# Make Parallel Programs Reliable with Stable Multithreading

Junfeng Yang, Heming Cui, Jingyue Wu, John Gallagher, Chia-Che Tsai, Huayang Guo, Yang Tang, Gang Hu
Department of Computer Science
Columbia University

## ABSTRACT

Our accelerating computational demand and the rise of multicore hardware have made parallel programs increasingly pervasive and critical. Yet, these programs remain extremely difficult to write, test, analyze, debug, and verify. In this article, we provide our view on why parallel programs, specifically multithreaded programs, are difficult to get right. We present 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 [7], 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" [9] 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 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. Many physical-world computations, such as search and social networking, are now computerized and hosted in the cloud. These computations are massive, run on the "big data," and serve millions of users and devices. To cope with these massive computations, almost all services employ parallel programs to boost performance.

If reliable software is an overarching challenge of computer science, reliable parallelism is surely the keystone. This keystone challenge has drawn decades of research efforts, producing numerous ideas and systems, ranging from new hardware, new programming languages, new programming models, to tools that detect, debug, avoid, and 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. We target these programs because they are the most widespread type of parallel programs. Many legacy programs, such as the `Apache` web server and the `MySQL` database, are multithreaded. New multithreaded programs are being written every day. This prevalence of multithreading is unsurprising given its wide support from many platforms (e.g., Linux, Windows, and MacOS) and languages (e.g., Java and C++11). For the same reasons, we believe multithreading will remain prevalent in the foreseeable future.

Unlike sequential programs, multithreaded programs are *nondeterministic*: repeated executions of the same multithreaded program on the same input may yield different (e.g., correct v.s. buggy) behaviors, depending on how the threads interleave. Conventional wisdom has long blamed this nondeterminism for the challenges in reliable multithreading [13]. For instance, testing becomes 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 nondeterminism, several groups of brilliant researchers have dedicated significant effort to build deterministic multithreading systems [2, 4, 6, 8, 12, 14, 19].

Unfortunately, as explained later in this article, nondeterminism is responsible for only a small piece of the puzzle. Its cure, determinism, is neither sufficient nor necessary for reliability. Worse, determinism sometimes *harms* reliability rather than improves it.

We believe the challenges in reliable multithreading have a rather quantitative root cause: multithreaded programs have too many possible thread interleavings, or *schedules*. For complete reliability, all schedules must be correct. Unfortunately, ensuring so requires a great deal of resources and efforts simply because the set of schedules is just enormous. For instance, testing all possible schedules demands astronomical CPU time. (See §2 for detailed discussion.)

We attacked this root cause by asking: are *all* the exponentially 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 (SMT)* that exponentially reduces the set of schedules for reliability. We have built three systems: TERN [10] and PEREGRINE [11], two compiler and runtime systems to address key challenges in implementing SMT systems; and an effective program analysis framework atop SMT to demonstrate key benefits of SMT [22]. 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 to save manual labor. We leave it for future work to create new SMT programming models and methods that help active development.

Our initial results with these systems are promising. Evaluation on a diverse set of widespread multithreaded programs, including `Apache` and `MySQL`, show that TERN and PEREGRINE dramatically reduce schedules. For instance, they reduce the number of schedules needed by parallel compression utility `PBZip2` down to *two* schedules per thread count, 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 remaining of this article, we first present our view on why multithreaded programs are hard to get right. We then describe our SMT approach, its benefits, and the three SMT 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 are correct, but some are *buggy* and cause various failures such as program crashes, incorrect computations, and deadlocked executions. 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 is buggy and 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 is buggy and corrupts `balance`.

## 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 the schedules described, all their other schedules lead to correct executions.

This nondeterminism appears to raise 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

To make a multithreaded program completely reliable, we must ensure that no schedules are buggy, either manually by thinking really hard or automatically by applying tools. Either way, the number of the schedules determines the amount of efforts and resources needed.

Unfortunately, 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 $n$ threads competing for a lock, each order of lock acquisitions forms a schedule, easily yielding $n!$ total schedules. (A lock implementation that grants locks to threads based on the arrival order, such as a ticket lock, does not reduce the set of schedules, because the threads may *arrive* at the lock operations in $n!$ different orders.) Aggregated over all inputs, the number of schedules is even greater.

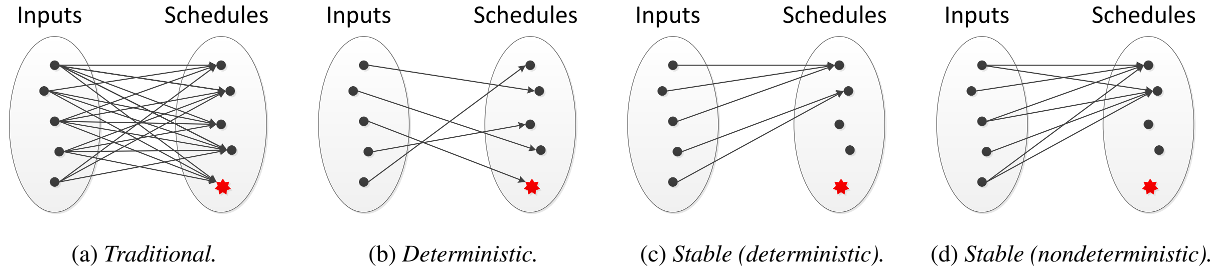
Finding a few buggy schedules in these exponentially many schedules thus raises a series of "needle-in-a-haystack" challenges. For instance, to write correct multithreaded programs, developers must carefully synchronize their code to weed out the buggy schedules. As usual, humans err when they must scrutinize 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 repeatedly while missing others. Recent work made testing more powerful by systematically enumerating through schedules for bugs [16], but we seriously lack resources to cover more than a tiny fraction of all exponentially many schedules.

## 2.4 Determinism is Overrated

Researchers have built several systems to make multithreading deterministic, hoping to address the challenges raised by nondeterminism. Yet, 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 that multithreaded programs simply have too many schedules. We argue that determinism, the cure of nondeterminism, is neither sufficient nor necessary for reliability.

**Determinism $\not\Longrightarrow$ reliability.** Determinism provides only a narrow type of repeatability: it guarantees that executions are repeatable if both the program and the input remain exactly the same, and has no jurisdiction otherwise. However, developers often expect repeatabilities for executions on slightly varied programs or inputs. For instance, adding a `printf` statement purely for debugging should in principle not make the bug disappear. Unfortunately, determinism does not provide these expected repeatabilities, and sometimes even undermines them [2, 5, 10].

To illustrate, consider some existing deterministic systems that force a multithreaded program to always use the same—but




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

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

arbitrary—schedule to process the same input. Figure 1b depicts such a system. This arbitrary mapping *destabilizes* program behaviors over multiple inputs. A slight input change, as slight as a 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 actually raises new reliability challenges. For instance, as confirmed in our experiments [10], testing one input provides little assurance on very similar inputs, despite that the input differences 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. In addition to inputs, these deterministic systems destabilize program behaviors over minor code changes as well, so adding a `printf` for debugging may cause the bug to deterministically disappear.

Another problem with an arbitrary mapping as in Figure 1b is that the number of all possible schedules remains enormous (asymptotically as enormous as the minimum of (1) the number of all possible inputs and (2) the number of all possible schedules). Thus, the needle-in-a-haystack challenges remain. For instance, testing all schedules may now require testing all inputs, another difficult challenge we have no idea how to solve.

For those curious minds, deterministic multithreading systems may be implemented in several ways. A frequent scheme is to schedule threads based on *logical clocks* [4, 19], instead of physical clocks which change nondeterministically across executions. Specifically, each thread maintains a logical clock that ticks based on the code the thread has run. For instance, if a thread has completed 50 `load` instructions, tick its clock by 50. Moreover, threads communicate only when their logical clocks have deterministic values (e.g., the smallest across the logical clocks of all threads [19]). In short, local executions tick logical clocks deterministically, and logical clocks force threads to communicate deterministically. By induction, a multithreaded execution becomes deterministic. It is straightforward to see that a slight input change or an additional `printf` statement may change the number of completed load instructions, thus altering the schedule and destabilizing program behaviors.

**Reliability $\not\Rightarrow$ determinism.** Determinism is not necessary for reliability because a nondeterministic system with a small set of schedules can be made reliable easily. Consider an extreme case. The system depicted in Figure 1d is nondeterministic because it maps some inputs to more than one schedules. However, each input maps to at most two schedules, so 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.

# 3 Shrink 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 popular multithreaded programs, ranging from server programs such as `Apache`, to desktop utilities such as the aforementioned `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 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 fixes the metadata in the input, but it allows the computational data to vary. Thus, 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. As long as the number of threads remains the same, `PBZip2` can use two schedules (one if the file can be evenly divided 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.

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 a most efficient schedule. However, results from the SMT systems we built

| 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.

invalidated this presumption. With carefully designed schedule representations (§4.2), our systems incurred less than 15% overhead enforcing schedules for most evaluated programs (§6). We believe this moderate overhead is worth the gains in reliability. In general, users most likely prefer a program that runs 20% slower but crashes 80% less often.

Leveraging these insights, we have invented *stable multithreading (SMT)*, a new multithreading approach that vastly shrinks the haystack by reusing schedules over inputs, addressing all the needle-in-a-haystack challenges at once. In addition, SMT also stabilizes program behaviors over inputs that map to the same schedule and minor program changes that do not affect the schedules. Figure 1c and Figure 1d depict two SMT systems.

SMT systems may be either deterministic, such as the many-to-one mapping in Figure 1c, or nondeterministic, such as the many-to-few mapping Figure 1d. A many-to-few nondeterministic 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 shrinking the haystack, SMT brings numerous reliability benefits to multithreaded programs. We describe several below.

**Testing.** By reducing the set of schedules, SMT automatically increases testing coverage, defined as the ratio of tested schedules over all schedules. For instance, consider `PBZip2` again which needs only two schedules per thread count. Testing 32 schedules effectively covers from 1 to 16 threads. Given that (1) `PBZip2` achieves peak performance when the thread count is identical or close to the core count and (2) the core count of a typical machine is up to 16, 32 tested schedules can practically cover all executions 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 state explosion problem. Interactive theorem proving becomes easier, too, because verifiers need prove theorems only over 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

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 every bit of the input. For these programs, SMT may degenerate to DMT.

SMT provides repeatabilities over similar inputs and programs when the input or program changes do not affect schedules, such as adding a `printf` for debugging. As discussed in the previous section, it is already better than deterministic multithreading systems. 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 and easy to compute.
- 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.
- 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 [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; others have also built systems that stabilize schedules [2, 14].

### 4.1 Computing Schedules

Crucial to implementing SMT 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.)

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 over 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 recording and checking one schedule over many executions that reuse the schedule. Recording and checking a schedule is more expensive than reusing a schedule, but, fortunately, TERN need do it only once for each schedule and then reuse the schedule over many inputs, amortizing the overhead.

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 (for example) TERN tries to enforce a schedule 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 research 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 `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`.

```
1 :  main(int argc, char *argv[]) {
2 :    int i, nthread = atoi(argv[1]);
3 :    for(i=0; i<nthread; ++i)
4 :      pthread_create(worker); // create worker threads
5 :    for(i=0; i<nthread; ++i)
6 :      worklist.add(read_block(i)); // add block to work list
7 :    // Error: missing pthread_join() operations
8 :    worklist.clear(); // clear work list
9 :    ...
10:  }
11:  worker() { // worker threads for compressing file blocks
12:    block = worklist.get(); // get a file block from work list
13:    compress(block);
14:  }
15:  compress(block_t block) {
16:    if(block.data[0] == block.data[1])
17:      ...
18:  }
```

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.

```
// 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 a line number from Figure 2.

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.

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 a line number from 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 $nthread = 2$.

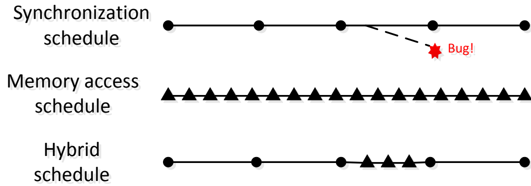



Figure 5: *Hybrid schedule idea.* Circles represent synchronizations, and triangles memory accesses. A synchronization schedule is efficient because it is coarse-grained, but it is not deterministic because data races may still cause executions to deviate from the schedule and potentially hit a bug. 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.

### 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 (§2) and many real-world multithreaded programs [15, 23]. This determinism v.s. 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 [11]. The insight in PEREGRINE is that although many programs have races, these races tend to occur only within minor portions of an execution, and the majority of the execution is still race-free. 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

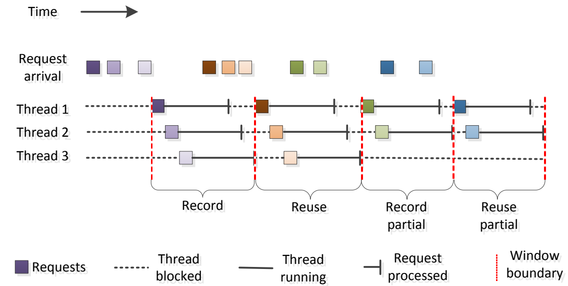



Figure 6: *Recording and reusing schedules for a server program with three threads.* The continuous execution stream is broken down into windows of requests, and PEREGRINE records and reuses schedules across windows.

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 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 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 (possibly partial) windows, stabilizing server programs. 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, a well-known open challenge in program analysis.

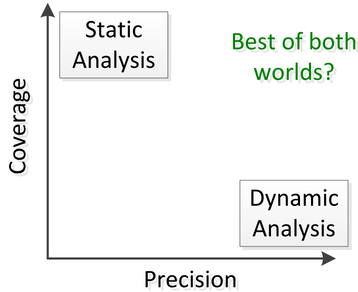



Figure 7: *The coverage v.s. precision tradeoff facing program analysis.* For multithreaded programs, static analysis tends to have high coverage (miss no bugs) but imprecise (many false error reports), whereas dynamic analysis tends to be precise (no false reports) but cover few schedules (miss many bugs).

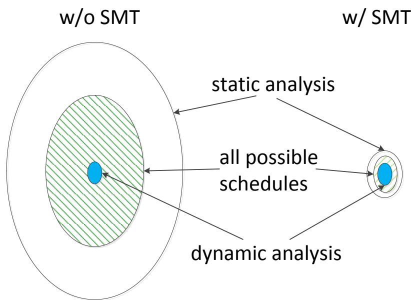



Figure 8: *Program analysis with and without SMT.* Without SMT, static analysis tends to analyze many more schedules than all possible schedules; dynamic analysis tends to analyze a tiny fraction of all possible schedules. SMT shrinks the schedule domain, automatically improving both static analysis and dynamic 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 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, which runs code and analyzes the executions, precisely identifies bugs because it sees the code's precise runtime effects. However, running code takes resources, and we just cannot afford to enumerate more than a tiny fraction of all possible schedules. Consequently, the next execution may well hit an unchecked schedule with errors. Figure 7 illustrates this difficult tradeoff of coverage and precision.

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. Figure 8 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 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. The results computed by this analysis are much more precise than regular def-use analyses, because ours reports only facts that may occur when the given schedule is enforced at runtime. These precise results 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 a variety of difficult problems. 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 one access, emitting false race reports.

Fortunately, such difficult problems are greatly simplified by SMT. 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, so distinguishing different worker threads becomes automatic.

Our framework enables the construction of many high coverage and highly precise analyses. For instance, we have built a static race detector within the framework. It found seven previously unknown, harmful races in programs extensively checked by previous tools. Moreover, it emits extremely few false reports, zero for 10 out of 18 programs, a huge reduction compared to other static race detectors.

# 6 Evaluation

To demonstrate the feasibility and benefits of SMT, we describe the main results of PEREGRINE, our latest SMT system. The following subsections focus on two evaluation questions:

§6.1: Can PEREGRINE frequently reuse schedules? The higher the 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 the `Apache` web server and the aforementioned `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

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

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

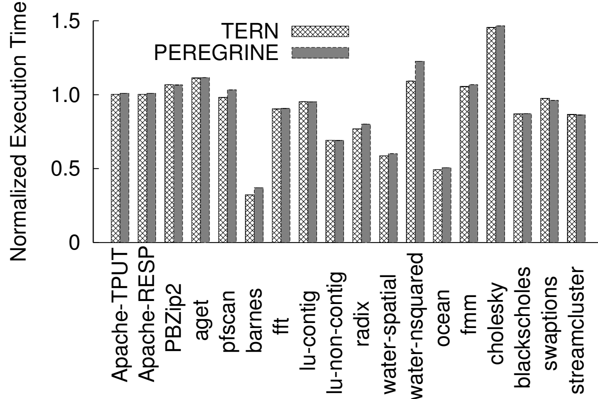



Figure 9: *Normalized execution time when reusing schedules.* A bar with value greater (smaller) than 1 indicates a slowdown (speedup) compared to traditional multithreading. The overhead is smaller than 15% for most programs, and up to 50% for two. Five programs run faster because TERN or PEREGRINE safely skips blocking operations.

the manually derived preconditions are shown in Table 1.) For half of the 18 programs, the preconditions PEREGRINE computes are as good as or close to the manually derived preconditions, and PEREGRINE can indeed frequently reuse schedules for the programs. 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 we collect or synthetic workloads used by the developers themselves (§A). For three out of the four workloads, TERN can reuse 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, TERN manages to process 44.2% of the queries with a small number of schedules.

### 6.2 Efficiency

The overhead of enforcing schedules is crucial for programs that frequently reuse schedules. Figure 9 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 numbers are less than 15%, demonstrating that SMT 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 1.2X-6X overhead of a prior DMT system [4]. For five programs (`barnes`, `lu-non-contig`, `radix`, `water-spatial`, and `ocean`), there is actually a speedup because TERN and PEREGRINE can safely skip blocking operations [10, 11].

Second, PEREGRINE's overhead is only slightly larger than TERN's, demonstrating that full determinism can also be efficient. Recall that TERN schedules only synchronizations, whereas PEREGRINE additionally schedules memory accesses to make data races deterministic. The TERN and PEREGRINE bars in the figure are very close, showing that the additional overhead in PEREGRINE is negligible.

## 7 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 promise to solve the grand parallel programming challenge. However, TERN and PEREGRINE are still research prototypes, and not yet ready for wide adoption. Moreover, the ideas we have explored are just the first few in this direction of SMT; the bulk of work still 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 we can build atop SMT. 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 models such as MPI and OpenMP.
- At a concept 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.

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

[6] 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 (OOP-*

SLA '09)*, pages 81–96, Oct. 2009.

[7] 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, February 2010.

[8] 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.

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

[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)*, Oct. 2011.

[12] D. R. Hower, P. Dudnik, M. D. Hill, and D. A. Wood. Calvin: Deterministic or not? free will to choose. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pages 333–334, 2011.

[13] E. A. Lee. The problem with threads. *Computer*, 39(5):33–42, 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)*, 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, 2007.

[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)*, 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.

# APPENDIX

## A Workloads for Evaluating Stability

We used the following four workloads for evaluating TERN's stability:

- `Apache`-CS: a four-day trace from the Columbia CS website with 122,000 HTTP requests. A script replays this trace at a rate of 100 concurrent requests per second onto `Apache`.
- `MySQL`-simple: random SQL select queries issued by SysBench onto `MySQL`. SysBench is used by MySQL developers for performance benchmarking.
- `MySQL`-tx: random SQL select, update, delete, and insert queries issued by SysBench onto `MySQL`.
- `PBZip2`-usr: a random selection of 10,000 files from `/usr` on our evaluation machine.

For each workload, we first randomly select 1%-3% of the workload to populate the schedule cache. We then run the entire workload to measure the overall reuse rates.