DOI:10.1145/2500875

Stable multithreading dramatically simplifies the interleaving behaviors of parallel programs, offering new hope for making parallel programming easier.

BY JUNFENG YANG, HEMING CUI, JINGYUE WU, YANG TANG, AND GANG HU

Making Parallel Programs Reliable with Stable Multithreading

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

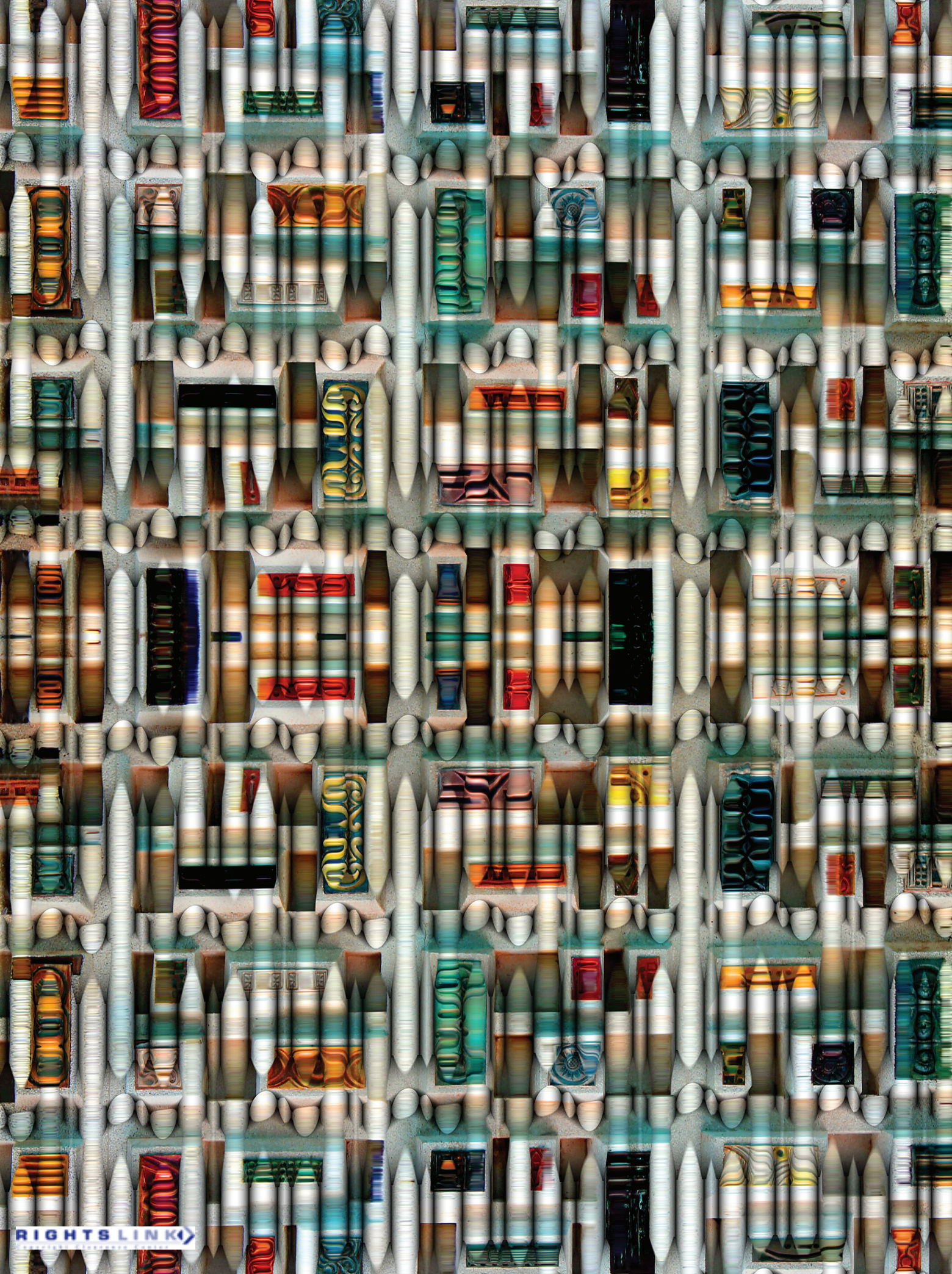
However, the same level of success has not yet propagated to parallel programs, which are notoriously difficult to write, test, analyze, debug, and verify, much more so than the sequential versions. Experts consider reliable parallelism "something of a black art"[8] and one of the grand challenges in computing.[1,18] But widespread parallel programs are plagued with insidious concurrency bugs[15] (such as data races, including concurrent accesses to the same memory location with at least one write, and deadlocks, including threads circularly waiting for resources). Some of the worst of them killed people in the Therac-25 radiation-therapy incidents by generating massive overdoses of radiation and caused the Northeast power blackout of 2003. These bugs can be exploited by attackers violating confidentiality, integrity, and availability of critical systems.[24]

Over the past decade, two technology trends have made the challenge of reliable parallelism even 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 and developers resorting to parallel code for best performance on multicore processors. The second is accelerating computational demand. Scientific computing, video and image processing, financial simulation, big-data analytics, Web search, and online social networking all involve massive computations and various kinds of parallel programs to

» key insights

- **Getting multithreaded programs right is difficult because they have too many possible thread interleavings, or schedules, not because they are nondeterministic.**
- **Not all schedules are necessary; for many programs, a small set is enough to process all possible inputs.**
- **StableMT dramatically reduces the set of schedules while keeping overhead low, greatly enhancing almost all reliability techniques, including testing, debugging, and program analysis.**

ILLUSTRATION BY JOEL ORMSBY



maximize performance.

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

We look to attack fundamental, open problems in making shared-memory multithreaded programs reliable. These programs express concurrency through threads, essentially lightweight, sequential processes that share memory. We target them because they are the most widespread type of parallel programs, with mature support from hardware, operating systems, libraries, and programming languages. Moreover, they will likely remain prevalent for the foreseeable future.

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

To eliminate nondeterminism, several groups of researchers, including us, have dedicated themselves to building deterministic multithreading (DMT) systems[2,4,5,7,12,14,19] that force multithreaded programs to always execute the same thread interleaving, or schedule, on the same input, 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.

Removing unnecessary schedules from the haystack would make the needles easier to find.

However, nondeterminism is only a small piece of the puzzle, and determinism (the cure for nondeterminism) is not as useful as commonly perceived, being neither sufficient nor necessary for reliability. It is not sufficient because a perfectly deterministic system can map each input to an arbitrary schedule, so small input perturbations lead to vastly different schedules, artificially reducing a 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.

What makes multithreading difficult for programmers to get right is quantitative; multithreaded programs have too many schedules. The number of schedules for each input is already enormous because the parallel threads can 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 schedules (so developers can prevent them) is like finding needles in a haystack. Although DMT reduces schedules for each input, it could map each input to a different schedule, so the total set of schedules for all inputs remains enormous.

We have attacked this root cause by asking if all the many schedules are necessary. Our 2010 study found that many real-world programs can use a small set of schedules to efficiently process a range of inputs.[10] Leveraging this insight, we envision a new approach we call stable multithreading, or StableMT, that reuses each schedule on a range of inputs, mapping all inputs to a dramatically reduced set of schedules. By vastly shrinking the haystack, the needles are much easier to find. By mapping many inputs to the same schedule, program behaviors are stabilized against small input perturbations. StableMT and DMT are not mutually exclusive; a system can be both deterministic and stable.

To realize our vision of StableMT, we built several systems: Tern[10] and Peregrine;[11] two compiler and run-

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

Our initial results are promising. Evaluation on a diverse set of multithreaded programs, including the `Apache` Web server and the `MySQL` database, shows TERN and PEREGRINE reduce schedules dramatically; for instance, under a typical setup, they reduce the number of schedules needed by parallel compression utility `PBZip2` down to two schedules for each different number of threads, regardless of file content. Their overhead is moderate, less than 15% for most programs. Our program-analysis framework enables 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.

Difficult to Get Right

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

Inputs, schedules, and buggy schedules. We say "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 command-line arguments, return values of external functions (such as `gettimeofday`), and any external data that can affect program execution. We say "schedule" to broadly refer to the (partially or totally) ordered set of communication operations in a multithreaded execution, including synchronizations (such as `lock` and `unlock` operations) and shared memory accesses (such as `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 this toy program:

```
// thread          // 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. Now consider another example, this one with data races on `balance`:

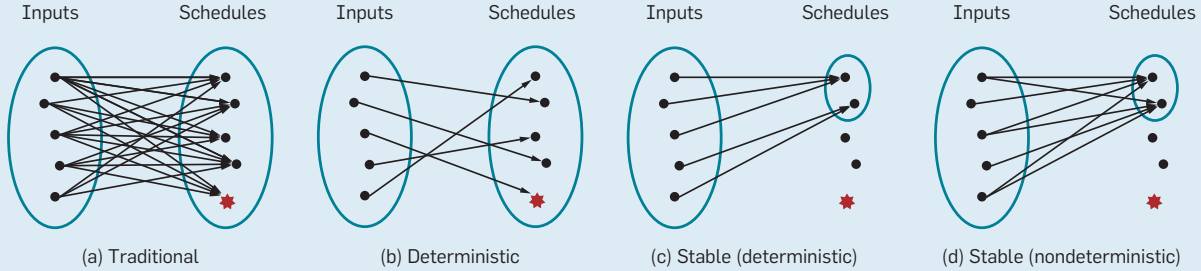
```
// thread 1        // thread 2
// deposit 100     // withdraw 100
t = balance + 100;
                   balance =
                   balance - 100;
balance = t;
```

The schedule with the statements executed in this order 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 in Lu et al.[15] and Yang et al.[24]) depend much more on the schedules than on inputs (such as once the schedule is fixed, the bug occurs for all inputs allowed by the schedule). Recent research on testing multithreaded programs (such as Musuvathi et al.[16]) focuses on testing schedules to find the buggy ones.

Challenges caused by nondeterminism. A multithreaded program is nondeterministic because, even with the same program and input, different schedules can still lead to different behaviors; for

Figure 1. Multithreading approaches.

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



instance, the two toy programs do not always run into the bugs. Except for the schedules described, the other schedules lead to correct executions.

This nondeterminism involves 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 could 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 outlines the traditional multithreading approach, conceptually, a many-to-many mapping, where one input can execute under many schedules due to nondeterminism, and many inputs can 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.

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

Finding a few buggy schedules in these exponentially many schedules raises a series of needle-in-a-haystack challenges; for instance, to write correct multithreaded programs, developers must carefully synchronize their code to weed out the buggy schedules. As usual, humans err when they must scrutinize many possibilities to locate corner cases. Various forms of testing tools suffer, too. Stress testing is the common method for (indirectly) testing schedules, but often redundantly tests the same schedules while missing others. Recent tools (such as by Musuvathi et al.[16]) systematically test schedules for bugs, but developers lack resources to cover more than a tiny fraction of all the exponentially many schedules.

Determinism not as useful as commonly perceived. To address the challenges due to 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, thus bringing determinism to multithreading. This determinism has value for reliability; for instance, one testing execution now validates all future executions on the same input, and reproducing a concurrency error now requires only the input.

Meanwhile, little has been done to solve the challenges caused by too many schedules. The research community has assigned to nondeterminism more than its share of guilt but overlooked the main culprit—a rather quantitative cause, that multithreaded programs simply have too many schedules. Although determinism has value, that value is less than commonly perceived and 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 developers 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. Likewise, a single bit flip of a file should usually not cause a compression utility to crash. Unfortunately, determinism does not provide this stability and if naïvely implemented even undermines it.

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

This instability is counterintuitive at least, raising new reliability challenges; for instance, testing one input provides little assurance on very similar inputs, despite the differences in input not invalidating 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, even a different user name or if an error report lacks 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 even a correct sequential program may show different behaviors for small input changes across boundary conditions, but these conditions are typically infrequent, and the different behaviors are intended by developers. In contrast, the instability introduced by the system in Figure 1b is artificial and on all inputs.

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

In practice, to mitigate such problems, researchers augment determinism with other techniques. To support debug `printf`, some have proposed temporarily reverting to nondeterministic execution.[12] Moreover, DMP,[12] as well as CoreDet[4] and Kendo,[19] change schedules only if the inputs change low-level instructions being executed. Although better than mapping each input to an arbitrary schedule, these systems still allow small input perturbations to destabilize schedules unnecessarily when the perturbations change the low-level instructions executed (such as one extra load executed) we have observed in our experiments.[10] Our Tern and Peregrine systems and Liu et al.'s DThreads[14] built after Tern combine DMT with StableMT to frequently reuse schedules on a range of inputs for stability.

Reliability $\not\Rightarrow$ determinism. Determinism is a binary property; if an input maps to $n > 1$ schedules, executions on this input may be nondeterministic, however small n is. Yet a nondeterministic system with a small set of total schedules is easily made reliable. Consider an extreme case of the nondeterministic system in Figure

1d, mapping all inputs to at most two schedules. In it, developers are able to easily solve the needle-in-a-haystack challenges due to nondeterminism; for instance, to reproduce a field failure given an input, they can afford to search for one of only two schedules. As an analogy, a coin toss is nondeterministic, but humans have no problem understanding and doing it, as only two outcomes are possible.

Shrinking the Haystack

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

We investigated on a diverse set of popular 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 synchronization primitives (such as locks, semaphores, condition variables, and barriers). This produced two insights: First, for many programs, a range of many inputs share the same equivalent class of schedules. One schedule out of the class is enough to process the entire input range. Intuitively, an input often contains two types of data: metadata that controls the communication of the execution (such as number of threads to spawn) and computational data the threads compute on locally. A schedule requires the input metadata to have certain values but also allows the computational data to vary. That is, it can process any input that has the same metadata; for instance, consider `PBZip2`, which splits an input file among multiple threads, each compressing one file block. The communication, or which thread gets which file block, is independent of the thread-local compression. Under a typical setup (such as 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, another to otherwise compress any file, regardless of file data.

This loose coupling of inputs and schedules is not unique to `PBZip2`; many other programs also exhibit this property. Table 1 is a sample of our findings, including 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.

The second is that the overhead of enforcing a schedule on different inputs is low. The exponentially many schedules presumably allow the runtime system to react to various timing factors and select an efficient schedule. However, results from our StableMT systems invalidated the presumption. With carefully designed schedule representations, they incurred less than 15% overhead enforcing schedules on different inputs for most evaluated programs. This moderate overhead is worth the gains in reliability.

Leveraging the insights, we invented stable multithreading, or StableMT, a new multithreading approach that reuses each schedule on a range of inputs, mapping all inputs to a dramatically reduced set of schedules. Vastly shrinking the haystack at once. In addition, StableMT stabilizes program behaviors on inputs that map to the same schedule and minor program changes that do not affect the schedules, providing the robustness and stability expected by developers and users alike.

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

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

Testing. StableMT automatically increases the coverage of schedule testing tools, with coverage defined as the ratio of tested schedules over all schedules; for instance, consider `PBZip2` again, which needs only two schedules

Table 1. Constraints on inputs sharing the same equivalent class of schedules. For each program, one schedule out of the class is enough to process any input satisfying the constraints in the third column under typical setups (such as no system-call failures or signals).

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

for each different number of threads under typical setups. Testing 32 schedules covers from one to 16 threads. Given that `PBZip2` achieves peak performance when the number of threads is identical or close to the number of cores, and a typical machine has up to 16 cores, 32 tested schedules can cover most schedules executed in the field.

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

Analyzing and verifying programs. Static analysis can focus on the set of schedules enforced in the field to gain precision. Dynamic analysis enjoys the same benefits as testing. Model checking can check dramatically fewer schedules, mitigating the so-called "state explosion" problem.[9] Interactive theorem proving becomes easier, too, because verifiers must prove theorems on only the set of schedules enforced in the field.

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.

Caveats. StableMT is not for every multithreaded program. It works well with programs with schedules loosely coupled with inputs, but a program may decide to, say, spawn threads or invoke synchronizations based on intricate conditions involving many bits in the input. An example is the parallel `grep`-like utility `pfscan`, which searches for a keyword in a set of files using multiple threads, and for each match grabs a lock to increment a counter. A schedule computed on one set of files is unlikely to suit another. To increase the input range each schedule covers, developers can exclude the operations on this lock from the schedule using annotations.

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

Figure 2. Example program based on parallel compression utility `PBZip2`, which spawns `nthread` worker threads, splits a file among the threads, and compresses the file blocks in parallel.

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

```
// 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();
```

Building Stable Multithreading Systems

Although the vision of stable multithreading is appealing, its realization faces numerous challenges, including the following:

Computing schedules. How can StableMT systems compute the schedules to map inputs to? The schedules must be feasible in real-world situations so executions reusing them do not get stuck. They should also be highly reusable;

Enforcing schedules. How can StableMT systems enforce schedules deterministically and efficiently? "Deterministically" so executions that reuse a schedule cannot deviate, even when there are data races, and "efficiently" so overhead does not offset reliability benefits, a challenge decades old in the area of deterministic execution and replay; and

Handling threads. How can StableMT systems handle multithreaded server programs? They often run for a long time, reacting to each client request as it arrives, thus making their schedules very specific to a stream of requests and difficult to reuse.

We have been tackling these challenges since 2009 in building our two StableMT prototypes—TERN[10] and PEREGRINE[11]—that frequently reuse schedules with low overhead. Here, we describe our solutions, though they are by no means the only ones; for example, subsequent to TERN, other researchers built a system that stabilizes schedules for general multithreaded programs.[14]

Computing schedules. Crucial to implementing StableMT is how to compute the set of schedules for process-

ing inputs. At 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 the sake of reliability. One 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 is to compute schedules on the fly as a program runs, but the computations may be complex and their overhead prohibitively high.

Our systems compute schedules by recording them from past executions; the recorded schedules can then be reused on future inputs to stabilize program behaviors. TERN works like this: At runtime, it maintains a persistent cache of schedules recorded from past executions. When an input arrives, it 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, saving the schedule into the cache for future reuse.

The TERN approach to computing schedules has several benefits: First, by reusing schedules shown to work, it might avoid potential errors in unknown schedules, improving reliability. A real-world analogy is the natural human (and animal) tendency to follow familiar routes to avoid possible hazards along unknown routes. Migrating birds often follow, for example, fixed flyways. (The name TERN comes from the Arctic Tern, a bird species that migrates farthest among all animals.) Why are our multithreading systems unable to learn from them and reuse familiar schedules?

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, then deploy the cache together with their program, improving testing effectiveness and avoiding the overhead to record schedules on user machines. Moreover, they can run their favorite checking tools on the schedules to detect a variety of errors and choose to keep only the correct schedules in the cache.

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

A key challenge for TERN is to check that an input is compatible with a schedule before executing the input under the schedule. Otherwise, if it tries to enforce a schedule of, say, two threads on an input requiring four, the execution would not follow the schedule. This challenge turns out to be the most difficult one we had to solve in building TERN. Our solution leverages several advanced program-analysis techniques, including two new ones we invented; see Cui[10] and Cui[11] for details.

When recording a schedule, TERN tracks how the synchronizations in the schedule depend on the input, capturing these dependencies in a relaxed, quickly checkable set of constraints we call 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, TERN adds a constraint that `X` must be nonzero to the precondition. A precondition computed this way is sufficient but 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 through a simple program based on the aforementioned parallel compression utility `PBZip2` (see Figure 2). Its input includes all command-line arguments in `argv` and 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, it handles Pthread synchronizations.) However, the example also has a bug: Because it is missing `pthread_join` operations at line 7, the work list may be used by function `worker` after it is cleared at line 8, causing potential program crashes; this bug is based on a real bug in `PBZip2`.

We first illustrate how TERN records a schedule and its precondition. Suppose we run this example with two threads, and TERN records a schedule (see Figure 3) that avoids the use-after-free bug. (Other schedules are 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 includes the set of collected constraints. It then applies advanced program analyses to remove those that concern only local computations and have no effect on the schedule, including all constraints collected from function compress. The remaining ones simplify to `nthread = 2`, forming the precondition of the schedule. TERN stores the schedule and precondition into the schedule cache.

We now illustrate how TERN reuses a schedule. Suppose a user wants to compress a completely different

Figure 4. All input constraints collected for the schedule, each labeled with its line number in Figure 2. Those collected from function compression are later removed by TERN because they have no effect on the schedule; the remaining ones simplify to `nthread = 2`.

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

file also with two threads. TERN would detect that `nthread` satisfies `nthread` = 2, so it reuses the schedule in Figure 3 to compress the file, regardless of 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 the user runs the program again with four threads. TERN would detect that the input does not satisfy the precondition `nthread` = 2 so will record a new schedule and precondition.

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 (such as from 1.2X to 6X as slow as traditional multithreading[4]); synchronization schedules are much more efficient (such as average slowdown of only 16%[19]) because they are coarse grain but cannot make programs with data races deterministic (such as the second toy program discussed earlier, as well as many real-world multithreaded programs[15,23]). This determinism vs. performance challenge has been open for decades in the areas of deterministic execution and replay. As a result, TERN, our first StableMT system, enforces only synchronization schedules.

This is the challenge that prompted us to build PEREGRINE, our second StableMT system.[11] The PEREGRINE design insight is that although many programs have races, the races tend to occur only within small portions of an execution, with most of the execution still race-free. Intuitively, if a program is full of data races, most would have been caught during testing. We empirically analyzed the executions of seven real programs with races, finding, despite millions of memory accesses, up to only 10 data races per execution.

Since races are rare, PEREGRINE 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 grain as synchronization schedules so can also be reused frequently (see Figure 5).

How is PEREGRINE able to predict where data races might occur before an execution actually begins? One idea is to use static analysis to detect them at compile time. However, static race detectors are notoriously imprecise, as the tendency is for most of their reports to be false, not true data races. Scheduling many memory accesses in the false reports would slow execution significantly. PEREGRINE leverages the record-and-reuse approach in TERN to predict races; a recorded execution can effectively foretell what could happen for executions reusing the same schedule. Specifically, when recording a synchronization schedule, PEREGRINE records a detailed memory-access trace. From it, it detects data races that occurred (with respect to the schedule), adding the memory accesses involved in the races to the schedule. This hybrid schedule can be enforced efficiently and deterministically, solving the aforementioned open challenge of determinism vs. performance. To reuse the schedule on other inputs, PEREGRINE provides new precondition computation algorithms to guarantee 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 through a safe, efficient instrumentation framework called Loom we built in 2010.[21]

Figure 5. Hybrid schedule.

Circles represent synchronizations and triangles memory accesses. A synchronization schedule is efficient because it is coarse grain but not deterministic because data races could still cause executions to deviate from the schedule and fail. A memory access schedule is deterministic but slow because it is fine grain. A hybrid schedule combines the best of both by scheduling memory access for only the racy portion of an execution and for synchronizations otherwise.

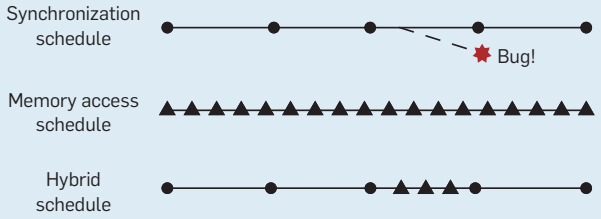
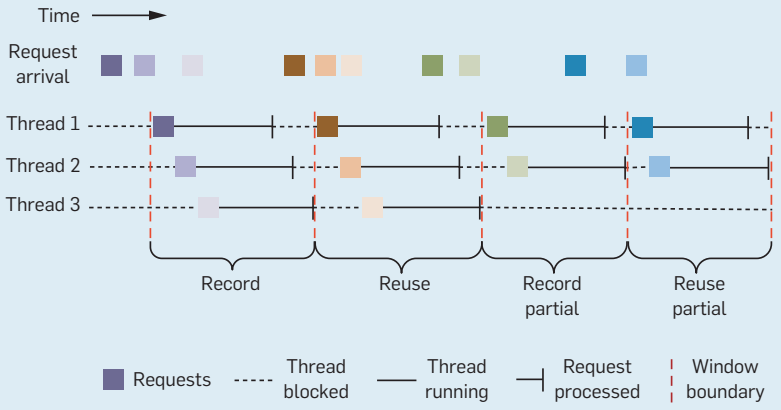


Figure 6. Recording and reusing schedules for a server program with three threads. The continuous execution stream is broken down into windows of requests, and PEREGRINE records and reuses schedules across windows.



Handling server programs. Server programs present three challenges for StableMT: First, they could run continuously, making their schedules effectively infinite and too specific to reuse; second, they often process inputs, or client requests, as soon as the requests arrive, with each request arriving at a random moment, causing a different schedule; and third, since requests do not arrive at the same time, PEREGRINE cannot check them against the precondition of a schedule up front.

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

Stable Multithreading for Better Program Analysis

StableMT can be applied in many ways to improve reliability. Here, we describe a program analysis framework we built atop PEREGRINE to analyze multithreaded programs, an open challenge in program analysis.

At its core is the trade-off 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 is imprecise (such as in issuing many false error reports). The cause is that static analysis must overapproximate the enormous number of schedules and thus may analyze a much larger set of schedules, including those impossible at runtime. Not surprisingly, static analysis can detect many "bugs" in the impossible schedules. Dynamic analysis, which runs code and analyzes the executions, precisely identifies bugs because it sees the code's precise runtime effects. However, it has poor coverage due to the exponentially many schedules.

Fortunately, StableMT shrinks the set of possible schedules, enabling a new program analysis approach reflecting the best of both static analysis and dynamic analysis (see Figure 7). It statically analyzes a parallel program over a small set of schedules at compile time, then dynamically enforces these schedules at runtime. By focusing on only a small set of schedules, StableMT greatly improves the precision of static analysis and reduces false error reports; by dynamically enforcing the analyzed schedules, StableMT guarantees high coverage. Dynamic analysis benefits, too, because StableMT greatly increases its coverage, as defined by the ratio of checked schedules over all schedules.

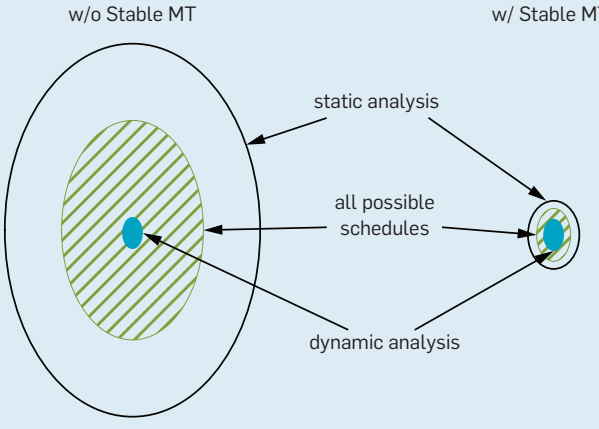
A key challenge in implementing this approach is how to statically analyze a program with respect to a schedule. A static tool typically invokes many analyses to compute the final results. A naïve method for modifying the tool for improved precision is to modify every analysis involved, though it would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

We thus created a new program-analysis framework and algorithms to specialize, or intelligently transform, a program according to a schedule. The resulting 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 significantly improved precision. In addition, our framework provides a precise "def-use" analysis that computes how values are defined and used in a program. Its results are much more precise than those of regular def-use analyses because it reports only the facts that may occur when the given schedule is enforced at runtime. This precision can be the foundation of many powerful tools, including race detectors.

Here, we illustrate the high-level idea of our framework recalling the example in Figure 2. Suppose a tool developer wants to build a static race detector that flags when different threads write the same shared memory location concurrently. Although different worker threads access disjoint file blocks, existing static analysis may be unable to determine this fact; for instance, since `nthread`, the number of threads, is determined at runtime, static analysis often has to approximate the

Figure 7. Program analysis with and without StableMT.

Without StableMT, static analysis tends to analyze many more schedules than all possible schedules; dynamic analysis tends to analyze a tiny fraction of all possible schedules. StableMT shrinks the set of schedules, improving both static analysis and dynamic analysis.



dynamic threads as one or two abstract thread instances. It may thus collapse different threads' accesses to distinct blocks as the same access, generating false race reports.

StableMT simplifies these problems. Suppose whenever `nthread` is 2, StableMT always enforces the schedule in Figure 3. Since the number of threads is fixed, our program-analysis framework rewrites the example program to replace `nthread` with 2. It then unrolls the loops and clones function `worker` to give each worker thread its own copy of `worker`, so distinguishing different worker threads becomes automatic.

Our framework also enables construction of many high-coverage and highly precise analyses; for instance, our static race detector found seven previously unknown harmful races in programs that had been extensively checked by previous tools. It generates extremely few false reports, none for 10 of 18 programs, a huge reduction compared to other static race detectors.

Evaluation

Here, we describe the main results of PEREGRINE, focusing on two evaluation questions:

Can it reuse schedules frequently? The higher the reuse rate, the more stable program behaviors become, and the more efficient PEREGRINE is; and

Can it enforce schedules efficiently? Low overhead is crucial for programs that reuse schedules frequently.

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

Stability. To evaluate PEREGRINE's stability, or how frequently it is able to reuse schedules, we compare the preconditions it computes to the best possible preconditions derived from manual inspection. Table 1 includes some of the manually derived preconditions; for nine of the 18 programs, the preconditions it computes are as good as or close to the best preconditions, allowing frequent reuses, while for the others, the preconditions are more restrictive.

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

Efficiency. The overhead of enforcing schedules is crucial for programs that frequently reuse schedules. Figure 8 shows this overhead for both TERN and PEREGRINE; each bar represents execution time, with TERN and PEREGRINE normalized to traditional multithreading, averaged over more than 500 runs; Figure 8 reports throughput (TPUT) and response time (RESP) for `Apache`.

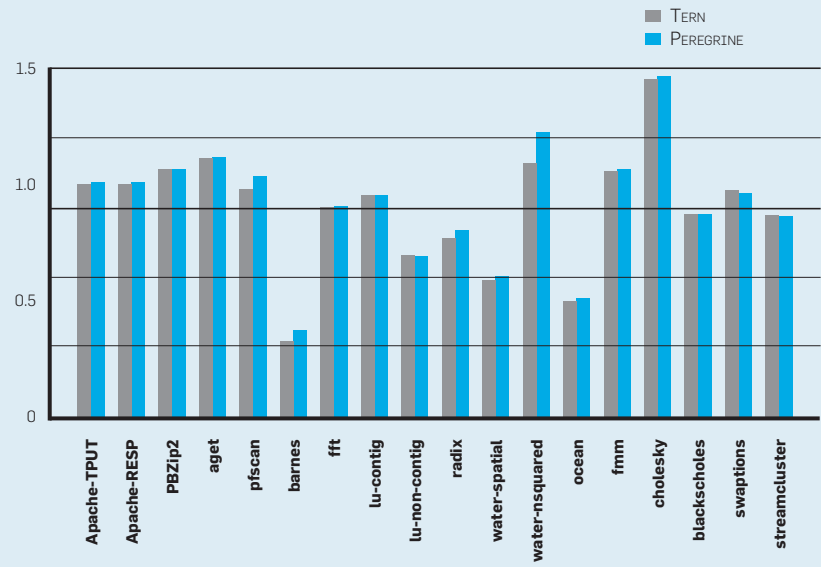
Two observations can be made about the figure: First, for most programs, overhead is less than 15%, demonstrating that StableMT can be efficient. For two programs—`water-nsquared` and `cholesky`—the overhead is relatively large because they do a large number of mutex operations within tight loops. However, overhead is still less than 50%, much less than the 1.1X–10X overhead of a prior DMT system.[4] Some programs enjoy a speedup, as TERN and PEREGRINE safely skip some blocking operations.[10,11] Second, PEREGRINE is only slightly slower than TERN, demonstrating full determinism can be efficient. (Recall TERN schedules only synchroniza-

Table 2. Schedule reuse rates under four workloads; the "Schedules" column lists number of schedules in the schedule cache.

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

Figure 8. Normalized execution time when reusing schedules.

A bar with value greater (or smaller) than 1 indicates slowdown (or speedup) compared to traditional multithreading. The overhead of reusing schedules is smaller than 15% of the total execution time of traditional multithreading for most programs and up to 50% for the other two programs. Five programs run faster because TERN or PEREGRINE safely skips some blocking operations.



tions, whereas PEREGRINE additionally schedules memory accesses to make data races deterministic.)

Conclusion

By conceiving, building, applying, and evaluating StableMT systems, we have demonstrated StableMT can stabilize program behaviors for better reliability, so it works efficiently and deterministically while greatly improving precision of static analysis. Moreover, it promises to help solve the grand challenge of making parallel programming easy. However, TERN and PEREGRINE are still research prototypes, not quite ready for general adoption. Moreover, the ideas we have explored are just the start of this direction in StableMT; the bulk of the work is ahead:

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

Application. At the application level, we have only scratched the surface. Improving program analysis is just one possible application; others include 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 could apply to other parallel programming methods (such as MPI, OpenMP, and Cilk-like tasks).

Conceptual. At the conceptual level, can we reinvent parallel programming to greatly reduce the set of schedules? For instance, a multithreading system might disallow schedules by default, allowing only the schedules' developers to explicitly write code to enable. Since developers are characterized by a range of skills, we may let only the best ones decide what schedules to use, reducing the likelihood of programming errors.

All are thus invited to explore with us this fertile and exciting direction in stable multithreading and reliable parallelism.

Acknowledgments

We thank all who helped with this work.[10,11,22,25] In particular, John Gallagher built the alias analysis in PEREGRINE, Chia-Che Tsai conducted the evaluation of TERN, and Huayang Guo helped evaluate PEREGRINE. We also thank Al Aho, Remzi Arpaci-Dusseau, Tom Bergan, Emery Berger, Luis Ceze, Xi Chen, Jason Flinn, Roxana Geambasu, Gail Kaiser, Jim Larus, Jinyang Li, Jiri Simsa, Ying Xu, and the anonymous reviewers for their many helpful comments. This work is supported in part by the National Science Foundation, including an NFS Career Award, Air Force Research Laboratory, an Air Force Office of Scientific Research Young Investigator Research Program Award, Office of Naval Research, and a Sloan Research Fellowship. C

References

1. Asanovic, K., Bodik, R., Demmel, J., Keaveny, T., Keutzer, K., Kubiatowicz, J., Morgan, N., Patterson, D., Sen, K., Wawrzynek, J., Wessel, D., and Yelick, K. A view of the parallel computing landscape. *Commun. ACM 52*, 10 (Oct. 2009), 56–67.
2. Aviram, A., Weng, S.-C., Hu, S., and Ford, B. Efficient system-enforced deterministic parallelism. *Commun. ACM 55*, 5 (May 2012), 111–119.
3. Ball, T. and Rajamani, S.K. Automatically validating temporal safety properties of interfaces. In *Proceedings of the Eighth International SPIN Workshop on Model Checking of Software* (Toronto, May 19–20, 2001),103–122.
4. Bergan, T., Anderson, O., Devietti, J., Ceze, L., and Grossman, D. CoreDet: A compiler and runtime system for deterministic multithreaded execution. In *Proceedings of the 15th International Conference on Architecture Support for Programming Languages and Operating Systems* (Pittsburgh, PA, Mar. 13–17). ACM Press, New York, 2010, 53–64.
5. Berger, E., Yang, T., Liu, T., Krishnan, D., and Novark, A. Grace: Safe and efficient concurrent programming. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications* (Orlando, FL, Oct. 25–29). ACM Press, New York, 2009, 81–96.
6. Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Henri-Gros, C., Kamsky, A., McPeak, S., and Engler, D. A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM 53*, 2 (Feb. 2010), 66–75.
7. Bocchino, Jr., R.L., Adve, V.S., Dig, D., Adve, S.V., Heumann, S., Komuravelli, R., Overbey, J., Simmons, P., Sung, H., and Vakilian, M. A type and effect system for deterministic parallel Java. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications* (Orlando, FL, Oct. 25–29). ACM Press, New York, 2009, 97–116.
8. Cantrill, B. and Bonwick, J. Real-world concurrency. *Commun. ACM 51*, 11 (Nov. 2008), 34–39.
9. Clarke, E., Grumberg, O., and Peled, D. *Model Checking.* MIT Press, Cambridge, MA, 1999.
10. Cui, H., Wu, J., Tsai, C.-C., and Yang, J. Stable deterministic multithreading through schedule memorization. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation*. USENIX Association, Berkeley, CA, 2010.
11. Cui, H., Wu, J., Gallagher, J., Guo, H., and Yang, J. Efficient deterministic multithreading through schedule relaxation. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*. ACM Press, New York, 2011, 337–351.
12. Devietti, J., Lucia, B., Ceze, L., and Oskin, M. DMP: Deterministic shared memory multiprocessing. In *Proceedings of the 14th International Conference on Architecture Support for Programming Languages and Operating Systems* (Washington, D.C., Mar. 7–11). ACM Press, New York, 2009, 85–96.
13. Lee, E.A. The problem with threads. *Computer 39*, 5 (May 2006), 33–42.
14. Liu, T., Curtsinger, C., and Berger, E.D. DTHREADS: Efficient deterministic multithreading. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles* (Cascais, Portugal, Oct. 23–26). ACM Press, New York, 2011, 327–336.
15. Lu, S., Park, S., Seo, E., and Zhou, Y. Learning from mistakes: A comprehensive study on real-world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architecture Support for Programming Languages and Operating Systems* (Seattle, Mar. 1–5). ACM Press, New York, 2008, 329–339.
16. Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P.A., and Neamtiu, I. Finding and reproducing Heisenbugs in concurrent programs. In *Proceedings of the Eighth Symposium on Operating Systems Design and Implementation* (San Diego, Dec. 8–10). USENIX Association, Berkeley, CA, 2008, 267–280.
17. Nethercote, N. and Seward, J. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation* (San Diego, June 11–13). ACM Press, New York, 2007, 89–100.
18. O'Hanlon, C. A conversation with John Hennessy and David Patterson. *Queue 4*, 10 (Dec. 2006), 14–22.
19. Olszewski, M., Ansel, J., and Amarasinghe, S. Kendo: Efficient deterministic multithreading in software. In *Proceedings of the 14th International Conference on Architecture Support for Programming Languages and Operating Systems* (Washington, D.C., Mar. 9–11). ACM Press, New York, 2009, 97–108.
20. Shao, Z. Certified software. *Commun. ACM 53*, 12 (Dec. 2010), 56–66.
21. Wu, J., Cui, H., and Yang, J. Bypassing races in live applications with execution filters. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation* (Vancouver, Canada, Oct. 4–6). USENIX Association, Berkeley, CA, 2010.
22. Wu, J., Tang, Y., Hu, G., Cui, H., and Yang, J. Sound and precise analysis of parallel programs through schedule specialization. In *Proceedings of the ACM SIGPLAN 2012 Conference on Programming Language Design and Implementation* (Beijing, June 11–16). ACM Press, New York, 2012, 205–216.
23. Xiong, W., Park, S., Zhang, J., Zhou, Y., and Ma, Z. Ad hoc synchronization considered harmful. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation* (Vancouver, Canada, Oct. 4–6). USENIX Association, Berkeley, CA, 2010.
24. Yang, J., Cui, A., Stolfo, S., and Sethumadhavan, S. Concurrency attacks. In *Proceedings of the Fourth USENIX Workshop on Hot Topics in Parallelism* (Berkeley, CA, June 7–8). USENIX Association, Berkeley, CA, 2012, 15.
25. Yang, J., Cui, H., and Wu, J. Determinism is overrated: What really makes multithreaded programs hard to get right and what can be done about it? In *Proceedings of the Fifth USENIX Workshop on Hot Topics in Parallelism* (San Jose, CA, June 24–25). USENIX Association, Berkeley, CA, 2013.

Junfeng Yang (http://www.cs.columbia.edu/~junfeng) is an associate professor and co-director of the Software Systems Lab in the Department of Computer Science at Columbia University, New York.

Heming Cui (http://www.cs.columbia.edu/~heming) is a Ph.D. student and a member of the Software Systems Lab in the Department of Computer Science at Columbia University, New York.

Jingyue Wu (http://www.cs.columbia.edu/~jingyue) is a Ph.D. student and a member of the Software Systems Lab in the Department of Computer Science at Columbia University, New York.

Gang Hu (http://www.cs.columbia.edu/~ganghu) is a Ph.D. student and a member of the Software Systems Lab in the Department of Computer Science at Columbia University, New York.

Yang Tang (http://ytang.com) is a Ph.D. student and a member of the Software Systems Lab in the Department of Computer Science at Columbia University, New York.