program analysis is just one cool 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 exciting direction of stable multithreading and reliable parallelism.

## Acknowledgments

We thank Al Aho, Tom Bergan, Emery Berger, Luis Ceze, Jim Larus, Ying Xu, and the anonymous reviewers for their many helpful comments. This work was supported in part by AFRL FA8650-11-C-7190, FA8650-10-C-7024, and FA8750-10-2-0253; ONR N00014-12-1-0166; NSF CCF-1162021, CNS-1117805, CNS-1054906 (CAREER award), and CNS-0905246; an AFOSR YIP Award; and a Sloan Research Fellowship.

## References

- [1] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiawicz, 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, 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. 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.
- [4] E. Berger, T. Yang, T. Liu, D. Krishnan, and A. Nark. Grace: safe and efficient concurrent programming. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '09)*, pages 81–96, Oct. 2009.
- [5] 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, 2009.
- [6] B. Cantrill and J. Bonwick. Real-world concurrency. *Commun. ACM*, 51(11):34–39, Nov. 2008.
- [7] 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.
- [8] 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.
- [9] 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.
- [10] E. A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.
- [11] 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.
- [12] 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.
- [13] 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.
- [14] C. O'Hanlon. A conversation with john hennessy and david patterson. *Queue*, 4(10):14–22, 2007.
- [15] 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.
- [16] 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.
- [17] 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.
- [18] 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.
- [19] 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.