Stable Multithreading: A New Paradigm for Reliable and Secure Threads

Heming Cui

Submitted in partial fulfillment of the
requirements for the degree
of Doctor of Philosophy
in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2015

ABSTRACT

Stable Multithreading: A New Paradigm for Reliable and Secure Threads

Heming Cui

Multithreaded programs have become pervasive and critical due to the rise of multi-core hardware and accelerating computational demands. Unfortunately, despite decades of research and engineering effort, these programs remain notoriously difficult to get right, and they are plagued with harmful concurrency bugs that can cause wrong outputs, program crashes, security breaches, and so on.

Our research reveals that a root cause of this difficulty is that multithreaded programs have too many possible thread interleavings (or *schedules*) at runtime. Even given only a single input, a program may run into a great number of schedules, depending on factors such as hardware timing and OS scheduling. Considering all inputs, the number of schedules is even much greater. It is extremely challenging to understand, test, analyze, or verify this huge number of schedules for a multithreaded program and ensure that all these schedules are free of concurrency bugs. Thus, multithreaded programs are extremely difficult to get right.

To reduce the number of possible schedules for all inputs, we looked into the relation between inputs and schedules of real-world programs and made an exciting discovery: many programs need only a small set of schedules to efficiently process a wide range of inputs! Leveraging this discovery, we have proposed a new idea called *Stable Multithreading* (or *StableMT*) that reuses each schedule on a wide range of inputs, greatly reducing the number of possible schedules for all inputs. By addressing the root cause that makes multithreading difficult to get right, StableMT makes understanding, testing, analyzing, and verification of multithreaded programs much easier.

To realize StableMT, we have built three StableMT systems, Tern, Peregrine, and Parrot, with each addressing a distinct research challenge. Evaluation on a wide range of 108 popular multithreaded programs with Parrot, our latest StableMT system, shows that StableMT is simple,

fast, and deployable. PARROT's source code, entire benchmarks, and raw evaluation results are available at `github.com/columbia/smt-mc`.

To encourage deployment, we have applied StableMT to improve several reliability techniques, including: (1) making reproducing real-world concurrency bugs much easier; (2) greatly improving the precision of static program analysis, leading to the detection of several new harmful data races in heavily-tested programs; and (3) greatly increasing the coverage of model checking, a systematic testing technique, by many orders of magnitudes. StableMT has attracted the research community's interests, and some techniques and ideas in our StableMT systems have been leveraged by other researchers to compute a small set of schedules to cover all or most inputs for multithreaded programs.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I extend special thanks to my advisor, Prof. Junfeng Yang, who has supported and directed all aspects of my PhD career. I also sincerely thank Prof. Jason Nieh, Prof. Gail Kaiser, Prof. Roxana Geambasu, Prof. Jinyang Li, Prof. Randal E. Bryant, Prof. Garth A. Gibson, Prof. Simha Sethumadhavan, and Prof. Martha Kim for their valuable suggestions, advice, and collaboration during my PhD study. Parts of this thesis were joint work with Dr. Jingyue Wu, Dr. Jiri Simsa, Chia-che Tsai, John Gallagher, Huayang Guo, Yang Tang, Gang Hu, Yi-Hong Lin, Dr. Hao Li, Ben Blum, and Xinan Xu. I also extend thanks to Jonathan Bell, Adrian Tang, Ben Barg, Lingyuan He, and Yunfei Wang for their feedback and comments on my research.

To my family.

Chapter 1

Introduction

Multithreading has become pervasive and critical because of two major computing trends. First, due to the physical constraints on circuit speed, computing platforms are having more and more cores rather than faster and faster single-core. In order to harness the power of multi-core, developers are writing more and more multithreaded programs on these platforms. Second, the emerging cloud computing trend requires networking services (e.g., HTTP servers and database servers) to process more and more requests concurrently, which also pushes developers to write multithreaded programs. These two trends will continue, and multithreading will become increasingly pervasive and critical.

Unfortunately, despite decades of effort from both academia and industry, multithreaded programs remain notoriously difficult to get right, and these programs are plagued with harmful concurrency bugs that can cause wrong outputs, program crashes, security breaches, and so on. Our research reveals that a root cause of this difficulty is that multithreaded programs have too many possible thread interleavings (or *schedules*) at runtime. Even running on the same input, the concurrently running threads of a program may interleave in too many different ways, depending on factors such as hardware timing and OS scheduling. Considering all inputs, the number of possible schedules is even greater. Chapter 2 will quantify the number of possible schedules in multithreaded programs running in a traditional multithreading approach. It is extremely challenging to understand, test, analyze, or verify all these schedules in a multithreaded program and ensure that they are free of concurrency bugs. Therefore, a concurrency bug within an unchecked schedule can show up in production runs and lead to severe failures and vulnerabilities.

To make multithreading easier to get right, researchers have proposed an idea called *Deterministic Multithreading* (or *DMT*) [12, 21, 34, 66, 80] that always enforces the same schedule on the same input, greatly improving reliability for a multithreaded program on each input. However, as we will further analyze in Chapter 2, although DMT is useful, it is not as useful as commonly perceived. The reason is that a typical DMT system can enforce very different schedules on slightly different inputs, artificially reducing programs' robustness on input perturbations, and the number of possible schedules on all inputs remains enormous. Therefore, multithreaded programs remain very hard to understand, test, analyze, or verify.

To reduce the number of schedules for all inputs, we looked into the relation between inputs and schedules of real-world programs and made an exciting discovery: many programs need only a small set of schedules to efficiently process a wide range of inputs [122]! Leveraging this discovery, we have invented a new idea called *Stable Multithreading* (or *StableMT*) that reuses each schedule on a wide range of inputs, greatly reducing the huge number of possible schedules for all inputs, the root cause that makes multithreading difficult to get right. By reusing each schedule on as many as inputs, StableMT stabilizes program behaviors against small input perturbations. In short, by greatly reducing the number of possible schedules for all inputs, StableMT addresses at once the challenges of understanding, testing, analyzing, and verification of multithreaded programs and makes these programs much easier to get right. Actually, StableMT is complementary to DMT; a multithreading system can be both stable and deterministic. To fully illustrate the advantages of StableMT, Chapter 2 will discuss in detail the three types of multithreading approaches: traditional multithreading, DMT, and StableMT.

To realize StableMT, I have worked with Columbia and CMU researchers to build three StableMT systems, Tern, Peregrine, and Parrot, with each addressing a distinct research challenge. We identify and address these three challenges as follows.

Challenge 1: How to compute highly reusable schedules for different inputs? The more reusable a schedule is, the fewer schedules are needed for all inputs. However, finding highly reusable schedules is hard with existing static or dynamic techniques because statically computed schedules are not guaranteed to work at runtime due to the halting problem, and dynamically computing schedules may cause prohibitive overhead.

To address this challenge, our first StableMT system, Tern (Chapter 3), proposes a technique

called *schedule memoizatoin* that memoizes a set of past, working schedules, and then reuses these schedules on future inputs when possible. This technique is inspired by a real-world analogy that human and animals tend to migrate along past, familiar routes and avoid possible hazards in unknown ones. In order to find a schedule suitable for an input, TERN leverages a set of advanced program analysis techniques to compute input constraints (or *preconditions*) that match a schedule. Evaluation on a diverse set of popular programs shows that TERN can reuse a small set of schedules to process a wide range of inputs. For instance, just 100 schedules for the `Apache` web server can process 90.3% of a 4-day trace (122K requests) from the Columbia CS website.

Challenge 2: How to efficiently enforce schedules without deviation? This challenge has existed in the area of deterministic execution and replay for decades. Existing work typically enforces two types of schedules: a total order of shared memory accesses (or *mem-schedule*), and a total order of synchronization operations (or *sync-schedule*). Mem-schedules are fully deterministic even with data races, but they are several times slower than traditional multithreading. Sync-schedules incur only modest overhead because most code is not synchronization and thus can still run in parallel, but these schedules may deviate if there are data races. Overall, despite much research effort, people can only choose either full determinism or efficiency, but not both.

To address this challenge, our second StableMT (and also DMT) system, PEREGRINE (Chapter 4), takes advantage of an observation: although many programs have races, the races tend to occur only within minor portions of an execution, and the majority of the execution is still race-free. Therefore, we can enforce a sync-schedule in the race-free portions of an execution and resort to a mem-schedule only in the racy portions, combining both the efficiency of sync-schedules and the determinism of mem-schedules. PEREGRINE implements this form of hybrid-schedule with a new technique called *schedule relaxation*: it first records an execution trace of all executed instructions on a new input, and then relaxes the trace into a highly reusable hybrid-schedule. Evaluation on a diverse set of programs shows that PEREGRINE is deterministic and efficient, and it can frequently reuse schedules for half of the evaluated programs. PEREGRINE has been featured in sites such as `ACM TechNews`, `TG Daily`, and `Physorg`.

Challenge 3: How to make StableMT simple, fast, and deployable? In the last five years, StableMT has achieved promising advances and attracted the research community's interests. Several notable StableMT systems [8, 15, 31, 32, 66] have been built, including our TERN and

PEREGRINE systems. However, it remains an open challenge that whether StableMT can be made simple, fast, and deployable. Existing StableMT systems are either fairly difficult to deploy due to their high complexity (e.g., TERN and PEREGRINE require sophisticated program analysis), or they run into slow schedules that *serialize* parallel computation (e.g., we observed 30× slowdown when evaluating a notable system [66] in Chapter 5).

To address this challenge, our third StableMT system, PARROT (Chapter 5), presents a simple, deployable runtime that enforces a well-defined round-robin schedule for synchronization operations, vastly reducing the number of schedules. To address the serialization problem in StableMT, we have come up with an insight based on the famous 80-20 rule: most threads spend most execution time in only a few core computations, and we only need to make these core computations parallel. Accordingly, we create a new abstraction called *performance hints* for developers to annotate core computations. These hints, which just try to get to faster schedules that improve parallelism of core computations, are not real synchronization, and can be safely ignored without affecting correctness of a program. Evaluation on a wide range of 108 popular programs (e.g., Berkeley DB and MPlayer), roughly 10× more programs than any previous StableMT or DMT evaluation, and about 4× more programs than all previous evaluations combined, shows that, these hints are easy to add, and they make PARROT fast (merely 12.7% mean overhead on 24-core machines). To encourage StableMT deployment, we have made PARROT's source code, entire benchmarks, and raw evaluation results publicly available at `github.com/columbia/smt-mc`.

In addition to building StableMT systems, we have applied StableMT to improve the three following reliability techniques, demonstrating its advantages. First, we have shown that our StableMT systems consistently avoided or reproduced several real-world concurrency bugs across different executions [31, 32], while in a traditional Pthreads runtime these bugs showed up randomly. Second, we have applied StableMT to greatly improve the precision of program analysis and verification, leading to the detection of several new harmful data races in heavily-tested programs [115]. Third, we have quantitatively shown that StableMT can greatly increase the coverage of model checking [48, 101, 120], an advanced technique that systematically tests schedules for concurrency bugs, by many orders of magnitudes [33].

Due to its advantages for improving software reliability, StableMT has attracted the research community's interests. For instance, some techniques and ideas in our StableMT systems have been

leveraged by University of Washington researchers to compute a small set of schedules to cover all or most inputs for multithreaded programs [15].

The rest of the thesis is organized as follows. Chapter 2 presents the motivation and background of StableMT. Chapter 3 presents the Tern system, and our evaluation results from applying it to reproduce concurrency bugs. Chapter 4 describes the Peregrine system, and how much it can improve the precision of existing program analysis techniques as well as reproducing concurrency bugs. Chapter 5 introduces the Parrot system, and our advances in applying it to greatly improve the coverage of model checking. Chapter 6 concludes.

Chapter 2

Motivation and Background of StableMT

This chapter first points out a root cause that makes multithreading so difficult to get right (§2.1), and then introduces StableMT, our radical approach to address the root cause (§2.2). StableMT is not the only approach that aims to make multithreading easier to get right, and previously researchers have proposed a complementary approach called DMT, so this chapter also clarifies the differences between StableMT and DMT (§2.3).

2.1 Why is Multithreading So Hard to Get Right?

This section starts with preliminaries, and then points out a root cause that makes multithreading difficult to get right.

2.1.1 Preliminaries: Inputs, Schedules, and Buggy Schedules

To ease discussion, we use *input* to broadly refer to the data a program reads from its execution environment, including not only the data read from files and sockets, but also command line arguments, return values of external functions such as `gettimeofday`, and any external data that can affect program execution. We use *schedule* to broadly refer to the (partially or totally) ordered set of communication operations in a multithreaded execution, including synchronizations (e.g., `lock` and `unlock` operations) and shared memory accesses (e.g., `load` and `store` instructions to

shared memory). Of all the schedules, most run fine, but some trigger concurrency errors, causing program crashes, wrong outputs, security breaches, and other failures. Consider the toy program below:

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

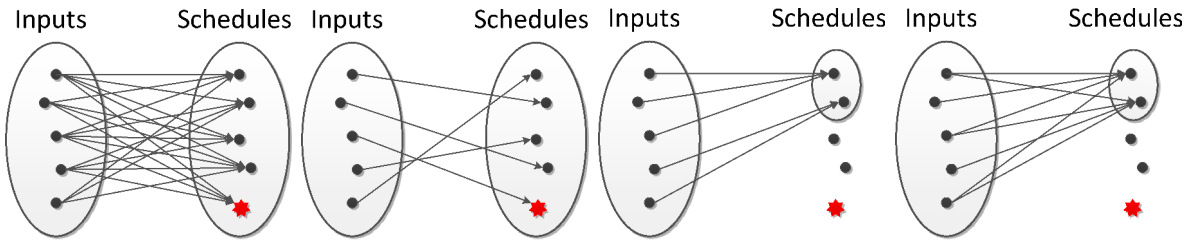
The schedule in which thread 2 gets the lock before thread 1 causes a dereference-of-NULL failure. Consider another example. The toy program below has data races on `balance`:

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

The schedule with the statements executed in the order shown corrupts `balance`. We call the schedules that trigger concurrency errors *buggy schedules*. Strictly speaking, the errors are in the programs, triggered by a combination of inputs and schedules. However, typical concurrency errors, such as most errors appeared in previous studies [71, 121], depend much more on the schedules than the inputs (e.g., once the schedule is fixed, the bug occurs for all inputs allowed by the schedule). Thus, recent research on testing multithreaded programs (e.g., [77]) is focused on effectively testing schedules to find the buggy ones.

2.1.2 Root Cause: Too Many Schedules for All Inputs

A typical multithreaded program has an enormous number of schedules. For a single input, the number of schedules is asymptotically exponential in the schedule length. For instance, given m threads each competing for a lock k times, each order of lock acquisitions forms a schedule, easily yielding $\frac{(mk)!}{(k!)^m} \geq (m!)^k$ total schedules—a number exponential in both m and k. Aggregated over all inputs, the number of schedules is even greater. Figure 2.1a depicts the traditional multithreading approach. Conceptually, traditional multithreading approaches (e.g., the Pthreads runtime) maintain a many-to-many mapping from inputs to schedules, where one input may execute under many schedules depending on factors such as hardware timing and OS scheduling, and many inputs may



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

Figure 2.1: Different multithreading approaches. Red stars represent buggy schedules. Traditional multithreading (2.1a) is a conceptual many-to-many mapping between inputs and schedules. DMT (2.1b) may map each input to an arbitrary schedule, reducing programs' robustness on input perturbations. StableMT (2.1c and 2.1d) reduces the total set of schedules for all inputs (represented by the shrunk ellipses), increasing robustness and improving reliability. StableMT is complementary to DMT: a StableMT system can be deterministic (2.1c) or nondeterministic (2.1d).

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.

Finding a few buggy schedules in these exponentially many schedules raises a series of "needle-in-a-haystack" challenges on understanding, testing, analyzing, and verification of multithreaded programs. For instance, when facing these excessive number of schedules, developers' understanding is prone to mistakes, and we have seen tons of concurrency bug reports sent to the developers' email lists. Various forms of testing tools also suffer. Stress testing is a common method for (indirectly) testing schedules, but it often redundantly tests the same schedules while missing others. To mitigate redundant testing effort, recent advanced testing tools (e.g., [48, 77, 100, 120]) can systematically test schedules, and these tools have included several remarkable reduction algorithms (e.g., [39, 48]) to avoid testing the same schedules and improve schedule coverage. Recent advanced program analysis and verification tools (e.g., [48]) also make notable attempts to increase the number of checked schedules based on these reduction algorithms. These systematic testing, analysis, and verification tools have effectively found new harmful concurrency bugs in real-world software. Unfortunately, despite these great effort, these tools still can not cover more than a tiny fraction of all the exponentially many schedules, and concurrency bugs within an unchecked

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_cb`	Matrix decomposition	Same number of threads, similar sizes of matrices and blocks.
`blackscholes`	Option pricing	Same number of threads, and the number of options is no less than the number of threads.
`swaptions`	Swaption pricing	Same number of threads, and the number of swaptions is no less than the number of threads.

Table 2.1: *Constraints on inputs sharing the same equivalent class of schedules.* For each program, one schedule out of the class suffices to process any input satisfying the constraints in the third column under typical setups (e.g., no system call failures or signals). We describe how to compute such constraints in Chapter 3.

schedule can show up in production runs and lead to severe failures and vulnerabilities. In short, the exponentially many schedules for all inputs is a root cause that makes a multithreaded program extremely difficult to get right.

2.2 Shrinking the Haystack with StableMT

To reduce the number of schedules and make multithreading easier to get right, 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 can 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 parallel compression utility `PBZip2`,

to parallel implementations of computation-intensive algorithms such as `fft`. These programs use diverse synchronization primitives such as mutex locks, semaphores, condition variables, and barriers. Our investigation reveals the following two insights.

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

This loose coupling of inputs and schedules is not unique to `PBZip2`; many other programs also exhibit this property. Table 2.1 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 SPLASH-2 and Princeton's PARSEC. (We will describe how to compute the constraints that a schedule places on the inputs in Chapter 3.)

Second, the overhead of enforcing a schedule on different inputs is often low. Presumably, the exponentially many schedules allow the runtime system to react to various timing factors and select an efficient schedule. However, results from the StableMT systems we built invalidated this presumption. With carefully designed schedule representations (Chapter 4), our systems incurred less than 15% overhead enforcing schedules on different inputs for most evaluated programs. Relevant systems (e.g., [8, 80]) also show that carefully enforcing schedules can achieve only moderate overhead. After all, considering the reliability benefits introduced by StableMT, we believe that this moderate overhead is worthwhile.

Leveraging these two insights, we have invented StableMT, a new multithreading approach that

reuses each schedule on a wide range of inputs, mapping all inputs to a dramatically reduced set of schedules. By vastly shrinking the haystack, it addresses all the "needle-in-a-haystack" challenges in understanding, testing, analyzing, and verification of multithreaded programs at once, making these programs much easier to get right.

2.2.1 Benefits

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

Understanding. Developers now only need to focus on understanding a much smaller set of schedules to ensure that they are free of concurrency bugs, which can greatly reduce their burden. For instance, because StableMT stabilizes program behaviors on a set of inputs that can share the same schedule, then after developers check that the program behavior on one input is correct, they are sure that all the other inputs within this set (e.g., inputs that control only thread-local computation) will run the same schedule and thus have the same correct behavior.

Testing. StableMT automatically increases the coverage of schedule testing tools, with coverage defined as the ratio of tested schedules over all schedules. For instance, consider `PBZip2` again which needs only two schedules for each different number of threads under typical setups. Testing 32 schedules effectively covers from 1 to 16 threads. Given that (1) `PBZip2` achieves peak performance when the number of threads is identical or close to the number of cores and (2) a typical machine has up to 16 cores, 32 tested schedules can practically cover most schedules executed in the field. Researchers have computed a small set of schedules to cover all or most inputs for multithreaded programs [15] by leveraging some techniques and ideas in our StableMT systems [31, 32].

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. 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. We will describe this benefit in detail in Chapter 3 and Chapter 4.

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 detail in Chapter 3.

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 now only need to check drastically fewer schedules, mitigating the so-called "state explosion" problem [28]. We have integrated our PARROT [33] system with an open source model checker called DBUG [101], and PARROT significantly increases the number of programs that DBUG can exhaust searching schedules under our evaluation settings. More details will be given in Chapter 5. Interactive theorem proving becomes much easier, too, because verifiers need to prove theorems only on the set of schedules enforced in the field. We will describe these benefits in detail in Chapter 4.

2.2.2 Caveats

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

2.3 Determinism: Not as Good as Commonly Perceived

A multithreaded program is *nondeterministic* because even with the same program and input, different executions may still run into different schedules and trigger different behaviors, depending on factors such as hardware timing and OS scheduling. For instance, the two toy programs in §2.1 do not always run into the bugs. Except for the schedules described, the other schedules lead to correct executions. Nondeterminism raises many challenges, especially in testing and debugging. Suppose an input can execute under n schedules. Testing $n-1$ schedules is not enough for complete reliability because the single untested schedule may still be buggy. An execution in the field may hit this untested schedule and fail. Debugging is challenging as well. 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.

To address the challenges raised by nondeterminism, researchers have dedicated much effort and built several DMT systems that force a multithreaded program to always run the same schedule on the same input. This determinism does have 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 exact input.

However, DMT only focuses on reducing the number schedules on each input, and it does not help much on reducing the excessive number of schedules for all inputs, the root cause that makes multithreading difficult to get right. We believe the research community has charged nondeterminism more than its share of the guilt and overlooked the main culprit—a rather quantitative cause that multithreaded programs simply have too many schedules. We argue that, although determinism has value, its value is smaller than commonly perceived: it is neither sufficient nor necessary for reliability.

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

To illustrate, consider the system depicted in Figure 2.1b that maps each input to an arbitrary schedule. This mapping is perfectly deterministic, but it destabilizes program behaviors on multiple inputs. A single bit flip may force a program to discard a correct schedule and adventure into a vastly different, buggy schedule. This *instability* problem raises new reliability challenges. For instance, testing one input provides little assurance on very similar inputs, despite that the differences in input do not invalidate the tested schedule. Debugging now requires every bit of the bug-inducing input, including not only the data a user typed, but also environment variables, shared libraries, etc. A different user name? Error report doesn't include credit card numbers? The bug may never be reproduced regardless of how many times developers retry because the schedule chosen by the deterministic system for the altered input happens to be correct. Besides inputs, naïvely implemented determinism can destabilize program behaviors on minor code changes, so

adding a debug `printf` causes the bug to deterministically disappear. Chapter 3 will analyze why this instability problem is inherent in existing DMT systems and present our evaluation results that confirms this problem. Because of this problem, when running with DMT, the number of possible schedules for all inputs remains enormous; therefore, a multithreaded program remains extremely difficult to understand, test, analyze, or verify.

In practice, to mitigate these problems, researchers have to augment determinism with other techniques. To support debug `printf`, some propose to temporarily revert to nondeterministic execution [34]. DMP [34], CoreDet [12], and Kendo [80] change schedules only if the inputs change low-level instructions executed. Although better than mapping each input to an arbitrary schedule, they still allow small input perturbations to destabilize schedules unnecessarily when the perturbations change the low-level instructions executed (e.g., one extra `load` executed), observed in our experiments in Chapter 3.

Reliability $\not\Rightarrow$ determinism. Determinism is a binary property: if an input maps to $n > 1$ schedules, executions on this input may be nondeterministic, however small n is. Yet, a nondeterministic system with a small set of total schedules can be made reliable easily. Consider an extreme case: the nondeterministic system depicted in Figure 2.1d that maps all inputs to at most two schedules. In this system, the challenges caused by nondeterminism are easy to solve. For instance, to reproduce a field failure given an input, developers can easily afford to search for one out of only two schedules. To offer an analogy, a coin toss is nondeterministic, but humans have no problem understanding and doing it because there are only two possible outcomes.

DMT is complementary to StableMT. 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 2.1c and Figure 2.1d depict two StableMT systems: the many-to-one mapping in Figure 2.1c is deterministic, while the many-to-few mapping in Figure 2.1d is nondeterministic. A many-to-few mapping improves performance because the runtime system can choose an efficient schedule out of a few for an input based on current timing factors, but it increases the effort 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. Our Tern, Peregrine, and Parrot systems and others' DThreads [66] built subsequently to Tern combine DMT with StableMT to frequently reuse

schedules on a wide range of inputs for stability. Chapter 3–5 will present the three systems we built.

2.4 Summary

A root cause that makes multithreading difficult to get right is that a program may run into exponentially many possible schedules for all inputs at runtime. This excessive number of possible schedules brings a series of "needle-in-a-haystack" challenges for reliability and security, including the understanding, testing, analyzing, and verification of multithreaded programs.

To address these challenges, we have proposed StableMT, a new approach that reuses each schedule on a wide range of inputs, greatly reducing the number of possible schedules for all inputs. By vastly shrinking the haystack, StableMT addresses all the "needle-in-a-haystack" challenges in understanding, testing, analyzing, and verification of multithreaded programs at once, making these programs much easier to get right.

StableMT is not the only technique that aims to reduce the number of possible schedules, and previously a technique called DMT has been proposed to reduce the number of schedules on each input. Although DMT is useful, we have explained that it is not as useful as commonly perceived, and that StableMT is better for reliability. StableMT is complementary to DMT, and a multithreading system can be both stable and deterministic.

Chapter 3

Computing Highly Reusable Schedules

This chapter describes Tern, our first StableMT system that addresses the first challenge on building StableMT: how to compute highly reusable schedules for different inputs? The more reusable a schedule is, the fewer schedules are needed for all inputs. We also aim to build Tern as a DMT system because determinism is especially useful in testing and debugging multithreaded programs. Building a multithreading system that is both stable and deterministic is another significant challenge: as we describe in Chapter 2, the instability problem in existing DMT systems destabilizes program behaviors on input perturbations, defeating the stability benefit brought by StableMT. Tern addresses these two challenges at once with a new technique called *schedule memoization*.

3.1 Introduction

Tern addresses two crucial research challenges. First, 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 pre-compute 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 may be high.

Second, how to combine DMT with StableMT? Existing DMT systems [12, 34, 80] constrain a multithreaded program to always use the same thread schedule for the same input, greatly increasing testing confidence and making bug reproduction much more easier. Unfortunately, these DMT systems may defeat the input stability benefit in StableMT: when scheduling the threads to process an input, existing DMT systems consider only current input and ignore previous similar inputs. This stateless design makes schedules over-dependent on inputs, so that a slight change to inputs may force a program to (ad)venture into a vastly different, potentially buggy schedule, defeating the key stability benefit of StableMT. This problem is defined as the instability problem in Chapter 2, and it has been confirmed by our results (§3.8.2.1) from an existing DMT system [12]. In fact, even with the same input, existing DMT systems may still force a program into different schedules for minor changes in the execution environment such as processor type and shared library. Thus, developers may no longer be able to reproduce bugs by running their program on the bug-inducing input because their machine may differ from the machine where the bug occurred. §3.2.1 will analyze in detail why this instability problem is inherent in existing DMT systems.

This chapter presents Tern, a schedule-centric, stateful multithreading system that is both stable and deterministic. It addresses the aforementioned two research challenges with a new idea called *schedule memoization* that memoizes past working schedules and reuses them for future inputs. Specifically, Tern maintains a cache of past schedules and the input constraints required to reuse these schedules. When an input arrives, Tern checks the input against the memoized constraints for a compatible schedule. If it finds one, it simply runs the program while enforcing this schedule. Otherwise, it runs the program to memoize a schedule and the input constraints of this schedule for future reuse. This schedule-centric approach maps as many as inputs that satisfy the input constraints to each schedule, greatly reducing the number of schedules required for all inputs, the central goal of StableMT. This stateful approach stabilizes program behaviors on input perturbations and avoids the instability problem. In sum, Tern's schedule memoization is the first approach that implements StableMT, and the first approach that combines StableMT and DMT, greatly reducing the number of schedules on all inputs as well as the number of schedules on each input (down to one) barring some limitations (§3.3.4).

Tern's schedule memoization approach has two major benefits on software reliability. First, by reusing schedules shown to work, Tern can avoid potential errors in unknown schedules. This

advantage is illustrated in Figure 2.1c in Chapter 2. A real-world analogy to schedule memoization 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." We thus name our system after the Arctic Tern, a bird species that migrates the farthest among all migrants [7]. Second, TERN makes schedules explicit, providing flexibility in deciding when to memoize certain schedules. For instance, TERN allows developers to populate a schedule cache offline, to avoid the overhead of doing so online. Moreover, TERN can check for errors (e.g., races) in schedules and memoize only the correct ones, thus avoiding the buggy schedules and amortizing the cost of checking for errors.

To make TERN practical, it must handle server programs which frequently use threads for performance. These programs present two challenges for TERN: (1) they often process client inputs (requests) as they arrive, thus suffering from *input timing nondeterminism*, which existing DMT systems do not handle and (2) they may run continuously, making their schedules effectively infinite and too specific to reuse.

TERN addresses these challenges using a simple idea called *windowing*. Our insight is that server programs tend to return to the same quiescent states. Thus, TERN splits the continuous request stream of a server into *windows* and lets the server quiesce in between, so that TERN can memoize and reuse schedules across windows. Within a window, it admits requests only at fixed schedule points, reducing timing nondeterminism.

We implemented TERN in Linux. It runs as "parasitic" user-space schedulers within the application's address space, overseeing the decisions of the OS scheduler and synchronization library. It memoizes and reuses synchronization orders as schedules to increase performance and reuse rates. It tracks input constraints using KLEE [24], a symbolic execution engine. Our implementation is software-only, works with general C/C++ programs using Pthreads, and requires no kernel modifications and only a few lines of modification to applications, thus simplifying deployment.

We evaluated TERN on a diverse set of 14 programs, including two popular server programs `Apache` [6] and `MySQL` [78], a parallel compression utility `PBZip2` [89], and 11 scientific programs in SPLASH-2 [103]. Our workload included a Columbia CS web trace and benchmarks used by `Apache` and `MySQL` developers. Our results show that

1. TERN is easy to use. For most programs, we modified only a few lines to make them work

with Tern.

2. Tern enforces stability across different inputs. In particular, it reused 100 schedules to process 90.3% of a 4-day Columbia CS web trace. Moreover, while an existing DMT system [12] made three concurrency bugs inconsistently occur or disappear depending on minor input changes, Tern's memoized schedules consistently avoided these bugs.

3. Tern has reasonable overhead. For nine out of fourteen evaluated programs, Tern has negligible overhead or improves performance; for the other programs, Tern has up to 39.1% overhead.

4. Tern makes threads deterministic. For twelve out of fourteen evaluated programs, the schedules Tern memoized can be deterministically reused barring the assumption discussed in §3.7.

Our main conceptual contributions are that we addressed the two research challenges on building a multithreading system that is both stable and deterministic with a new idea called schedule memoization. To make Tern practically support server programs that have input timing nondeterminism and infinite schedules, we proposed another new idea called windowing. Our engineering contributions include the Tern system and its evaluation on real programs. To the best of our knowledge, Tern is the first multithreading system that is both stable and deterministic, the first to mitigate input timing nondeterminism, and the first shown to work on programs as large, complex, and nondeterministic as `Apache` and `MySQL`.

This chapter is organized as follows. We first present a background (§3.2) and a high-level design of Tern (§3.3). We then describe Tern's interface (§3.4), schedule memoization for batch programs (§3.5), and windowing to extend Tern to server programs (§3.6). We then present refinements we made to optimize Tern (§3.7). Lastly, we show our experimental results (§3.8), discuss related work (§3.9), and summarize Tern (§3.10).

3.2 Background

This section first explains why the instability problem is inherent in existing DMT systems (§3.2.1), and then our choice of schedule representation in Tern (§3.2.2).

3.2.1 The Instability Problem in DMT

A DMT system is, conceptually, a function that maps an input I to a schedule S. The properties of this function are that the same I should map to the same S and that S is a feasible schedule for processing I. A stable DMT system such as Tern has an additional property: it maps similar inputs to the same schedule. Existing DMT systems, however, tend to map similar inputs to different schedules, thus suffering from the instability problem.

We argue that this problem is inherent in existing DMT systems because they are stateless. They must provide the same schedule for an input across different runs, using information only from the current run. To force threads to communicate (e.g., acquire locks or access shared memory) deterministically, existing DMT systems cannot rely on physical clocks. Instead, they maintain a logical clock per thread that ticks deterministically based on the code this thread has run. Moreover, threads may communicate only when their logical clocks have deterministic values (e.g., smallest across the logical clocks of all threads [80]). By induction, logical clocks make threads deterministic.

However, the problem with logical clocks is that for efficiency, they must tick at roughly the same rate to prevent a thread with a slower clock from starving others. Thus, existing DMT systems have to tie their logical clocks to low-level instructions executed (e.g., completed loads [80]). Consequently, a small change to the input or execution environment may alter a few instructions executed, in turn altering the logical clocks and subsequent thread communications. That is, a small change to the input or execution environment may cascade into a much different (e.g., correct vs. buggy) schedule.

3.2.2 Schedule Representation

Typical StableMT or DMT systems have considered two types of schedules: (1) a deterministic order of shared memory accesses [12, 34] and (2) a synchronization order (i.e., a total order of synchronization operations) [80]. The first type of schedules are fully deterministic even if there are races, but they are costly to enforce on commodity hardware (e.g., up to 10 times overhead [12]). The second type can be efficiently enforced (e.g., 16% overhead [80]) because most code is not synchronization code and can run in parallel; however, they are deterministic only for inputs that lead to race-free runs [80, 95].

Tern represents schedules as synchronization orders for efficiency. An additional benefit is

that synchronization orders can be reused more frequently than memory access orders (cf next subsection). Moreover, researchers have found that many concurrency errors are not data races, but atomicity and order violations [71]. These errors can be deterministically reproduced or avoided using only synchronization orders.

Although data races may still make runs which reuse schedules nondeterministic, TERN is less prone to this problem than existing DMT systems [80] because it has the flexibility to select schedules. If it detects a race in a memoized schedule, it can simply discard this schedule and memoize another. This selection task is often easy because most schedules are race-free. In rare cases, TERN may be unable to find a race-free schedule, resulting in nondeterministic runs. However, we argue that input nondeterminism cannot be fully eliminated anyway, so we may as well tolerate some scheduling nondeterminism, following the end-to-end argument.

3.3 High-level Design

Our design of TERN adheres to the following goals:

1. **Backward compatibility**. We design TERN for general multithreaded programs because of their dominance in parallel programs today and likely tomorrow. We design TERN to run in user-space and on commodity hardware to ease deployment.

2. **Stability**. We design TERN to bias multithreaded programs toward repeating their past, familiar schedules, instead of venturing into unfamiliar ones.

3. **Efficiency**. We design TERN to be efficient because it operates during the normal executions of programs, not replayed executions.

4. **Best-effort determinism**. We design TERN to make threads deterministic, but we sacrifice determinism when it contradicts the preceding goals.

The remaining of this section presents architecture (§3.3.1), workflow (§3.3.2), deployment scenarios (§3.3.3), and limitations (§3.3.4).

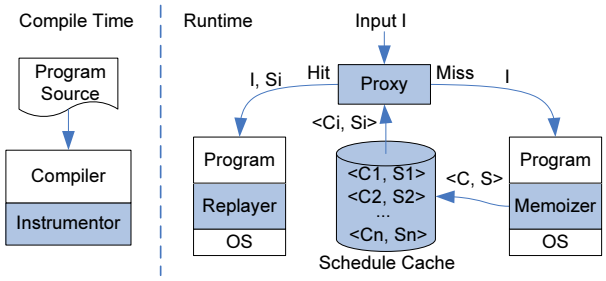


Figure 3.1: TERN *architecture.* Its components are shaded.

3.3.1 Architecture

Figure 3.1 shows the architecture of TERN and its five components: *instrumentor*, *schedule cache*, *proxy*, *replayer*, and *memoizer*. To use TERN, developers first annotates their application by marking the input data that may affect synchronization operations. They then compile their program with the *instrumentor*, which intercepts standard synchronization operations such as `pthread_mutex_lock()` so that at runtime TERN can control these operations. (We describe additional annotations and instrumentations that TERN needs in §3.4). The instrumentor runs as a plugin to LLVM [67], requiring no modifications to the compiler.

The *schedule cache* stores all memoized schedules and their input constraints. This cache can be marshalled to disk and read back upon program start, so that it need not be repopulated. Each memoized schedule is conceptually a tuple $\langle C, S \rangle$, where S is a synchronization order and C is the set of input constraints required to reuse S. (We explain the actual representation in §3.5.2).

At runtime, once an input I arrives, the *proxy* intercepts the input and queries the schedule cache for a constraint-schedule tuple $\langle C_i, S_i \rangle$ such that I satisfies C_i. On a cache hit, the proxy lets the *replayer* run the program on input I and enforce schedule S_i. On a cache miss, it lets the *memoizer* run the program on input I to memoize a new schedule.

During a memoization run, the memoizer records all synchronization operations into a schedule S. It also computes C, the input constraints for reusing S, via symbolic execution [24]. The basic idea of symbolic execution is to track the outcomes of branches that observe symbolic data, in our case, the data marked by developers as affecting synchronizations. Once the memoization run ends, the set of branch outcomes we collected describes the input constraints needed to reuse the memoized schedule.

```
1 :  main(int argc, char *argv[]) {
2 :    int i, nthread = argv[1], nblock = argv[2];
3 :    symbolic(&nthread, sizeof(int)); // mark input data
4 :    symbolic(&nblock, sizeof(int)); // that affects schedules
5 :    for(i=0; i<nthread; ++i)
6 :      pthread_create(worker); // create worker threads
7 :    for(i=0; i<nblock; ++i) {
8 :      block = read_block(i); // read i'th file block
9 :      worklist.add(block); // add block to work list
10:    }
11:  }
12:  worker() { // worker threads for compressing file blocks
13:    for(;;) {
14:      block = worklist.get(); // get a file block from work list
15:      compress(block);
16:    }
17:  }
```

Figure 3.2: Simplified `PBZip2` code.

For determinism, the memoizer can optionally check a memoization run for data races. If it detects no races, it simply stores $\langle C, S \rangle$ into the schedule cache. Otherwise, it can discard the memoized schedule and rerun the program with a different scheduling algorithm to memoize another schedule.

The proxy performs an additional task for server programs to reduce input timing nondeterminism and to reuse schedules for these programs. Specifically, it buffers the requests of a server into a window with a fixed size. When the window becomes full, or remains partial for a predefined timeout, Tern runs the server to process the window as if the server were a batch program. It then lets the server quiesce before moving to the next window to avoid interference between windows.

3.3.2 Workflow and An Example

We illustrate how Tern works using `PBZip2` as an example. Figure 3.2 shows the simplified code of `PBZip2`. Variables `nthread` and `nblock` affect synchronizations, so developers mark them by calling the Tern-provided method `symbolic()` (line 3 and line 4). This code spawns `nthread` worker

```
// main          worker 1          worker 2
9: worklist.add();
                 14: worklist.get();
9: worklist.add();
                                   14: worklist.get();
```

Figure 3.3: Synchronization order of a `PBZip2` run.

```
5: 0 < nthread ? true
5: 1 < nthread ? true
5: 2 < nthread ? false
7: 0 < nblock ? true
7: 1 < nblock ? true
7: 2 < nblock ? false
```

Figure 3.4: Input constraints of a `PBZip2` run.

threads, splits the file into `nblock` blocks, and compresses them in parallel by calling `compress()`. To coordinate the worker threads, it uses a synchronized work list. (Note TERN tracks low-level synchronizations such as pthread primitives; we use a work list here only for clarity.)

Suppose we run `PBZip2` with two threads on a two-block file. Suppose the schedule cache is empty and TERN runs the memoizer to memoize a new schedule. As `PBZip2` runs, TERN controls and records the synchronization operations (line 9 and line 14). It also tracks the outcomes of branch statements that observe symbolic data (line 5 and line 7). At the end of the run, TERN records a schedule as shown in Figure 3.3. It also collects constraints as shown in Figure 3.4, which simplify to $nthread = 2 \wedge nblock = 2$.[1] It stores the schedule and the input constraints into the schedule cache.

If we run `PBZip2` again with two threads on a different two-block file, TERN will check if variable `nthread` and `nblock` satisfy any set of constraints in the schedule cache. In this case, TERN will succeed. It will then reuse the schedule (Figure 3.3) to compress the file, even though the file data may differ completely.

[1] Although in this example the constraints are collected from one thread, TERN can actually collect constraints from multiple threads.

3.3.3 Deployment Scenarios

We anticipate three ways users may deploy Tern to make their programs stable and deterministic.

Schedule-carrying code. Developers pre-populate a cache of correct, representative schedules on typical workloads, then ship their program with the cache hardwired and marked read-only.

Online memoization. Users can turn on memoization at their local sites so that Tern can memoize schedules as the programs run on real inputs.

Shadow memoization. Since tracking input constraints is slow, users can configure Tern to memoize schedules asynchronously. Specifically, for an input that misses the schedule cache, the proxy runs the program as is, while forwarding a copy of the input to the memoizer.

Each deployment mode has pros and cons. The first mode makes a program stable and deterministic across different sites, but may react poorly to site-specific workloads. The second mode updates the schedule cache based on site-specific workloads, but may be slow because memoization runs tend to be slow. The last approach avoids the slowdown, but allows a program to run nondeterministically when an input misses the schedule cache. For server programs with high performance requirements, we recommend the first and the third modes.

3.3.4 Limitations

Determinism. Tern aims for best-effort determinism for reasons discussed in §3.2.2. If Tern is unable to find a race-free schedule for an input, the run may be nondeterministic. We foresee several strategies to handle this corner case while adhering to the other goals of Tern. For instance, we can instrument the program to fix the detected races or apply one of the existing DMT algorithms to resolve the races deterministically. The advantage of combining these techniques with Tern is that we apply these expensive techniques only to a small portion of schedules, and use Tern to efficiently handle the common case. We leave these ideas for future work.

Applicability. We anticipate our approach will work well for many programs/workloads as long as (1) they can benefit from determinism and stability, (2) their constraints can be tracked by Tern, (3) their schedules can be frequently reused, and (4) if windowing is needed, their inputs can be buffered. For programs/workloads that violate these assumptions, Tern may work poorly. These programs/workloads may include parallel simulators that require nondeterminism for statistical results, GUI programs that cannot buffer user actions for latency reasons, randomly generated

Annotations	Inserted by	Semantics
symbolic(*data*, *len*)	Developer	Marks data that may affect schedules. The memoizer tracks constraints on this data. The replayer checks this data against the memoized constraints.
begin_task() end_task()	Developer	Mark the beginning and end of a logical task. Used to divide the executions of threads in servers. (§3.6).
lock_wrapper(*l*) unlock_wrapper(*l*)	Developer or TERN	Synch wrappers used by the memoizer for memoizing schedules, and by the replayer for reusing schedules.
before_blocking() after_blocking()	TERN	Inserted before and after blocking system calls for the memoizer to log the order of these calls, and for the replayer to enforce the same order of these calls.

Table 3.1: TERN *interface.* Some annotations are inserted by developers, and others are inserted by the instrumentor, indicated by Column **Inserted By**. Both the memoizer and the replayer use this interface, but they implement this interface differently (§3.5).

workloads that prevent schedule reuses, and programs whose schedules depend on floating point inputs (which cannot be tracked by TERN's underlying symbolic execution engine).

Manual annotation. TERN requires manual annotations. However, this annotation overhead tends to be small. (See §3.7.4 for how TERN reduces this overhead and §3.8.1 for an evaluation of this overhead). This overhead may be further reduced using simple static analysis.

3.4 Interface

Table 3.1 shows TERN's annotation interface which developers and the instrumentor use to annotate multithreaded programs. The annotations fall into four categories: (1) `symbolic()` for marking data that may affect schedules; (2) task boundary annotations for marking the beginning and end of logical tasks, in case threads get reused for different logical tasks (§3.6); (3) wrappers to synchronization operations (more examples in the next paragraph); and (4) hook functions inserted around blocking system calls, which TERN memoizes because blocking systems calls are natural scheduling points.

TERN hooks 28 Pthreads operations (e.g., `pthread_mutex_lock()`, `pthread_create()`, and `pthread_cond_wait()`). It also handles common atomic operations such as `atomic_dec()` and

`atomic_inc()`. It hooks eight blocking system calls (e.g., `sleep()`, `accept()`, `recv()`, `select()`, and `read()`). These hooks are sufficient to run the programs evaluated, and we can easily add more.

Developers manually insert annotations in the first two categories. They also annotate custom synchronizations (e.g., custom spin locks). TERN's instrumentor automatically hooks standard synchronization and blocking system calls. These annotations allow TERN's memoizer and replayer to run as "parasitic" user-space schedulers that oversee the scheduling decisions of the OS and synchronization library, requiring no modifications to either.

3.5 Schedule Memoization

This section presents the idea of schedule memoization in the context of batch programs. We describe how TERN memoizes schedules (§3.5.1), tracks input constraints (§3.5.2), merges a schedule into the schedule cache (§3.5.3), and reuses schedules (§3.5.4).

3.5.1 Memoizing Schedules

To memoize schedules, the memoizer controls and logs synchronization operations. By default, it uses a simple round-robin (RR) algorithm that forces each thread to do synchronizations in turn. One advantage of this algorithm is that independent sites may memoize the same schedules, making program behaviors deterministic and stable across sites.

The memoizer implements this algorithm by implementing the wrappers in Table 3.1. Figure 3.5 shows the wrappers to `pthread_mutex_lock()` and `pthread_mutex_unlock()`. The memoizer maintains a queue of active threads. Only the thread at the head of the queue "has the turn" (line 4 and 14). Once the thread is done with the operation, it gives up the turn by moving itself to the tail of the queue (line 7 and 18).

We explain three subtleties of the code. First, to avoid the deadlock scenario when the head of the queue attempts to grab an unavailable mutex, we call the non-blocking lock operation instead of the blocking one (line 5). If the mutex is not available, the thread gives up its turn and waits on a TERN-maintained wait queue (line 10). TERN uses its own wait queues to avoid nondeterministic wakeup orders in pthread library. Second, we log synchronizations (line 6 and line 17) only when

```
1 :  queue_t activeq, waitq[N];
2 :  pthread_mutex_lock_wrapper(pthread_mutex_t *mutex) {
3 :   retry:
4 :    while(self()!=activeq.head); // wait for our turn
5 :    if(!phtread_mutex_trylock(mutex)) { // mutex acquired
6 :      append(schedule, self()); // add tid to schedule
7 :      move(self(), activeq.tail); // give turn to next thread
8 :      return;
9 :    }
10:    move(self(), waitq[mutex].tail); // deterministically wait
11:    goto retry; // wait for our turn again
12:  }
13:  pthread_mutex_unlock_wrapper(pthread_mutex_t *mutex) {
14:    while(self()!=activeq.head); // wait for our turn
15:    pthread_mutex_unlock(mutex); // mutex released
16:    wake_up(waitq[mutex].head); // deterministically wake up
17:    append(schedule, self()); // add tid to schedule
18:    move(self(), activeq.tail); // give turn to next thread
19:  }
```

Figure 3.5: *The memoizer's round-robin scheduling algorithm.*

the thread has the turn, so that the log faithfully reflects the actual order of synchronizations. Lastly, we maintain our internal thread IDs to avoid nondeterminism in the OS thread IDs across runs. Function `self()` returns this internal ID for the current thread (line 6 and line 17).

The memoizer allows a thread to break out of the round-robin when the thread has waited for its turn for over a second. The rationale is that if a thread has waited too long, the current schedule will likely perform poorly in reuse runs. However, such timeouts do not affect nondeterminism, because the memoizer still logs the order of the occurred operations and the replayer simply enforces the same order. In our experiments, we never observed such timeouts because most threads synchronize or call blocking system calls frequently.

Unlike previous DMT systems, Tern has the flexibility to select scheduling algorithms. In addition to the RR algorithm, it implements a first-come first-served (FCFS) algorithm that lets threads run as is. If the memoizer detects a race using RR, it can restart the run and switch

to FCFS. Implementing FCFS requires only minor modifications to the algorithm presented in Figure 3.5. Specifically, we replace line 4 and line 14 with a lock operation; line 7, line 10, and line 18 with an unlock operation; and line 16 a NOP.

In addition to synchronizations, the memoizer includes the hooks around blocking system calls (§3.4) in the schedule it memoizes because blocking system calls are natural scheduling points. However, the replayer will only opportunistically replay these hooks when reusing a schedule because the returns from blocking system calls are driven by the program's environment.

3.5.2 Tracking Input Constraints

Given the symbolic data marked by developers, the memoizer tracks the constraints on this data by tracking (1) what data is derived from the symbolic data and (2) the outcomes of the branch statements that observe this symbolic and derived data. At the end of this memoization run, the set of branch outcomes together describe the constraints to place on the symbolic data required to reuse the memoized schedule. That is, if an input satisfies these constraints, we can re-run the program in the same way as the memoization run. The constraints collected this way may be over-constraining if developers annotate too much data as symbolic. We describe a technique to address this problem in §3.7.4.

Tern leverages klee [24], an open-source symbolic execution engine to track input constraints. To adapt klee to Tern, we made two key modifications. First, klee works only with sequential programs, thus we extended it to support threads. Specifically, we modified klee to spawn a new klee instance for each new thread. At the end of the run, we unify the constraints collected from each thread as the input constraints of the schedule. Second, we simplified klee to only collect constraints without solving them, because unlike klee, Tern need not explore different execution paths.

3.5.3 Merging Schedules into the Schedule Cache

Once Tern memoized a schedule S and its constraints C, Tern stores the tuple into the schedule cache. Although the schedule cache is conceptually a set of $\langle C, S \rangle$ tuples, its actual structure is a decision tree because a program may incrementally read inputs from its environment, calling `symbolic()` multiple times. For example, the code in Figure 3.2 calls `symbolic()` twice.

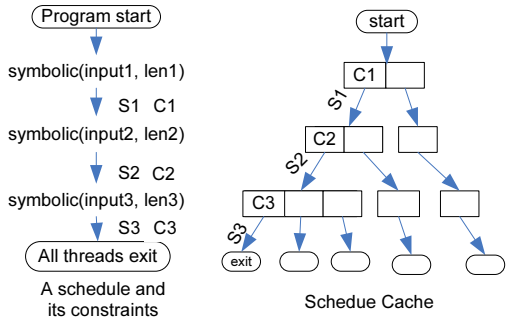


Figure 3.6: *Decision tree of* TERN*'s schedule cache.*

Figure 3.6 illustrates how TERN constructs the decision tree of the schedule cache. Given a $\langle C, S \rangle$ tuple, TERN breaks it down to sub-tuples $\langle C_i, S_i \rangle$ separated by `symbolic()` calls, where S_i contains the synchronization operations logged and C_i contains the constraints collected between the i^{th} and $(i+1)^{th}$ `symbolic()` calls. It then merges the sub-tuples into the i^{th} level of the decision tree.

TERN avoids merging redundant tuples into the cache. That is, if the cache contains a tuple with less restrictive constraints that the tuple being merged, TERN simply discards the new tuple. Note that the tuples may overlap (i.e., one input satisfies more than one set of constraints), and TERN simply returns the first match if there are multiple matches.

To speed up cache lookup, TERN sorts all $\langle C_i, S_i \rangle$ tuples within the same decision node based on their *reuse rates*, defined as the number of successful reuses of S_i over the number of inputs that have satisfied C_i. Reusing a schedule may fail even if the input satisfies the schedule's input constraints (cf next subsection). However, by sorting the tuples based on reuse rates, we automatically prefer good schedules over bad ones that have many failed reuse attempts. To bound the size of the schedule cache, TERN can throw away bad schedules based on reuse rates. However, we have not found the need to do so because the schedule cache is often small.

3.5.4 Reusing Schedules

To reuse a schedule, TERN must check that the input satisfies the input constraints of the schedule. To do so, it maintains an iterator to the decision tree of the schedule cache. The iterator starts from the root. As the program runs and calls `symbolic()`, TERN moves the iterator down the tree.

```
pthread_mutex_lock_wrapper(mutex) {
  down(sem[self()]); // wait for our turn
  pthread_mutex_lock(mutex);
  next = shift schedule; // find next thread in schedule
  up(sem[next]); // wake up next thread
}
```

Figure 3.7: *Pseudo code of the replayer.*

It checks if the data passed into a `symbolic()` call satisfies any set of constraints stored at the corresponding decision tree node and, if so, enforces the corresponding schedule.

The performance of the replayer is crucial because it runs during a program's normal executions. To efficiently enforce a synchronization order, the replayer uses a technique we call *semaphore relay*. Specifically, the replayer assigns each thread a semaphore. Before doing a synchronization operation, a thread has to wait on its semaphore for its turn. Once it is done with the operation, it passes the turn to the next thread in the schedule by signaling the semaphore of the next thread. Compared to an approach using locks or condition variables, semaphore relay avoids unnecessary lock contentions. Figure 3.7 illustrates semaphore relay using the replayer's `pthread_mutex_lock()` wrapper.

We note several subtleties of the pseudo code in Figure 3.7. First, we do not use non-blocking lock operations (line 3) as in Figure 3.5 because the memoizer only logs successful lock acquisitions. Second, the replayer maintains internal thread IDs the same way as the memoizer to avoid mismatches. Lastly, the `down()` (line 2) is actually a timed wait (with a default 0.1ms timeout), so that a thread can break out of a schedule when the dynamic load mismatches the schedule's assumptions. Note that these timeouts merely cause delays and do not affect correctness. They rarely occurred in our experiments.

3.6 Windowing

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

TERN addresses these challenges using a simple idea called *windowing*. Our insight is that server programs tend to return to the same quiescent states. Thus, instead of processing requests as they arrive, TERN breaks a continuous request stream down to windows of requests. Within each window, it admits requests only at fixed points in the current schedule. If no requests arrive at an admission point for a predefined timeout, TERN simply proceeds with the partial window. While a window is running, TERN buffers newly arrived requests so that they do not interfere with the running window. With this approach, TERN can memoize and reuse schedules across (possibly partial) windows. The cost of windowing is that it may reduce concurrency and degrade server throughput and speed. However, our experiments show that this cost is reasonable and justified by the gain in determinism and stability.

To buffer requests, TERN needs to know when a server receives a request and when it is done processing the request. Inferring these task boundaries based on thread creation and exit is unreliable because server programs frequently use thread pools. Thus, TERN currently lets developers annotate these boundaries using `begin_task()` and `end_task()`. Manually locating task boundaries is often easy: a request tends to begin after an `accept()` of a client connection and ends after the server sends out a reply.

Exposing hidden states. The assumption of windowing is that a server program returns to the same state when it quiesces. However, in practice, server states evolve over time. For instance, when Apache first serves a page, it may load the page from disk and cache it in memory. When this page is requested again, Apache can serve it directly from its cache.

These state changes may affect schedules. In the example above, Apache will perform different synchronizations for the two runs. Thus, for TERN to accurately select a schedule to reuse, it must know the hidden states that affect schedules. Currently TERN lets developers annotate such hidden states using `symbolic()`. Doing so is often straightforward. For instance, we inserted a `symbolic()` call to mark the return of Apache's `cache_find()` as symbolic.

Exposing hidden states may not always be easy. We thus created a technique to tolerate missed `symbolic()` annotations. The basic idea is to store backup schedules under the same set of input constraints to tolerate annotation inaccuracy. For instance, suppose a `symbolic()` had not been missed, TERN would have memoized two different constraint-schedule tuples $\langle C_1, S_1 \rangle$ and $\langle C_2, S_2 \rangle$. However, because of the missed annotation, TERN missed the corresponding constraints, wrongly

```
// T1        // T2
++x;
lock(l1);
                lock(l2);
                ++x;
```

Figure 3.8: *A conventional race, not a schedule race.*

```
// T1        // T2
lock(l1);
a[i]++;     lock(l2);
                a[j]−−;
unlock(l1);
                unlock(l2);
```

Figure 3.9: *A symbolic race that occurs only when* $i = j$.

collapsing C_1 and C_2 into the same set C. Now the two original tuples become $\langle C, S_1 \rangle$ and $\langle C, S_2 \rangle$, which appear redundant. Instead of discarding one of these seemingly redundant schedules, Tern will store both schedules with the same set of constraints. To select between these schedules, Tern can select the one with higher reuse rate, which likely matches the hidden state of the program.

3.7 Refinements

This section describes four refinements we made, one for determinism (§3.7.1) and three for speed (§3.7.2-§3.7.4).

3.7.1 Detecting Data Races

As discussed in §3.2.2, if a memoized schedule allows data races, runs reusing this schedule may become nondeterministic. Thus, for determinism, we would like to detect races in memoized schedules and discard them from the schedule cache. A general race detector would flag too many races for Tern because it detects conventional races with respect to the original synchronization constraints of the program, whereas we want to detect races with respect to the order constraints of a schedule [95] (call them *schedule races*). Figure 3.8 shows a conventional race, but not a schedule race because the synchronization order shown "kills" the race.

Thus, we built a simple race detector to detect schedule races. It runs with the memoizer and is happens-before based. It considers one memory access happens before another with respect to the synchronization order the memoizer records. Sometimes a pair of instructions may appear to be a race, when in fact their relative order does not alter a run. For instance, a write-write race is

benign if both instructions write the same value. Similarly, a read-write race is benign if the value written by one instruction does not affect the value read by another. Our race detector prunes these benign races.

Our detector also flags *symbolic races*, the races that are data-dependent on inputs. Figure 3.9 shows an example. Both variables i and j are inputs, and the race occurs only when $i = j$. The risk of a symbolic races is that it may be absent in a memoization run and thus skip detection, but show up nondeterministically in a reuse run. To detect symbolic races, our race detector queries the underlying symbolic execution engine for pointer equality. For example, to detect the race in Figure 3.9, it would query the underlying symbolic execution engine for the satisfiability of $\&a[i] = \&a[j]$. It flags a symbolic race if this constraint is satisfiable. Once a symbolic race is flagged, TERN adds additional input constraints to ensure that the race does not occur in reuse runs. For Figure 3.9, we would add $\&a[i] \neq \&a[j]$, which simplifies to $i \neq j$.

Our race detector can detect all schedule races in a memoization run. It can also detect all symbolic races if developers correctly annotate all data that affect synchronization operations and memory locations accessed. If this assumption holds and our race detector reports no races in a memoization run, TERN ensures that the memoized schedule can be deterministically reused.

3.7.2 Skipping Unnecessary Synchronizations

When reusing a schedule, TERN enforces a total synchronization order according to the schedule. These TERN-enforced execution order constraints are more stringent than the constraints enforced by the original synchronizations in the program. Thus, for speed, TERN can actually skip these unnecessary synchronizations. In our current implementation, we skip `sleep()`, `usleep()`, and `pthread_barrier_wait()` because they are frequently used. We found that this optimization was quite effective and even made programs run faster than nondeterministic execution (§3.8.3).

3.7.3 Simplifying Constraints

To reuse a schedule, TERN must check if the current input satisfies the constraints of the schedule. The overhead of this check depends on the number of constraints, yet the set of constraints TERN collects may not always be in simplified form. That is, a subset of the constraints may imply the entire set. For example, consider a loop "`for(int i=0;i!=n;++i)`" with a symbolic bound n.

When running this code with $n = 10$, we will collect a set of constraints $\{0 \neq n, 1 \neq n, ..., 10 = n\}$, but the last constraint alone implies the entire set.

To simplify constraints, TERN uses a greedy algorithm. Given a set of constraints C, it iterates through each constraint c, and checks if $C/\{c\}$ implies $\{c\}$. If so, it simply discards c. Our observation is that constraints collected later in a run tend to be more compact than the earlier ones. Thus, when pruning constraints, we start from the ones collected earlier. Although we could have used the underlying symbolic execution engine to simplify constraints, it lacks this domain knowledge and may perform poorly.

3.7.4 Slicing Out Irrelevant Branches

A branch statement may observe a piece of symbolic data but perform no synchronization operation in either branch. The constraints collected from this branch are unlikely to affect schedules. If we include irrelevant constraints in the input constraints of a schedule, we not only increase constraint checking time, but also preclude legal reuses of the schedule.

To address this problem, TERN employs a simple static analysis to automatically prune likely irrelevant constraints. At the heart of this technique is a slicing analysis that identifies branch statements unlikely to affect synchronization operations. Specifically, given a branch statement s, this analysis computes s_d, the immediate post-dominator [2] of s, and marks s as irrelevant if no synchronization operations are between s and s_d. Although simple, this technique reduced constraint checking time significantly (§3.8.3). However, we note that our analysis is unsound because it ignores data dependencies. Thus, we plan to implement a sound slicing algorithm [30] in our future work.

3.8 Evaluation

Our TERN implementation consists of 8,934 lines of C++ code, including 827 lines for the instrumentor implemented as an LLVM pass; 5,451 lines for the proxy, schedule cache, memoizer, and replayer; and 2,656 lines for modifications to KLEE.

We evaluated TERN on a diverse set of 14 programs, ranging from two server programs, `Apache`

and `MySQL`, to one parallel compression utility, `PBZip2`, to 11 scientific programs in SPLASH-2.[2]

Our main evaluation machine is a 2.66 GHz quad-core Intel machine with 4 GB memory running Linux 2.6.24. When evaluating TERN on server programs, we ran the server on this machine and the client on another to avoid unnecessary contention. These machines are connected via 1Gbps LAN. We compiled all programs down to machine code using `llvm-gcc -O2` and LLVM's bitcode compiler `llc`.

We focused our evaluation on four key questions:

1. Is TERN easy to use (§3.8.1)?
2. Does TERN make multithreaded programs stable across different inputs (§3.8.2)?
3. Does TERN incur high overhead (§3.8.3)?
4. Does TERN make multithreaded programs deterministic on the same input (§3.8.4)?

3.8.1 Ease of Use

Table 3.2 summarizes the modifications we made to make the programs work with TERN. For each program but `MySQL`, we modified only 3-10 lines. For `Apache`, we marked the HTTP command, URL, HTTP version, and the return of `cache_find()` as symbolic (§3.6). For `MySQL`, we marked the SQL query. For `PBZip2`, we marked the number of threads and file blocks. (The number of file blocks is set in two places, contributing two symbolic annotations.) For all these scientific programs, we marked all input arguments as symbolic except those configuring output verbosity.[3] We marked three custom synchronization operations in three SPLASH-2 programs. We made two miscellaneous changes to `Apache` and `MySQL`. The line counts are shown in parenthesis under the Total column. For `Apache`, we had to fix an uninitialized memory read in `ap_signal_server()` to make it work with KLEE. For `MySQL`, we wrote a 28-line function to mark the numbers in each SQL query as concrete (i.e., not affecting schedules) to avoid making the input constraints too specific.

[2]The version of the SPLASH2 [68] we acquired has 12 programs, one of which does not compile on our evaluation machine.

[3]Note that we could have used a two-line loop to mark these arguments as symbolic. Instead, we report the total number of symbolic variables to avoid masking real data.

Program	Size	Symbolic	Task	Sync	Total
Apache	464K	4	2	0	6 (+1)
MySQL	1,182K	1	2	0	3 (+28)
PBZip2	1,551	3	N/A	0	3
fft	1,403	4	N/A	0	4
lu	1,265	3	N/A	0	3
barnes	1,954	9	N/A	0	9
radix	661	4	N/A	0	4
fmm	3,208	8	N/A	1	9
ocean	6,494	5	N/A	0	5
volrend	18,082	1	N/A	1	2
water-spatial	1,573	9	N/A	0	9
raytrace	5,808	3	N/A	0	3
water-nsquared	1,188	10	N/A	0	10
cholesky	3,683	3	N/A	1	4

Table 3.2: *Statistics of programs evaluated.* **Size** counts the lines of code for each program. **Symbolic** counts the symbolic variables we marked. **Task** counts the task boundary annotations (`begin_task()` and `end_task()`) we inserted. **Sync** counts the annotations for custom synchronizations we inserted. The numbers in parenthesis under **Total** count miscellaneous changes.

3.8.2 Stability

We evaluated Tern's stability via two sets of experiments. The first set compares it to an existing DMT system (§3.8.2.1). The second quantifies how frequently it can reuse schedules on real and synthetic workloads (§3.8.2.2).

3.8.2.1 Bug Stability

We compared Tern to CoreDet [12] in terms of *bug stability*: does a bug occur in one run but disappear in another when the input varies slightly? We ran three buggy SPLASH-2 programs, fft, lu, and barnes, in three modes: nondeterministic execution (Nondet), with CoreDet, and with Tern. We varied their inputs by varying the number of threads and the amount of computation. For each program, execution mode, and input combination, we ran the program 100 times, and recorded whether the corresponding bug occurred.

	Nondet			CoreDet			Tern		
-p2	✔	✔	✔	✔	✖	✔	✔	✔	✔
-p4	✔	✔	✔	✖	✖	✔	✔	✔	✔
-p8	✔	✔	✔	✖	✖	✖	✔	✔	✔
Args.	**-m10**	**12**	**14**	**-m10**	**12**	**14**	**-m10**	**12**	**14**

Table 3.3: *Bug stability results on SPLASH-2* `fft`. The leftmost column and the bottommost row show the command line arguments. Option **-p** specifies the number of threads, and **-m** the amount of computation (matrix size). Symbol ✖ indicates that the bug occured, and ✔ the bug never occured.

We present only the fft results; the results of the other programs are similar. Table 3.3 shows the buggy behaviors of fft. In nondeterministic mode, the bug never occurred, despite that each run almost always yielded a new synchronization order. With CoreDet, slight changes in computation made the bug occur or disappear. With Tern, the bug never occurred, and Tern reused only three schedules for all runs, one for each thread count.

3.8.2.2 Reuse Rates

We also quantified how frequently Tern could reuse schedules. Specifically, we measured the overall reuse rate, defined as the number of inputs processed using memoized schedules over the total number of inputs. The higher the reuse rates, the more stable the programs become. Tern had nearly 100% overall reuse rates for the scientific programs after a small number of memoization runs. Thus, we focused on `Apache`, `MySQL`, and `PBZip2` in out experiments.

We used four workloads to evaluate overall reuse rates:

Apache-CS: a real 4-day trace from the Columbia CS website with 122,000 HTTP requests. We wrote a script to replay this trace at a rate of 100 concurrent requests per second.

SysBench-simple: SysBench [106] in simple mode. This synthetic workload consists of random select queries.

SysBench-tx: SysBench in transaction mode. This synthetic workload consists of random select, update, delete, and insert queries.

PBZip2-usr: a random selection of 10,000 files from `/usr` on our evaluation machine.

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

Table 3.4: Tern *stability.* Column **Schedules** indicates the number of schedules in the schedule cache.

For each workload, we first randomly selected 1%-3% of the workload and ran the memoizer to populate the schedule cache. We then ran the entire workload with the replayer and measured the overall reuse rates. We ran eight worker threads for each program because they performed best (with or without Tern) with this setting.

Table 3.4 shows the results. For three out of the four workloads, Tern could reuse a small number of schedules to process over 90% of the inputs. For MySQL-tx, Tern had a lower overall reuse rate. The reasons are two fold. First, this workload makes it unlikely to reuse schedules because it mixes many randomly generated queries with different types and parameters. Second, we annotated only the SQL command as symbolic without exposing the hidden states of `MySQL` (§3.6) so that we could measure Tern's performance in an adversarial setting. Nonetheless, Tern managed to process 44.2% of inputs with a small number of schedules.

3.8.3 Overhead

We used the following workloads to evaluate Tern's overhead. For `Apache`, we used ApacheBench [5] to repeatedly download a 50KB webpage. For `MySQL`, we used the SysBench-simple workload from the previous subsection. Both ApacheBench and SysBench are used by the server developers themselves. We made these benchmarks CPU bound by fitting the web or database in memory and by connecting the server and client via a 1 Gbps LAN. For `PBZip2`, we decompressed a 10 MB file. For SPLASH-2 programs, we ran them typically for 10-100 ms. We measured the execution time for batch programs and the throughput (TPUT) and response time (RESP) for server programs. All numbers reported in this section were averaged over 50 runs.

The most performance-critical component is the replayer because it operates during the normal execution of a program. Figure 3.10 shows the relative overhead of the replayer over nondeter-

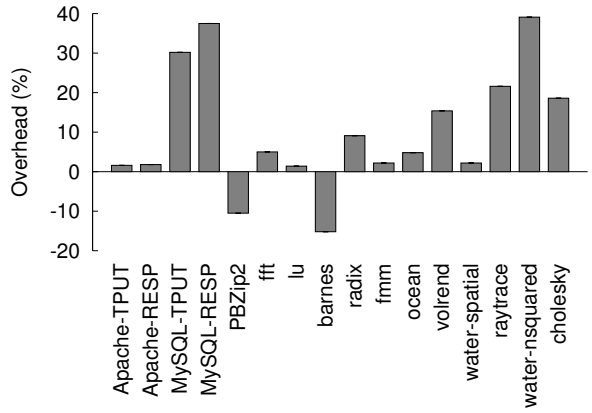


Figure 3.10: *Relative overhead of the replayer over nondeterministic execution.* Negative overhead means speedup.

Program	**Nondet**	**Memoization**	**Overhead (times)**
Apache-TPUT	462.2 req/s	2.1 req/s	219.1
Apache-RESP	0.22 s	3.96 s	17.0
MySQL-TPUT	13779.3 req/s	172.2 req/s	79.0
MySQL-RESP	0.6 ms	61 ms	100.6
`PBZip2`	0.18 s	15.19 s	83.4

Table 3.5: *Overhead of the memoizer.*

ministic execution, the smaller the better. For seven out of the fourteen programs, the replayer performed almost identically to nondeterministic execution. For `PBZip2` and barnes, Tern performed better. This speedup came partially from the optimization to remove unnecessary synchronizations, discussed in the next paragraph. Tern's overhead for `MySQL`, volrend, raytrace, water-nsquared, and choleskey is relatively large because these programs performed many synchronization operations over a short period of time. For instance, water-nsquared and cholesky both call `pthread_mutex_lock()` and `pthread_mutex_unlock()` in a tight loop.

We also measured the effects of skipping unnecessary synchronizations (§3.7.2). Figure 3.11 shows the results. This optimization significantly reduced the replayer's overhead for four programs. Specifically, it made `PBZip2` and barnes run faster than nondeterministic execution, and reduced the overhead of water-nsquared from 172.4% to 39.1%. Its effects on the other programs are negligible and thus not shown.

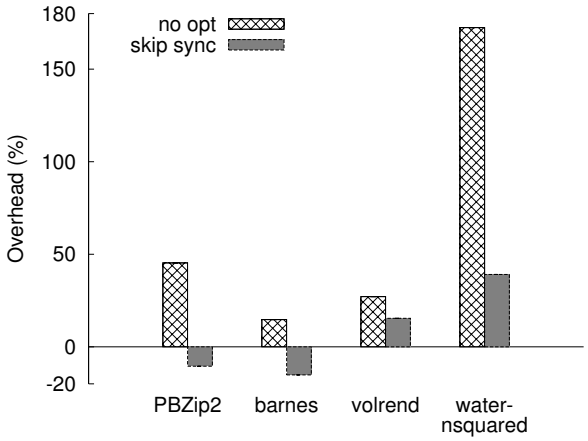


Figure 3.11: *Overhead reduction by skipping unnecessary synchronizations.* "no opt" indicates the baseline overhead.

To reuse a schedule on an input, TERN must check the input against memoized constraints. Constraint checking can be costly, and TERN provides two optimizations to speed it up (§3.7.3 and §3.7.4). Figure 3.12 shows these optimizations can effectively speed up constraint checking for `Apache`, fft, lu, and radix. In particular, they reduced the constraint checking time for lu by 16x.

Compared to the replayer, the memoizer can run offline, thus its performance is not as critical. Table 3.5 shows that this slowdown can sometimes exceed 200x. The main reason is that KLEE, the symbolic engine used, interprets programs instead of running them natively. An instrumentation-based approach can greatly reduce this slowdown [23], which we plan to implement in our future work.

3.8.4 Determinism

We evaluated TERN's determinism via three sets of experiments. The first set checked the memoized schedules for races (§3.8.4.1). The second evaluated TERN's ability to deterministically reproduce or avoid bugs (§3.8.4.2). The third measured how deterministic memory accesses are with and without TERN (§3.8.4.3).

3.8.4.1 Race Detection Results

When memoizing schedules for each of the 14 programs, we turned on TERN's race detector. We found that except for radix and cholesky, the schedules TERN memoized for all other programs were

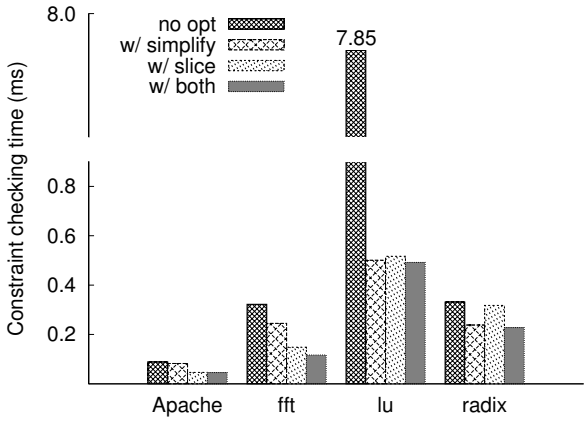


Figure 3.12: *Optimizations to speed up constraint checking.* Note the y-axis is broken. "no opt" indicates the baseline constraint checking time. "simplify" refers to the optimization in §3.7.3. "slice" refers to the optimization in §3.7.4.

free of schedule races and symbolic races with respect to the symbolic data we annotated (§3.7.1). Our race detection result is not surprising because most schedules are indeed race free. It implies that, for runs that reuse the memoized schedules of all programs but radix and cholesky, TERN ensures determinism, barring the assumption discussed in §3.7.1.

3.8.4.2 Bug Determinism

We also evaluated how deterministically TERN could reproduce or avoid bugs. Table 3.6 lists five real concurrency bugs we used. We selected them because they were frequently used in previous studies [69, 71, 86, 87] and we could reproduce them on our evaluation machine. To measure bug determinism, we first memoized schedules for programs listed in Table 3.6. We then manually inserted `usleep()` to these programs to get alternate schedules. We then ran the buggy programs again, reusing the memoized schedules. We also injected random delays into the reuse runs to perturb timing. We found that, TERN consistently reproduced or avoided all five bugs. We verified this result by inspecting the memoized schedules.

3.8.4.3 Memory Access Determinism

TERN enforces synchronization orders, which should make memory access orders more deterministic. We quantified this effect over `Apache` and `PBZip2`. Specifically, we instrumented `Apache` with

Program	Error Description
`Apache`	Reference count decrement and check against 0 are not atomic.
`PBZip2`	Variable `fifo` is used in one thread after being freed by another.
fft	`initdonetime` and `finishtime` are read before assigned the correct values.
lu	Variable `rf` is read before assigned the correct value.
barnes	Variable `tracktime` is read before assigned the correct value.

Table 3.6: *Concurrency errors used in evaluation.*

Program	Length	Nondet	Tern	Ratio
`Apache`	148,058	86,215	10,821	7.97
`PBZip2`	1,234	161	69	2.33

Table 3.7: *Memory access determinism.* We traced memory accessed only from `PBZip2`, not the external BZip2 library.

LLVM to trace accesses to global variables and the heap, a crude approximation of shared memory. We ran `Apache` with Tern to serve five HTTP requests and collected a trace of memory accesses. We then repeated this experiment 20 times to collect 20 traces, and computed the average pairwise edit distance [110]. We then measured the same edit distance for `Apache` in nondeterministic execution mode and compared the two. We did the same comparison for `PBZip2` with a decompression workload of 2MB. Table 3.7 shows the result. For `Apache`, runs with Tern were 7.97 times more deterministic than those without. For `PBZip2`, Tern was 2.33 times more deterministic, but the memory trace had only 1,234 accesses on average.

3.9 Related Work

StableMT and DMT systems. Although TERN provides determinism, it differs from existing DMT systems [12, 34, 80] by making threads stable, i.e., repeating familiar behaviors across different inputs. Another difference is that TERN reduces timing nondeterminism for server programs through the windowing approach.

The closest system to TERN in this category is Kendo [80], a software-only DMT system that also enforces synchronization orders instead of memory access orders for efficiency. COREDET [12] is another software-only DMT system that enforces deterministic memory access orders. Both systems are based on logical clocks and have been shown to work on scientific benchmarks, such as SPLASH-2. The authors of COREDET have noted that a small modification to the original program leads to a much different COREDET-instrumented program, which the idea of schedule memoization may address. COREDET is a software implementation (with extensions) of DMP [34], a hardware DMT system .

Grace [16] proposes a novel approach to making C and C++ programs with fork-join parallelism behave like sequential programs. It runs each thread within a process and commits memory writes atomically and deterministically. It detects memory access conflicts efficiently using hardware page protection. Grace has been shown to perform and scale well on Phoenix benchmarks [93] and a Cilk [19] benchmark. Unlike Grace, TERN aims to make general multithreaded programs, not just fork-join programs, deterministic and stable.

Deterministic Replay. Deterministic replay [4, 35, 36, 42, 47, 61, 62, 74, 87, 104, 108] aims to replay the exact recorded executions, whereas TERN "replays" memoized schedules on different inputs. Some recent deterministic replay systems include Scribe, which tracks page ownership to enforce deterministic memory access [62]; Capo, which defines a novel software-hardware interface and a set of abstractions for efficient replay [74]; PRES and ODR, which systematically search for a complete execution based on a partial one [4, 87]; and SMP-ReVirt, which uses clever page protection trick for recording the order of conflicting memory accesses [36].

Concurrency Errors. The complexity in developing multithreaded programs has led to many concurrency errors [71]. A significant number of them are not data races, but atomicity and order errors [71], which can be deterministically reproduced or avoided using only synchronization orders.

Much work exists on concurrency error detection [38, 69, 70, 97, 123, 126], diagnosis [85, 86, 99],

and correction [55, 111]. TERN aims to make the executions of multithreaded programs deterministic and stable, and is complementary to existing work on concurrency errors. Specifically, TERN can use existing work to detect and fix the errors in the schedules it selects. Moreover, even for programs free of concurrency errors, TERN still provides value by making their behaviors repeatable.

Symbolic Execution. The combination of symbolic and concrete executions has been a hot research topic. Researchers have built scalable and effective symbolic execution systems to detect errors [23–25, 27, 43–45, 98, 119], block malicious inputs [30], and preserve privacy in error reports [26]. Compared to these systems, TERN applies symbolic execution to a new domain: tracking input constraints to reuse schedules.

3.10 Summary

We have presented TERN, the first stable and deterministic multithreading system that makes general multithreaded programs stable by repeating the same schedules on different inputs. TERN does so using schedule memoization: if a schedule is shown to work on an input, TERN memoizes the schedule; if a similar input arrives later, TERN simply reuses the memoized schedule. TERN is also the first DMT system to mitigate input timing nondeterminism for server programs.

Our TERN implementation runs on Linux. It requires no new hardware, no modifications to the underlying OS or synchronization library, and only a few lines of modifications to the multithreaded programs. We evaluated TERN on a diverse set of real programs, including two server programs, one desktop program, and 11 scientific programs. Our results show that TERN is easy to use, makes programs more deterministic and stable, and has reasonable overhead (i.e., good efficiency). TERN is the first stable and deterministic multithreading system shown to work on applications as large, complex, and nondeterministic as `MySQL` and `Apache`. It demonstrates that StableMT and DMT have the potential to greatly improve understanding, testing, and debugging of multithreaded programs, making these programs much easier to get right.

Chapter 4

Efficiently Enforcing Schedules without Deviation

The last chapter presents Tern, the first multithreading system that is efficient, stable, and deterministic. However, Tern enforces best-effort determinism, and its executions may deviate from the memoized schedules when data races exist. Thus, a second challenge on building StableMT arises: how to efficiently enforce schedules without deviation? This challenge also exists in the area of deterministic execution and replay for decades. To address this challenge, this chapter presents Peregrine, our second StableMT (and also DMT) system with a new technique called *schedule relaxation*.

4.1 Introduction

As described in Chapter 2, a root cause that makes multithreaded programs so difficult to get right is: these programs have exponentially many possible schedules for all inputs at runtime. Even running on the same input, the concurrently running threads of a program may interleave in too many different ways, depending on factors such as hardware timing and OS scheduling. This is the so called "nondeterminism." Considering all inputs, the number of possible schedules is even greater. It is extremely challenging to understand, test, analyze, or verify all these schedules in a multithreaded program. Therefore, a concurrency bug within an unchecked schedule can show up in production runs and lead to severe failures and vulnerabilities.

To reduce the number of possible schedules and make multithreading easier to get right, two complementary techniques have been invented by researchers recently. StableMT [8, 31], which is created by my collaborators and me, aims to reduce the number possibles schedules on all inputs. DMT [12, 13, 16, 34, 80] addresses the nondeterminism problem, and it focuses on reducing the number of possible schedules on each input down to one. Notably, our TERN system described in Chapter 3 is the first multithreading system that is both stable and deterministic. Unfortunately, despite these effort, an open challenge [14] well recognized by the research community remains: how to efficiently enforce schedules without deviation for general multithreaded programs on commodity multiprocessors? When enforcing existing DMT systems' schedules, program executions either incur prohibitive overhead, or may deviate from the schedules if there are data races (i.e., these executions are not *fully deterministic*).

As mentioned in Chapter 1, existing DMT systems specifically enforce two forms of schedules: (1) a mem-schedule is a deterministic schedule of shared memory accesses [12, 13, 34], such as `load/store` instructions, and (2) a sync-schedule is a deterministic order of synchronization operations [31, 80], such as `lock()/unlock()`. Enforcing a mem-schedule is fully deterministic even for programs with data races, but may incur prohibitive overhead (e.g., roughly 1.2×-6× [12]). Enforcing a sync-schedule is efficient (e.g., average 16% slowdown [80]) because most code does not control synchronization and can still run in parallel, but a sync-schedule is only deterministic for race-free programs, when, in fact, most real programs have races, harmful or benign [71, 116]. The dilemma is, then, to pick either fully determinism or efficiency, but not both.

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

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

race. When the same input is provided again, PEREGRINE can reuse this schedule deterministically and efficiently.

Reusing a schedule only when the program input matches exactly is too limiting, and we aim to make PEREGRINE also a StableMT system that can frequently reuse schedules on a wide range of inputs. Fortunately, the schedules PEREGRINE computes are often "coarse-grained" and reusable on a broad range of inputs. Indeed, our previous work has shown that a small number of sync-schedules can often cover over 90% of the workloads for real programs such as `Apache` [31]. The higher the reuse rates, the more efficient PEREGRINE is.

Before reusing a schedule on an input, PEREGRINE must check that the input satisfies the input constraints of the schedule, so that (1) the schedule is feasible, i.e., the execution on the input will reach all events in the same deterministic order as in the schedule and (2) the execution will not introduce new races. (New races may occur if they are *input-dependent*; see §4.4.1.) A naïve approach is to collect preconditions from all input-dependent branches in an execution trace. For instance, if a branch instruction inspects input variable `X` and goes down the true branch, we collect a precondition that `X` must be nonzero. Preconditions collected via this approach ensures that an execution on an input satisfying the preconditions will always follow the path of the recorded execution in all threads. However, many of these branches concern thread-local computations and do not affect the program's ability to follow the schedule. Including them in the preconditions thus unnecessarily decreases schedule-reuse rates.

How can PEREGRINE compute sufficient preconditions to avoid new races and ensure that a schedule is feasible? How can PEREGRINE filter out unnecessary branches to increase schedule-reuse rates? Our previous work, TERN [31], requires developers to grovel through the code and mark the input affecting schedules; even so, it does not guarantee full determinism if there are data races.

PEREGRINE addresses these challenges with two new program analysis techniques. First, given an execution trace and a hybrid schedule, it computes sufficient preconditions using *determinism-preserving slicing*, a new precondition slicing [30] technique designed for multithreaded programs. Precondition slicing takes an execution trace and a *target* instruction in the trace, and computes a *trace slice* that captures the instructions required for the execution to reach the target with equivalent operand values. Intuitively, these instructions include "branches whose outcome matters"

to reach the target and "mutations that affect the outcome of those branches" [30]. This trace slice typically has much fewer branches than the original execution trace, so that we can compute more relaxed preconditions. However, previous work [30] does not compute correct trace slices for multithreaded programs or handle multiple targets; our slicing technique correctly handles both cases.

Our slicing technique often needs to determine whether two pointer variables may point to the same object. *Alias analysis* is the standard static technique to answer these queries. Unfortunately, one of the best alias analyses [113] yields overly imprecise results for 30% of the evaluated programs, forcing PEREGRINE to reuse schedules only when the input matches almost exactly. The reason is that standard alias analysis has to be conservative and assume all possible executions, yet PEREGRINE cares about alias results according only to the executions that reuse a specific schedule. To improve precision, PEREGRINE uses *schedule-guided simplification* to first simplify a program according to a schedule, then runs standard alias analysis on the simplified program to get more precise results. For instance, if the schedule dictates eight threads, PEREGRINE can clone the corresponding thread function eight times, so that alias analysis can separate the results for each thread, instead of imprecisely merging results for all threads.

We have built a prototype of PEREGRINE that runs in user-space. It automatically tracks `main()` arguments, data read from files and sockets, and values returned by `random()`-variants as input. It handles long-running servers by splitting their executions into windows and reusing schedules across windows [31]. The hybrid schedules it computes are fully deterministic for programs that (1) have no nondeterminism sources beyond thread scheduling, data races, and inputs tracked by PEREGRINE and (2) adhere to the assumptions of the tools PEREGRINE uses.

We evaluated PEREGRINE on a diverse set of 18 programs, including the `Apache` web server [6]; three desktop programs, such as `PBZip2` [89], a parallel compression utility; implementations of 12 computation-intensive algorithms in the popular SPLASH-2 and PARSEC benchmark suites; and `racey` [50], a benchmark with numerous intentional races for evaluating deterministic execution and replay systems. Our results show that PEREGRINE is both deterministic and efficient (executions reusing schedules range from 68.7% faster to 46.6% slower than nondeterministic executions); it can frequently reuse schedules for half of the programs (e.g., two schedules cover all possible inputs to `PBZip2` compression as long as the number of threads is the same); both its slicing and simplification

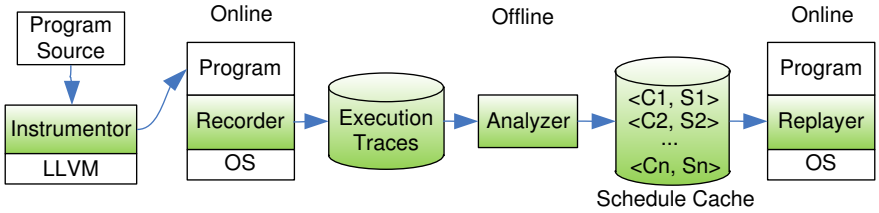


Figure 4.1: PEREGRINE *Architecture: components and data structures are shaded (and in green).*

techniques are crucial for increasing schedule-reuse rates, and have reasonable overhead when run offline; its recording overhead is relatively high, but can be reduced using existing techniques [18]; and it requires no manual effort except a few annotations for handling server programs and for improving precision.

Our main contributions are the schedule-relaxation approach and PEREGRINE, an efficient stable and deterministic multithreading system. Additional contributions include the ideas of hybrid schedules, determinism-preserving slicing, and schedule-guided simplification. To our knowledge, our slicing technique is the first to compute correct (non-trivial) preconditions for multithreaded programs. We believe these ideas apply beyond PEREGRINE (§4.2.2).

The remainder of this chapter is organized as follows. We first present a high-level design of PEREGRINE (§4.2). We then describe its core ideas: hybrid schedules (§4.3), determinism-preserving slicing (§4.4), and schedule-guided simplification (§4.5). We then present implementation issues (§4.6) and evaluation (§4.7). We finally discuss related work (§4.8) and conclude (§4.9).

4.2 High-level Design

Figure 4.1 shows the architecture of PEREGRINE. It has four main components: the instrumentor, recorder, analyzer, and replayer. The *instrumentor* is an LLVM [67] compiler plugin that prepares a program for use with PEREGRINE. It instruments synchronization operations such as `pthread_mutex_lock()`, which the recorder and replayer control at runtime. It marks the `main()` arguments, data read from `read()`, `fscanf()`, and `recv()`, and values returned by `random()`-variants as inputs. We chose LLVM [67] as our instrumentation framework for its compatibility with GCC and easy-to-analyze intermediate representation (IR). However, our approach is general and should apply beyond LLVM. For clarity, we will present our examples and algorithms at the

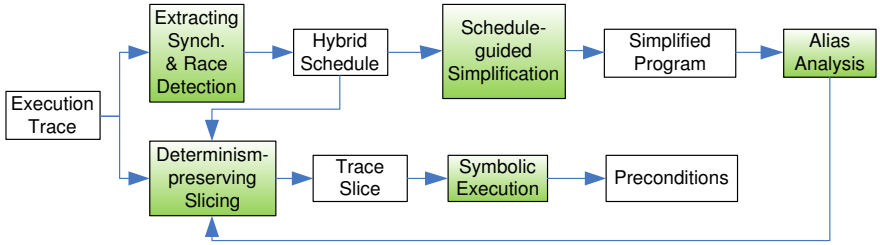


Figure 4.2: *Analyses performed by the analyzer.*

source level, instead of the LLVM IR level.

The *recorder* is similar to existing systems that deterministically record executions [18, 36, 62]. Our current recorder is implemented as an LLVM interpreter. When a program runs, the recorder saves the LLVM instructions interpreted for each thread into a central log file. The recorder does not record external input data, such as data read from a file, because our analysis does not need this information. To schedule synchronization operations issued by different threads, the recorder can use a variety of DMT algorithms [31].

The *analyzer* is a stand-alone program that computes (1) a hybrid schedule S and (2) the preconditions C required for reusing the schedule on future inputs. It does so using a series of analyses, shown in Figure 4.2. To compute a hybrid schedule, the analyzer first extracts a total order of synchronization operations from the execution trace. It then detects data races according to this synchronization order, and computes additional *execution order* constraints to deterministically resolve the detected races. To compute the preconditions of a schedule, the analyzer first *simplifies* the program according to the schedule, so that alias analysis can compute more precise results. It then slices the execution trace into a trace slice with instructions required to avoid new races and reach all events in the schedule. It then uses *symbolic execution* [59] to collect preconditions from the input-dependent branches in the slice. The trace slice is typically much smaller than the execution trace, so that the analyzer can compute relaxed preconditions, allowing frequent reuses of the schedule. The analyzer finally stores $\langle C, S \rangle$ into the schedule cache, which conceptually holds a set of such tuples. (The actual representation is tree-based for fast lookup [31].)

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

Otherwise, it forwards the input to the recorder.

In the remainder of this section, we first use an example to illustrate how PEREGRINE works, highlighting the operation of the analyzer (§4.2.1). We then describe PEREGRINE's deployment and usage scenarios (§4.2.2) and assumptions (§4.2.3).

4.2.1 An Example

Figure 4.3 shows our running example, a simple multithreaded program based on the real ones used in our evaluation. It first parses the command line arguments into `nthread` (line L1) and `size` (L2), then spawns `nthread` threads including the main thread (L4–L5) and processes `size/nthread` bytes of data in each thread. The thread function `worker()` allocates a local buffer (L10), reads data from a file (L11), processes the data (L12–L13), and sums the results into the shared variable `result` (L14–L16). The `main()` function may further update `result` depending on `argv[3]` (L7–L8), and finally prints out `result` (L9). This example has read-write and write-write races on `result` due to missing `pthread_join()`. This error pattern matches some of the real errors in the evaluated programs such as `PBZip2`.

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

Recorder: execution trace. When we run the instrumented program with arguments "`2 2 0`" to spawn two threads and process two bytes of data, suppose that the recorder records the execution trace in Figure 4.4. (This figure also shows the hybrid schedule and preconditions PEREGRINE computes, explained later in this subsection.) This trace is just one possible trace depending on the scheduling algorithm the recorder uses.

Analyzer: hybrid schedule. Given the execution trace, the analyzer starts by computing a hybrid schedule. It first extracts a sync-schedule consisting of the operations tagged with (1), (2), ..., (8) in Figure 4.4. It then detects races in the trace according to this sync-schedule, and finds the race on `result` between L15 of thread t_1 and L9 of t_0. It then computes an execution order constraint to deterministically resolve this race, shown as the dotted arrow in Figure 4.4. The

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

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

Figure 4.3: *Running example.* It uses the common divide-and-conquer idiom to split work among multiple threads. It contains write-write (lines L8 and L15) and read-write (lines L9 and L15) races on `result` because of missing `pthread_join()`.

sync-schedule and execution order constraint together form the hybrid schedule. Although this hybrid schedule constrains the order of synchronization and the last two accesses to `result`, it can still be efficiently reused because the core computation done by `worker` can still run in parallel.

Analyzer: simplified program. To improve analysis precision, the analyzer simplifies the pro-

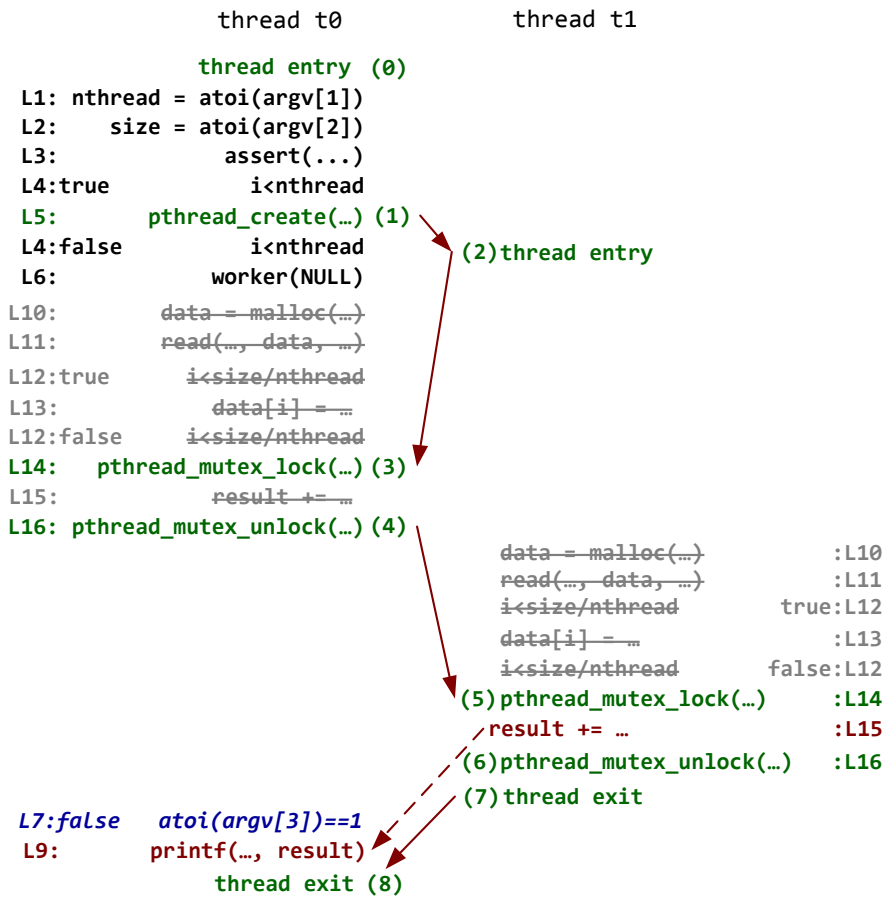


Figure 4.4: *Execution trace, hybrid schedule, and trace slice.* An execution trace of the program in Figure 4.3 on arguments "2 2 0" is shown. Each executed instruction is tagged with its static line number Li. Branch instructions are also tagged with their outcome (true or false). Synchronization operations (green), including thread entry and exit, are tagged with their relative positions in the synchronization order. They form a sync-schedule whose order constraints are shown with solid arrows. L15 of thread t_1 and L9 of thread t_0 race on `result`, and this race is deterministically resolved by enforcing an execution order constraint shown by the dotted arrow. Together, these order constraints form a hybrid schedule. Instruction L7 of t_0 (italic and blue) is included in the trace slice to avoid new races, while L6, L4:false, L4:true, L3, L2, and L1 of t_0 are included due to intra-thread dependencies. Crossed-out (gray) instructions are elided from the slice.

gram according to the hybrid schedule. For instance, based on the number of `pthread_create()` operations in the schedule, the analyzer clones function `worker()` to give each thread a copy, so

that the alias analysis separates different threads and determines that the two instances of L13 in t_0 and t_1 access different `malloc`'ed locations and never race.

Analyzer: trace slice. The analyzer uses determinism-preserving slicing to reduce the execution trace into a trace slice, so that it can compute relaxed preconditions. The final trace slice consists of the instructions not crossed out in Figure 4.4. The analyzer computes this trace slice using inter-thread and intra-thread steps. In the inter-thread step, it adds instructions required to avoid new races into the slice. Specifically, for t_0 it adds the false branch of L7, or L7:false, because if the true branch is taken, a new race between L8 of t_0 and L15 of t_1 occurs. It ignores branches of line L12 because alias analysis already determines that L13 of t_0 and L13 of t_1 never race.

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

Analyzer: preconditions. After slicing, all branches from L12 are gone. The analyzer joins the remaining branches together as the preconditions, using a version of KLEE [24] augmented with thread support [31]. Specifically, the analyzer marks input data as *symbolic*, and then uses KLEE to track how this symbolic data is propagated and observed by the instructions in the trace slice. (Our PEREGRINE prototype runs symbolic execution within the recorder for simplicity; see §4.6.1.) If a branch instruction inspects symbolic data and proceeds down the true branch, the analyzer adds the precondition that the symbolic data makes the branch condition true. The analyzer uses symbolic summaries [30] to succinctly generalize common library functions. For instance, it considers the return of `atoi(arg)` symbolic if `arg` is symbolic.

Figure 4.5 shows the preconditions the analyzer computes from the trace slice in Figure 4.4. These preconditions illustrate two key benefits of PEREGRINE. First, they are sufficient to ensure deterministic reuses of the schedule. Second, they only loosely constrain the data size ($atoi\_argv_2$)

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

Figure 4.5: *Preconditions computed from the trace slice in Figure 4.4.* Variable $atoi\_argv_i$ represents the return of `atoi(arg[i])`.

and do not constrain the data contents (from `read()`), allowing frequent schedule-reuses. The reason is that L10–L13 are all sliced out. One way to leverage this benefit is to populate a schedule cache with small workloads to reduce analysis time, and then reuse the schedules on large workloads.

Replayer. Suppose we run this program again on different arguments "`2 1000 8`." The replayer checks the new arguments against the preconditions in Figure 4.5 using KLEE's constraint checker, and finds that these arguments satisfy the preconditions, despite the much larger data size. It can therefore reuse the hybrid schedule in Figure 4.4 on this new input by enforcing the same order of synchronization operations and accesses to `result`.

4.2.2 Deployment and Usage Scenarios

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

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

Record and replay. Existing deterministic record-replay systems tend to incur high CPU and storage overhead (e.g., 15X slowdown [18] and 11.7 GB/day storage [36]). A record-replay system on top of PEREGRINE may drastically reduce this overhead: for inputs that hit the schedule cache, we do not have to log any schedule.

Replication. To keep replicas of a multithreaded program consistent, a replication tool often records the thread schedules at one replica and replays them at others. This technique is essen-

tially *online* replay [64]. It may thus incur high CPU, storage, and bandwidth overhead. With PEREGRINE, replicas can maintain a consistent schedule cache. If an input hits the schedule cache, all replicas will automatically select the same deterministic schedule, incurring zero bandwidth overhead.

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

Applications of individual techniques. The individual ideas in PEREGRINE can also benefit other research effort. For instance, hybrid schedules can make the sync-schedule approach deterministic without recording executions, by coupling it with a sound static race detector. Determinism-preserving slicing can (1) compute input filters to block bad inputs [30] causing concurrency errors and (2) randomize an input causing a concurrency error for use with anonymous bug reporting [26]. Schedule-guided simplification can transparently improve the precision of many existing static analyses: simply run them on the simplified programs. This improved precision may be leveraged to accurately detect errors or even verify the correctness of a program according to a set of schedules. Indeed, from a verification perspective, our simplification technique helps *verify* that executions reusing schedules have *no* new races.

4.2.3 Assumptions

At a design level, we anticipate the schedule-relaxation approach to work well for many programs/workloads as long as (1) they can benefit from repeatability, (2) their schedules can be frequently reused, (3) their races are rare, and (4) their nondeterminism comes from the sources tracked by PEREGRINE. This approach is certainly not designed for every multithreaded program. For instance, like other DMT systems, PEREGRINE should not be used for parallel simulators that desire nondeterminism for statistical confidence. For programs/workloads that rarely reuse schedules, PEREGRINE may be unable to amortize the cost of recording and analyzing execution traces. For programs full of races, enforcing hybrid schedules may be as slow as mem-schedules.

PEREGRINE addresses nondeterminism due to thread scheduling and data races. It mitigates input nondeterminism by reusing schedules on different inputs. It currently considers command line arguments, data read from a file or a socket, and the values returned by řandom()-variants as inputs. PEREGRINE ensures that schedule-reuses are fully deterministic if a program contains only these nondeterminism sources, an assumption met by typical programs. If a program is nondeterministic due to other sources, such as functions that query physical time (e.g., ğettimeofday()), pointer addresses returned by m̆alloc(), and nondeterminism in the kernel or external libraries, PEREGRINE relies on developers to annotate these sources.

The techniques that PEREGRINE leverages make assumptions as well. PEREGRINE computes preconditions from a trace slice using the symbolic execution engine KLEE, which does not handle floating point operations; though recent work [29] has made advances in symbolic execution of floating point programs. (Note that floating point operations not in trace slices are not an issue.) We explicitly designed PEREGRINE's slicing technique to compute sufficient preconditions, but these preconditions may still include unnecessary ones, because computing the *weakest* (most relaxed) preconditions in general is undecidable [2]. The alias analysis PEREGRINE uses makes a few assumptions about the analyzed programs [10]; a "sounder" alias analysis [49] would remove these assumptions. These analyses may all get expensive for large programs. For server programs, PEREGRINE borrows the windowing idea from our previous work [31]; it is thus similarly limited (§4.6.3).

At an implementation level, PEREGRINE uses the LLVM framework, thus requiring that a program is in either source (so we can compile using LLVM) or LLVM IR format. PEREGRINE ignores inline x86 assembly or calls to external functions it does not know. For soundness, developers have to lift x86 assembly to LLVM IR and provide summaries for external functions. (The external function problem is alleviated because KLEE comes with a L̆ibc implementation.) Currently PEREGRINE works only with a single process, but previous work [13] has demonstrated how DMT systems can be extended to multiple processes.

4.3 Hybrid Schedules

This section describes how PEREGRINE computes (§4.3.1) and enforces (§4.3.2) hybrid schedules.

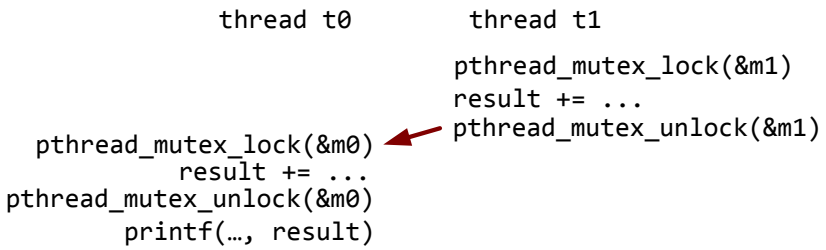


Figure 4.6: *No* PEREGRINE *race with respect to this schedule.*

4.3.1 Computing Hybrid Schedules

To compute a hybrid schedule, PEREGRINE first extracts a total order of synchronization operations from an execution trace. It considers 28 Pthreads operations, such as `pthread_mutex_lock()` and `pthread_cond_wait()`. It also considers the entry and exit of a thread as synchronization operations so that it can order these events together with other synchronization operations. These operations are sufficient to run the programs evaluated, and more can be easily added. PEREGRINE uses a total, instead of a partial, order because previous work has shown that a total order is already efficient [31, 80].

For determinism, PEREGRINE must detect races that occurred during the recorded execution and compute execution order constraints to deterministically resolve the races. An off-the-shelf race detector would flag too many races because it considers the original synchronization constraints of the program, whereas PEREGRINE wants to detect races according to a sync-schedule [87, 95]. To illustrate, consider Figure 4.6, a modified sync-schedule based on the one in Figure 4.4. Suppose the two threads acquire different mutex variables, and thread t_1 acquires and releases its mutex before t_0. Typical lockset-based [97] or happens-before-based [63] race detectors would flag a race on `result`, but our race detector does not: the sync-schedule in the figure deterministically resolves the order of accesses to `result`. Sync-schedules anecdotally reduced the number of possible races greatly, in one extreme case, from more than a million to four [87].

Mechanically, PEREGRINE detects occurred races using a happens-before-based algorithm. It flags two memory accesses as a race iff (1) they access the same memory location and at least one is a `store` and (2) they are *concurrent*. To determine whether two accesses are concurrent, typical happens-before-based detectors use vector clocks [72] to track logically when the accesses occur. Since PEREGRINE already enforces a total synchronization order, it uses a simpler and more

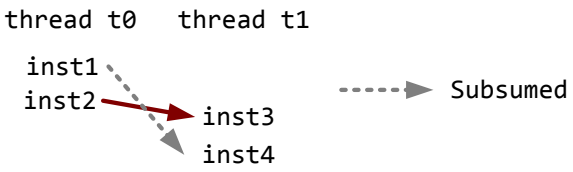


Figure 4.7: *Example subsumed execution order constraint.*

memory-efficient logical clock representation.

Specifically, given two adjacent synchronization operations within one thread with relative positions m and n in the sync-schedule, PEREGRINE uses $[m, n)$ as the logical clock of all instructions executed by the thread between the two synchronization operations. For instance, in Figure 4.4, all instructions run by thread t_0 between the `pthread_mutex_unlock()` operation and the thread exit have clock $[4, 8)$. PEREGRINE considers two accesses with clocks $[m_0, n_0)$ and $[m_1, n_1)$ concurrent if the two clock ranges overlap, i.e., $m_0 < n_1 \wedge m_1 < n_0$. For instance, $[4, 8)$ and $[5, 6)$ are concurrent.

To deterministically resolve a race, PEREGRINE enforces an execution order constraint $inst_1 \rightarrow inst_2$ where $inst_1$ and $inst_2$ are the two dynamic instruction instances involved in the race. PEREGRINE identifies a dynamic instruction instance by $\langle sid, tid, nbr \rangle$ where sid refers to the unique ID of a static instruction in the executable file; tid refers to the internal thread ID maintained by PEREGRINE, which always starts from zero and increments deterministically upon each `pthread_create()`; and nbr refers to the number of control-transfer instructions (branch, call, and return) locally executed within the thread from the last synchronization to instruction $inst_i$. For instance, PEREGRINE represents the execution order constraint in Figure 4.4 as $\langle L15, t_1, 0 \rangle \rightarrow \langle L9, t_0, 2 \rangle$, where the branch count 2 includes the return from `worker` and the branch L7 of thread t_0. We must distinguish different dynamic instances of a static instruction because some of these dynamic instances may be involved in races while others are not. We do so by counting branches because if an instruction is executed twice, there must be a control-transfer between the two instances [36]. We count branches starting from the last synchronization operation because the partial schedule preceding this operation is already made deterministic.

If one execution order constraint subsumes another, PEREGRINE does not add the subsumed one to the schedule. Figure 4.7 shows a subsumed constraint example. Algorithmically, PEREGRINE considers an execution order constraint $inst_1 \rightarrow inst_4$ subsumed by $inst_2 \rightarrow inst_3$ if (1) $inst_1$ and $inst_2$ have the same logical clock (so they must be executed by the same thread) and $inst_2$ occurs no

earlier than $inst_1$ in the recorded execution trace; (2) $inst_3$ and $inst_4$ have the same logical clock and $inst_3$ occurs no later than $inst_4$ in the trace. This algorithm ignores transitive order constraints, so it may miss some subsumed constraints. For instance, it does not consider $inst_1 \rightarrow inst_4$ subsumed if we replace constraint $inst_2 \rightarrow inst_3$ with $inst_2 \rightarrow inst_{other}$ and $inst_{other} \rightarrow inst_3$ where $inst_{other}$ is executed by a third thread.

4.3.2 Enforcing Hybrid Schedules

To enforce a synchronization order, PEREGRINE uses a technique called *semaphore relay* [31] that orders synchronization operations with per-thread semaphores. At runtime, a synchronization wrapper (recall that PEREGRINE instruments synchronization operations for runtime control) waits on the semaphore of the current thread. Once it is woken up, it proceeds with the actual synchronization operation, then wakes up the next thread according to the synchronization order. For programs that frequently do synchronization operations, the overhead of semaphore may be large because it may cause a thread to block. Thus, PEREGRINE also provides a spin-wait version of semaphore relay called *flag relay*. This technique turns out to be very fast for many programs evaluated (§4.7.2).

To enforce an execution order constraint, PEREGRINE uses program instrumentation, avoiding the need for special hardware, such as the often imprecise hardware branch counters [36]. Specifically, given a dynamic instruction instance $\langle sid, tid, nbr \rangle$, PEREGRINE instruments the static instruction *sid* with a semaphore `up()` or `down()` operation. It also instruments the branch instructions counted in *nbr* so that when each of these branch instructions runs, a per-thread branch counter is incremented. PEREGRINE activates the inserted semaphore operation for thread *tid* only when the thread's branch counter matches *nbr*. To avoid interference and unnecessary contention when there are multiple order constraints, PEREGRINE assigns a unique semaphore to each constraint.

PEREGRINE instruments a program by leveraging a fast instrumentation framework we previously built [114]. It keeps two versions of each basic block: a normally compiled, fast version, and a slow backup padded with calls to a `slot()` function before each instruction. As shown in Figure 4.8, the `slot()` function interprets the actions (semaphore `up/down`) to be taken at each instruction. To instrument an instruction, PEREGRINE simply updates the actions for that instruction. This

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

Figure 4.8: *Instrumentation to enforce execution order constraints.*

instrumentation may be expensive, but fortunately, PEREGRINE leaves it off most of the time and turns it on only at the last synchronization operation before an inserted semaphore operation.

PEREGRINE turns on/off this instrumentation by switching a per-thread flag. Upon each function entry, PEREGRINE inserts code to check this flag and determine whether to run the normal or slow version of the basic blocks. PEREGRINE also inserts this check after each function returns in case the callee has switched the per-thread flag. The overhead of these checks tend to be small because the flags are rarely switched and hardware branch predication works well in this case [114].

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

4.4 Determinism-Preserving Slicing

PEREGRINE uses determinism-preserving slicing to (1) compute sufficient preconditions to avoid new races and ensure that a schedule is feasible, and (2) filter many unnecessary preconditions to increase schedule-reuse rates. It does so using inter- and intra-thread steps. In the inter-thread step (§4.4.1), it detects and avoids *input-dependent* races that do not occur in the execution trace, but may occur if we reuse the schedule on a different input. In the intra-thread step (§4.4.1), the analyzer computes a *path slice* per thread by including instructions that may affect the events in

the schedule or the instructions identified in the inter-thread step.

4.4.1 Inter-thread Step

In the inter-thread step, PEREGRINE detects and avoids input-dependent races with respect to a hybrid schedule. An example input-dependent race is the one between lines L8 and L15 in Figure 4.3, which occurs when `atoi(argv[3])` returns 1 causing the true branch of L7 to be taken. Figure 4.9 shows two more types of input-dependent races.

To detect such races, PEREGRINE starts by refining the logical clocks computed based on the sync-schedule (§4.3.1) with execution order constraints because it will also enforce these constraints. PEREGRINE then iterates through all pairs of concurrent *regions*, where a region is a set of instructions with an identical logical clock. For each pair, it detects input-dependent races, and adds the racy instructions to a list of *slicing targets* used by the intra-thread step.

Figure 4.10 shows the algorithm to detect input-dependent races for two concurrent regions. The algorithm iterates through each pair of instructions respectively from the two regions, and handles three types of input-dependent races. First, if neither instruction is a branch instruction, it queries alias analysis to determine whether the instructions *may* race. If so, it adds both instructions to `slicing_targets` and adds additional preconditions to ensure that the pointers dereferenced are different, so that reusing the schedule on a different input does not cause the may-race to become a real race. Figure 4.9(a) shows a race of this type.

Second, if exactly one of the instructions is a branch instruction, the algorithm checks whether the instructions contained in the not-taken branch of this instruction may race with the other instruction (using an interprocedural *post-dominator analysis* [2]). It must check the not-taken branch because a new execution may well take the not-taken branch and cause a race. To avoid

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

(a)

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

(b)

Figure 4.9: *Input-dependent races.* Race (a) occurs when `input1` and `input2` are the same; Race (b) occurs when both true branches are taken.

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

Figure 4.10: *Input-dependent race detection algorithm.*

such a race, PEREGRINE adds the taken branch into the trace slice so that executions reusing the schedule always go down the taken branch. For instance, to avoid the input-dependent race between lines L8 and L15 in Figure 4.3, PEREGRINE includes the false branch of L7 in the trace slice.

Third, if both instructions are branch instructions, the algorithm checks whether the not-taken branches of the instructions may race, and if so, it adds either taken branch to `slicing_targets`. For instance, to avoid the race in Figure 4.9(b), PEREGRINE includes one of the false branches in the trace slice.

For efficiency, PEREGRINE avoids iterating through all pairs of instructions from two concurrent regions because instructions in one region often repeatedly access the same memory locations. Thus, PEREGRINE computes memory locations read or written by all instructions in one region, then checks whether instructions in the other region also read or write these memory locations. These locations are static operands, not dynamic addresses [22], so that PEREGRINE can aggressively cache them per static function or branch. The complexity of our algorithm thus drops from $O(MN)$ to $O(M+N)$ where M and N are the numbers of memory instructions in the two regions respectively.

4.4.2 Intra-thread Step

In the intra-thread step, PEREGRINE leverages an algorithm [30] to compute a per-thread path slice, by including instructions required for the thread to reach the `slicing_targets` identified in the inter-thread step and the events in the hybrid schedule. To do so, PEREGRINE first prepares a per-thread ordered target list by splitting `slicing_targets` and events in the hybrid schedule and sorting them based on their order in the execution trace.

PEREGRINE then traverses the execution trace backwards to compute path slices. When it sees a target, it adds the target to the path slice of the corresponding thread, and starts to track the control- and data-dependencies of this target.[1] PEREGRINE adds a branch instruction to the path slice if taking the opposite branch may cause the thread not to reach any instruction in the current (partial) path slice; L3 in Figure 4.4 is added for this reason. It adds a non-branch instruction to

[1]For readers familiar with precondition slicing, PEREGRINE does not always track data-dependencies for the operands of a target. For instance, consider instruction L9 of thread t_0 in Figure 4.4. PEREGRINE's goal is to deterministically resolve the race involving L9 of t_0, but it allows the value of `result` to be different. Thus, PEREGRINE does not track dependencies for the value of `result`, and L15 of t_0 is elided.

the path slice if the result of this instruction may be used by instructions in the current path slice; L1 in Figure 4.4 is added for this reason.

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

Our main modification to [30] is to slice toward multiple ordered targets. To illustrate this need, consider branch L4:false of t_0 in Figure 4.4. PEREGRINE must add this branch to thread t_0's slice, because otherwise, the thread would reach another `pthread_create()`, a different synchronization operation than the `pthread_mutex_lock()` operation in the schedule.

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

4.5 Schedule-Guided Simplification

In both the inter- and intra-thread steps of determinism-preserving slicing, PEREGRINE frequently queries alias analysis. The inter-thread step needs alias information to determine whether two instructions may race (`mayalias()` in Figure 4.10). The intra-thread step needs alias information to track potential dependencies.

We thus integrated `bddbddb` [112, 113], one of the best alias analyses, into PEREGRINE by creating an LLVM frontend to collect program facts into the format `bddbddb` expects. However, our initial evaluation showed that `bddbddb` sometimes yielded overly imprecise results, causing PEREGRINE to prune few branches, reducing schedule-reuse rates (§4.7.3). The cause of the imprecision is that standard alias analysis is purely static, and has to be conservative and assume all possible executions. However, PEREGRINE requires alias results only for the executions that may

reuse a schedule, thus suffering from unnecessary imprecision of standard alias analysis.

To illustrate, consider the example in Figure 4.3. Since the number of threads is determined at runtime, static analysis has to abstract this unknown number of dynamic thread instances, often coalescing results for multiple threads into one. When PEREGRINE slices the trace in Figure 4.4, `bddbddb` reports that the accesses to `data` (L13 instances) in different threads may alias. PEREGRINE thus has to add them to the trace slice to avoid new races (§4.4.1). Since L13 depends on L12, L11, and L10, PEREGRINE has to add them to the trace slice, too. Eventually, an imprecise alias result snowballs into a slice as large as the trace itself. The preconditions from this slice constrains the data size to be exactly 2, so PEREGRINE cannot reuse the hybrid schedule in Figure 4.4 on other data sizes.

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

1. It clones the functions as needed. For instance, it gives each thread in a schedule a copy of the thread function.

2. It unrolls a loop when it can determine the loop bound based on a schedule. For instance, from the number of the `pthread_create()` operations in a schedule, it can determine how many times the loop at lines L4–L5 in Figure 4.3 executes.

3. It removes branches that contradict the schedule. Loop unrolling can be viewed as a special case of this simplification.

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

different thread instances (*thread-sensitivity* in programing language terms) and precisely reports that L13 of t_0 and L13 of t_1 are not aliases, enabling PEREGRINE to compute the small trace slice in Figure 4.4.

By simplifying a program, PEREGRINE can automatically improve the precision of not only alias analysis, but also other analyses. We have implemented *range analysis* [96] to improve the precision of alias analysis on programs that divide a global array into disjoint partitions, then process each partition within a thread. The accesses to these disjoint partitions from different threads do not alias, but `bddbddb` often collapses the elements of an array into one or two abstract locations, and reports the accesses as aliases. Range analysis can solve this problem by tracking the lower and upper bounds of the integers and pointers. With range analysis, PEREGRINE answers alias queries as follows. Given two pointers `(p+i)` and `(q+i)`, it first queries `bddbddb` whether `p` and `q` may alias. If so, it queries the more expensive range analysis whether `p+i` and `q+j` may be equal. It considers the pointers as aliases only when both queries are true. Note that our simplification technique is again key to precision because standard range analysis would merge ranges of different threads into one.

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

4.6 Implementation Issues

This section discusses implementation issues not covered by previous sections.

4.6.1 Recording an Execution

To record an execution trace, PEREGRINE can use one of the existing deterministic record-replay systems [18, 36, 62] provided that PEREGRINE can extract an instruction trace. For simplicity, we have built a crude recorder on top of the LLVM interpreter in KLEE. When an program calls the PEREGRINE-provided wrapper to `pthread_create(..., func, args)`, the recorder spawns a

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

4.6.2 Handling Blocking System Calls

Blocking system calls are natural scheduling points, so PEREGRINE includes them in the schedules [31]. It currently considers eight blocking system calls, such as `sleep()`, `accept()`, and `read()`. For each blocking system call, the recorder logs when the call is issued and when the call is returned. When PEREGRINE computes a schedule, it includes these blocking system call and return operations. When reusing a schedule, PEREGRINE attempts to enforce the same call and return order. This method works well for blocking system calls that access local state, such as `sleep()` or `read()` on local file descriptors. However, other blocking system calls receive input from the external world, which may or may not arrive each time a schedule is reused. Fortunately, programs that use these operations tend to be server programs, and PEREGRINE handles this class of programs differently.

4.6.3 Handling Server Programs

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

PEREGRINE addresses these challenges with the *windowing* idea from our previous work [31]. The insight is that server programs tend to return to the same quiescent states. Thus, instead of processing requests as they arrive, PEREGRINE breaks a continuous request stream down into windows of requests. Within each window, it admits requests only at fixed points in the current schedule. If no requests arrive at an admission point for a predefined timeout, PEREGRINE simply proceeds with the partial window. While a window is running, PEREGRINE buffers newly arrived requests so that they do not interfere with the running window. With windowing, PEREGRINE can

record and reuse schedules across windows.

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

One limitation of applying our PEREGRINE prototype to server programs is that our current implementation of schedule-guided simplification does not work well with thread pooling. To give each thread a distinct thread function, PEREGRINE identifies `pthread_create(...,func,...)` operations in a program and clones function `func`. Server programs that use thread pooling tend to create worker threads to run generic thread functions during program initialization, then repeatedly use the threads to process client requests. Cloning these generic thread functions thus helps little with precision. One method to solve this problem is to clone the relevant functions for processing client requests. We have not implemented this method because the programs we evaluated include only one server program, `Apache`, on which slicing already performs reasonably well without simplification (§4.7.3).

4.6.4 Skipping Wait Operations

When reusing a schedule, PEREGRINE enforces a total order of synchronization operations, which subsumes the execution order enforced by the original synchronization operations. Thus, for speed, PEREGRINE can skip the original synchronization operations as in [31]. PEREGRINE skips sleep-related operations such as `sleep()` and wait-related operations such as `pthread_barrier_wait()`. These operations often unconditionally block the calling thread, incurring context switch overhead, yet this blocking is unnecessary as PEREGRINE already enforces a correct execution order. Our evaluation shows that skipping blocking operations significantly speeds up executions.

4.6.5 Manual Annotations

PEREGRINE works automatically for most of the programs we evaluated. However, as discussed in §4.6.3, it requires manual annotations for server programs. In addition, if a program has non-determinism sources beyond what PEREGRINE automatically tracks, developers should annotate these sources with `input(void* addr, size_t nbyte)` to mark `nbyte` of data starting from `addr` as input, so that PEREGRINE can track this data.

Developers can also supply optional annotations to improve PEREGRINE's precision in four ways. First, for better alias results, developers can add custom memory allocators and `memcpy`-like functions to a configuration file of PEREGRINE. Second, they can help PEREGRINE better track ranges by adding `assert()` statements. For instance, a function in the FFT implementation we evaluated uses bit-flip operations to transform an array index into another, yet both indexes have the same range. The range analysis we implemented cannot precisely track these bit-flip operations, so it assumes the resultant index is unbounded. Developers can fix this problem by annotating the range of the index with an assertion "`assert(index<bound)`." Third, they can provide *symbolic summaries* to help PEREGRINE compute more relaxed constraints. For instance, consider Figure 4.5 and a typical implementation of `atoi()` that iterates through all characters in the input string and checks whether each character is a digit. Without a summary of `atoi()`, PEREGRINE would symbolically execute the body of `atoi()`. The preconditions it computes for `argv[3]` would be $(argv_{3,0} \neq 49) \wedge (argv_{3,1} < 48 \vee argv_{3,1} > 57)$, where $argv_{3,i}$ is the ith byte of `argv[3]` and 48, 49, and 57 are ASCII codes of '0', '1', and '9'. These preconditions thus unnecessarily constrain `argv[3]` to have a valid length of one. Another example is string search. When a program calls `strstr()`, it often concerns whether there exists a match, not specifically where the match occurs. Without a symbolic summary of `strstr()`, the preconditions from `strstr()` would constrain the exact location where the match occurs. Similarly, if a trace slice contains complex code such as a decryption function, users can provide a summary of this function to mark the decrypted data as symbolic when the argument is symbolic. Note that complex code not included in trace slices, such as the `read()` in Figure 4.3, is not an issue.

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

Table 4.1: *Programs used for evaluating* PEREGRINE*'s determinism.*

4.7 Evaluation

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

We evaluated our PEREGRINE implementation on a diverse set of 18 programs, including `Apache`, a popular web server; `PBZip2`, a parallel compression utility; `aget`, a parallel `wget`-like utility; `pfscan`, a parallel `grep`-like utility; parallel implementations of 13 computation-intensive algorithms, 10 in SPLASH-2 and 3 in PARSEC; and `racey`, a benchmark specifically designed to exercise deterministic execution and replay systems [50]. All SPLASH-2 benchmarks were included except one that we cannot compile, one that our current prototype cannot handle due to an implementation bug, and one that does not run correctly in 64-bit environment. The chosen PARSEC benchmarks (`blackscholes`, `swaptions` and `streamcluster`) include the ones that (1) we can compile, (2) use threads, and (3) use no x86 inline assemblies. These programs were widely used in previous studies (e.g., [16, 71, 116]).

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

Unless otherwise specified, we used the following workloads in our experiments. For `Apache`, we used `ApacheBench` [5] to repeatedly download a 100 KB webpage. For `PBZip2`, we compressed a 10 MB randomly generated text file. For `aget`, we downloaded a 77 MB file (`Linux-3.0.1.tar.bz2`). For `pfscan`, we scanned the keyword `return` from 100 randomly chosen files in GCC. For SPLASH-2 and PARSEC programs, we ran workloads which typically completed in 1-100 ms.

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

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

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

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

§4.7.4: Can PEREGRINE significantly reduce manual annotation overhead? Recall that our previous work [31] required developers to manually annotate the input affecting schedules.

4.7.1 Determinism

We evaluated PEREGRINE's determinism by checking whether PEREGRINE could deterministically resolve races. Table 4.1 lists the seven racy programs used in this experiment. We selected the first five because they were frequently used in previous studies [69, 71, 86, 87] and we could reproduce their races on our evaluation machine. We selected the integer flag race in PARSEC to test whether PEREGRINE can handle ad hoc synchronization [116]. We selected `racey` to stress test PEREGRINE:

Program	Races	Order Constraints
Apache	0	0
PBZip2	4	3
barnes	5	1
fft	10	4
lu_ncb	10	7
streamcluster	0	0
racey	167974	9963

Table 4.2: *Hybrid schedule statistics.* Column **Races** shows the number of races detected according the corresponding sync-schedule, and Column **Order Constraints** shows the number of execution order constraints PEREGRINE adds to the final hybrid schedule. The latter can be smaller than the former because PEREGRINE prunes subsumed execution order constraints (§4.3). PEREGRINE detected no races for `Apache` and `streamcluster` because the corresponding sync-schedules are sufficient to resolve the races deterministically; it thus adds no order constraints for these programs.

each run of `racey` may have thousands of races, and if any of these races is resolved differently, `racey`'s final output changes with high probability [50].

For each program with races, we recorded an execution trace and computed a hybrid schedule from the trace. Table 4.2 shows for each program (1) the number of dynamic races detected according to the sync-schedule and (2) the number of execution order constraints in the hybrid schedule. The reduction from the former to the latter shows how effectively PEREGRINE can prune redundant order constraints (§4.3). In particular, PEREGRINE prunes 94% of the constraints for `racey`. For `Apache` and `streamcluster`, their races are already resolved deterministically by their sync-schedules, so PEREGRINE adds no execution order constraints.

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

Program	Deterministic?	
	sync-schedule	hybrid schedule
Apache	✔	✔
PBZip2	✖	✔
barnes	✖	✔
fft	✖	✔
lu_ncb	✖	✔
streamcluster	✔	✔
racey	✖	✔

Table 4.3: *Determinism of sync-schedules v.s. hybrid schedules.*

We also compared the determinism of PEREGRINE to our previous work [31] which only enforces sync-schedules. Specifically, we reran the seven programs with races 50 times enforcing only the sync-schedules and injecting delays, and checked whether the reuse runs computed equivalent results as the recorded run. As shown in Table 4.3, sync-schedules are unsurprisingly deterministic for `Apache` and `streamcluster`, because no races are detected according to the corresponding sync-schedules. However, they are not deterministic for the other five programs, illustrating one advantage of PEREGRINE over the sync-schedule approach.

4.7.2 Efficiency

Replayer overhead. The most performance-critical component is the replayer because it operates within a deployed program. Figure 4.11 shows the execution times when reusing hybrid schedules; these times are normalized to the nondeterministic execution time. (The next paragraph compares these times to those of sync-schedules.) For `Apache`, we show the throughput (TPUT) and response time (RESP). All numbers reported were averaged over 500 runs. PEREGRINE has relatively high overhead on `water-nsquared` (22.6%) and `cholesky` (46.6%) because these programs do a large number of mutex operations within tight loops. Still, this overhead is lower than the reported 1.2X-6X overhead of a mem-schedule DMT system [12]. Moreover, PEREGRINE speeds up `barnes`, `lu_ncb`, `radix`, `water-spatial`, and `ocean` (by up to 68.7%) because it safely skips synchronization and sleep operations (§4.6.4). For the other programs, PEREGRINE's overhead or speedup is within 15%. (Note that increasing the page or file sizes of the workload tends to reduce PEREGRINE's

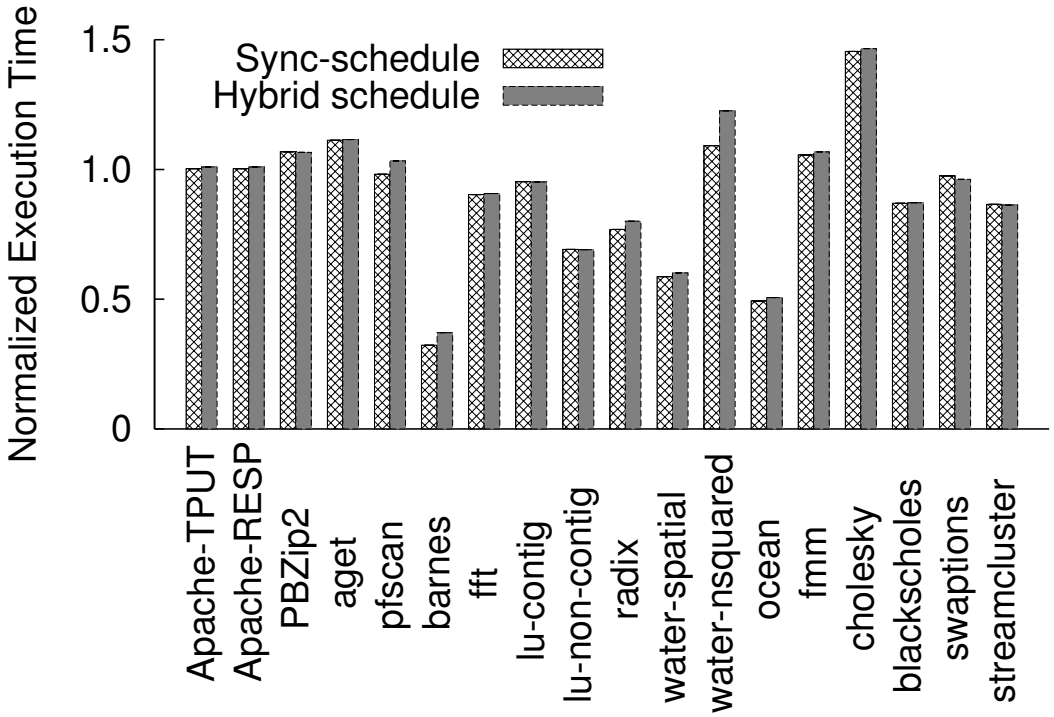


Figure 4.11: *Normalized execution time when reusing sync-schedules v.s. hybrid schedules.* A time value greater than 1 indicates a slowdown compared to a nondeterministic execution without PEREGRINE. We did not include `racey` because it was not designed for performance benchmarking.

relative overhead because the network and disk latencies dwarf PEREGRINE's.)

For comparison, Figure 4.11 shows the normalized execution time when enforcing just the sync-schedules. This overhead is comparable to our previous work [31]. For all programs except `water-nsquared`, the overhead of enforcing hybrid schedules is only slightly larger (at most 5.4%) than that of enforcing sync-schedules. This slight increase comes from two sources: (1) PEREGRINE has to enforce execution order constraints to resolve races deterministically for `PBZip2`, `barnes`, `fft`, and `lu_ncb`; and (2) the instrumentation framework PEREGRINE uses also incurs overhead (§4.3.2). The overhead for `water-nsquared` increases by 13.4% because it calls functions more frequently than the other benchmarks, and our instrumentation framework inserts code at each function entry and return (§4.3.2).

Figure 4.12 shows the speedup of flag relay (§4.3.2) and skipping blocking operations (§4.6.4). Besides `water-nsquared` and `cholesky`, a second group of programs, including `barnes`, `lu_ncb`,

`radix`, `water-spatial`, and `ocean`, also perform many synchronization operations, so flag relay speeds up both groups of programs significantly. Moreover, among the synchronization operations done by the second group of programs, many are `pthread_barrier_wait()` operations, so PEREGRINE further speeds up these programs by skipping these wait operations.

Analyzer and recorder overhead. Table 4.4 shows the execution time of PEREGRINE's various program analyses. The execution time largely depends on the size of the execution trace. All analyses typically finish within a few hours. For `PBZip2` and `fft`, we used small workloads (compressing 1 KB file and transforming a 256X256 matrix) to reduce analysis time and to illustrate that the schedules learned from small workloads can be efficiently reused on large workloads. The simplification and alias analysis time of `fft` is large compared to its slicing time because it performs many multiplications on array indexes, slowing down our range analysis. Although `lu_ncb`

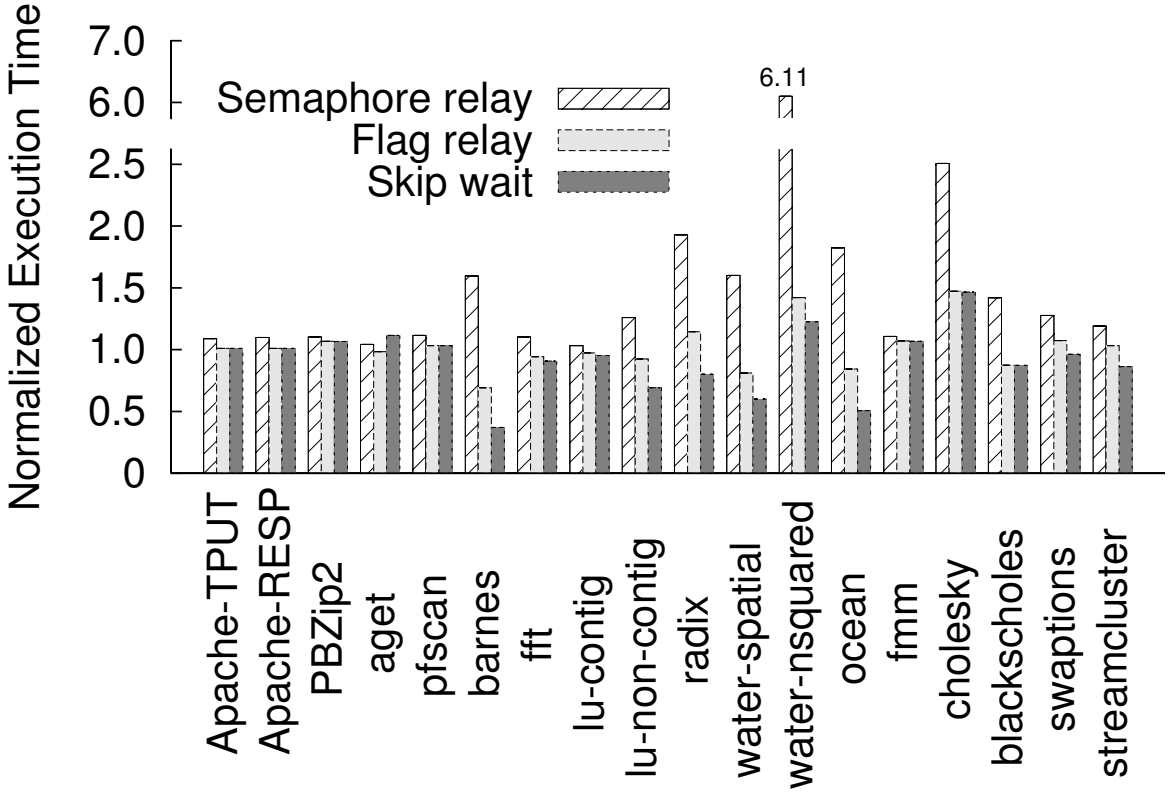


Figure 4.12: *Speedup of optimization techniques.* Note that Y axis is broken.

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

Table 4.4: *Analysis time.* **Trace** shows the number of thousand LLVM instructions in the execution trace of the evaluated programs, the main factor affecting the execution time of PEREGRINE's various analysis techniques, including race detection (**Det**), slicing (**Sli**), simplification and alias analysis (**Sim**), and symbolic execution (**Sym**). The execution time is measured in seconds. The `Apache` trace is collected from one window of eight requests. `Apache` uses thread pooling which our simplification technique currently does not handle well (§4.6.3); nonetheless, slicing without simplification works reasonably well for `Apache` already (§4.7.3).

and `lu_cb` implement the same scientific algorithm, their data access patterns are very different (§4.7.3), causing PEREGRINE to spend more time analyzing `lu_ncb` than `lu_cb`.

As discussed in §4.6.1, PEREGRINE currently runs KLEE to record executions. Column Sym is also the overhead of PEREGRINE's recorder. This crude, unoptimized recorder can incur large slowdown compared to the normal execution of a program. However, this slowdown can be reduced to around 10X using existing record-replay techniques [18, 62]. Indeed, we have experimented with a preliminary version of a new recorder that records an execution by instrumenting `load` and štore instructions and saving them into per-thread logs [18]. Figure 4.13 shows that this new recorder incurs roughly 2-35X slowdown on eight programs, comparable to existing record-replay systems.

Due to time constraints, we have not integrated this new recorder with PEREGRINE.

4.7.3 Stability

Stability measures how frequently PEREGRINE can reuse schedules. The more frequently PEREGRINE reuses schedules, the more efficient it is, and the more repeatable a program running on top of PEREGRINE becomes. While PEREGRINE achieves determinism and efficiency through hybrid schedules, it may have to pay the cost of slightly reduced reuse rates compared to a manual approach [31].

A key factor determining PEREGRINE's schedule-reuse rates is how effectively it can slice out irrelevant instructions from the execution traces. Figure 4.14 shows the ratio of the slice size over the trace size for PEREGRINE's determinism-preserving slicing technique, with and without schedule-guided simplification. The slicing technique alone reduces the trace size by over 50% for all programs except `PBZip2`, `aget`, `pfscan`, `fft`, `lu_ncb`, `ocean`, and `swaptions`. The slicing technique combined with scheduled-guide simplification vastly reduces the trace size for `PBZip2`, `aget`, `fft`, `lu_cb`, and `swaptions`.

Recall that PEREGRINE computes the preconditions of a schedule from the input-dependent branches in a trace slice. The fewer branches included in the slice, the more general the preconditions PEREGRINE computes tend to be. We further measured the number of such branches in

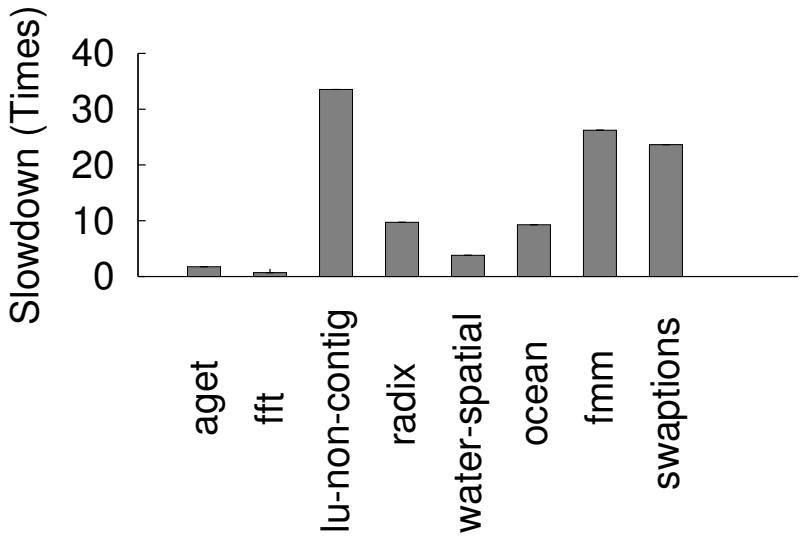


Figure 4.13: *Overhead of recording* `load` *and* štore *instructions.*

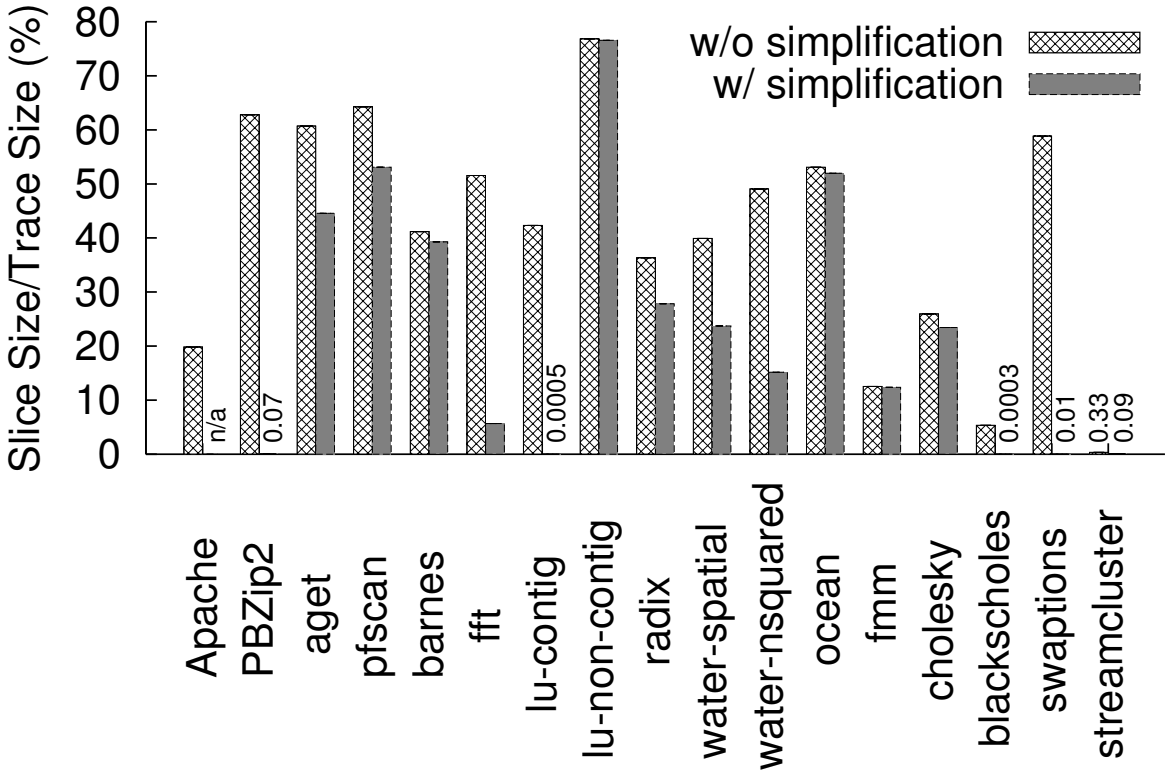


Figure 4.14: *Slicing ratio after applying determinism-preserving slicing alone (§4.4) and after further applying schedule-guided simplification (§4.5).*

the trace slices. Table 4.5 shows the results, together with a upper bound determined by the total number of input-dependent branches in the execution trace, and a lower bound determined by only including branches required to reach the recorded synchronization operations. This lower bound may not be tight as we ignored data dependency. For `barnes`, `fft`, `blackscholes`, `swaptions`, and `streamcluster`, slicing with simplification (Column "Slicing+Sim") achieves the best possible reduction. For `PBZip2`, `aget`, `pfscan`, and `lu_cb`, the number of input-dependent branches in the trace slice is close to the lower bound. In the remaining programs, `Apache`, `fmm`, and `cholesky` also enjoy large reduction, while the other five programs do not. This table also shows that schedule-guided simplification is key to reduce the number of input-dependent branches for `PBZip2`, `fft`, `lu_cb`, `blackscholes`, and `swaptions`, and to reach the lower bound for `blackscholes`, `swaptions`, and `streamcluster`.

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

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

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

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

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

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

4.7.4 Ease of Use

Table 4.6 shows the annotations (§4.6.5) we added to make the evaluated programs work with PEREGRINE. For most programs, PEREGRINE works out of the box. `Apache` uses its own library

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

Table 4.5: *Effectiveness of program analysis techniques.* **UB** shows the total number of input-dependent branches in the corresponding execution trace, an upper bound on the number included in the trace slice. **Slicing** and **Slicing+Sim** show the number of input-dependent branches in the slice after applying determinism-preserving slicing alone (§4.4) and after further applying schedule-guided simplification (§4.5). **LB** shows a lower bound on the number of input-dependent branches, determined by only including branches required to reach the recorded synchronization operations. This lower bound may not be tight as we ignored data dependency when computing it.

functions for common tasks such as memory allocation, so we annotated 21 such functions. We added two annotations to mark the boundaries of client request processing and one to expose the hidden state in `Apache` (§4.6.3). `PBZip2` decompression uses a custom search function (`memstr`) to scan through the input file for block boundaries. We added one annotation for this function to relax the preconditions Peregrine computes. (Peregrine works automatically with `PBZip2`

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

Table 4.6: *Source annotation requirements of* PEREGRINE *v.s.* TERN. **Peregrine** represents the number of annotations added for PEREGRINE, and **Tern** counts annotations added for TERN. Programs not included in the TERN evaluation are labeled n/a. LOC of `PBZip2` also includes the lines of code of the compression library `libbz2`.

compression.) We added one assertion to annotate the range of a variable in `fft` (§4.6.5).

For comparison, Table 4.6 also shows the annotation overhead of our previous DMT system TERN [31]. For all programs except `Apache`, PEREGRINE has fewer number of annotations than TERN. Although the number of annotations that TERN has is also small, adding these annotations may require developers to manually reconstruct the control- and data-dependencies between instructions.

In order to make the evaluated programs work with PEREGRINE, we had to fix several bugs in them. For `aget`, we fixed an off-by-one write in `revstr()` which prevented us from track-

ing constraints for the problematic write, and a missing check on the return value of `pwrite()` which prevented us from computing precise ranges. We fixed similar missing checks in `swaptions`, `streamcluster`, and `radix`. We did not count these modifications in Table 4.6 because they are real bug fixes. (This interesting side-effect illustrates the potential of PEREGRINE as an error detection tool: the precision gained from simplification enables PEREGRINE to detect real races in well-studied programs.)

4.8 Related Work

StableMT and DMT systems. PEREGRINE stabilizes program behaviors over input perturbations by reusing schedules. This method is based on the *schedule memoization* idea in our previous system TERN (Chapter 3), but PEREGRINE largely eliminates manual annotations, and provides stronger determinism guarantees than TERN.

PEREGRINE is complementary to other StableMT and DMT systems [8, 12, 31, 34, 66, 80]: PEREGRINE can use an existing StableMT or DMT algorithm when it runs a program on a new input so that it may compute the same schedules at different sites; existing StableMT or DMT systems can speed up their pathological cases using the schedule-relaxation idea.

Determinator [8] advocates a new, radical programming model that converts all races, including races on memory and other shared resources, into exceptions, to achieve pervasive determinism. This programming model is not designed to be backward-compatible. dOS [13] provides similar pervasive determinism with backward compatibility, using a DMT algorithm first proposed in [34] to enforce mem-schedules. While PEREGRINE currently focuses on multithreaded programs, the ideas in PEREGRINE can be applied to other shared resources to provide pervasive determinism. PEREGRINE's hybrid schedule idea may help reduce dOS's overhead. Grace [16] makes multithreaded programs with fork-join parallelism behave like sequential programs. It detects memory access conflicts efficiently using hardware page protection. Unlike Grace, PEREGRINE aims to make general multithreaded programs, not just fork-join programs, repeatable.

Concurrent to our work, DTHREADS [66] is another efficient multithreading system that is both stable and deterministic. It tracks memory modifications using hardware page protection and provides a protocol to deterministically commit these modifications. In contrast to DTHREADS,

PEREGRINE is software-only and does not rely on page protection hardware which may be expensive and suffer from false sharing; PEREGRINE records and reuses schedules, thus it can handle programs with ad hoc synchronizations [116] and make program behaviors stable.

Program analysis. Program slicing [107] is a general technique to prune irrelevant statements from a program or trace. Recently, systems researchers have leveraged or invented slicing techniques to block malicious input [30], synthesize executions for better error diagnosis [125], infer source code paths from log messages for postmortem analysis [124], and identify critical inter-thread reads that may lead to concurrency errors [127]. Our determinism-preserving slicing technique produces a correct trace slice for multithreaded programs and supports multiple ordered targets. It thus has the potential to benefit existing systems that use slicing.

Our schedule-guided simplification technique shares similarity with SherLog [124] such as the removal of branches contradicting a schedule. However, SherLog starts from log messages and tries to compute an execution trace, whereas PEREGRINE starts with a schedule and an execution trace and computes a simplified yet runnable program. PEREGRINE can thus transparently improve the precision of many existing analyses: simply run them on the simplified program.

Replay and re-execution. Deterministic replay [4, 35, 36, 42, 47, 61, 62, 74, 87, 104, 108] aims to replay the exact recorded executions, whereas PEREGRINE "replays" schedules on different inputs. Some recent deterministic replay systems include Scribe, which tracks page ownership to enforce deterministic memory access [62]; Capo, which defines a novel software-hardware interface and a set of abstractions for efficient replay [74]; PRES and ODR, which systematically search for a complete execution based on a partial one [4, 87]; SMP-ReVirt, which uses page protection for recording the order of conflicting memory accesses [36]; and Respec [64], which uses online replay to keep multiple replicas of a multithreaded program in sync. Several systems [64, 87] share the same insight as PEREGRINE: although many programs have races, these races tend to occur infrequently.

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

Retro [58] shares some similarity with PEREGRINE because it also supports "mutated" replay. When repairing a compromised system, Retro can replay legal actions while removing malicious ones using a novel dependency graph and *predicates* to detect when changes to an object need not be propagated further. PEREGRINE's determinism-preserving slicing algorithm may be used

to automatically compute these predicates, so that Retro does not have to rely on programmer annotations.

Concurrency errors. The complexity in developing multithreaded programs has led to many concurrency errors [71]. Much work exists on concurrency error detection, diagnosis, and correction (e.g., [38, 40, 41, 70, 86, 123, 126, 127]). PEREGRINE aims to make the executions of multithreaded programs repeatable, and is complementary to existing work on concurrency errors. PEREGRINE may use existing work to detect and fix the errors in the schedules it computes. Even for programs free of concurrency errors, PEREGRINE still provides value by making their behaviors repeatable.

4.9 Summary

PEREGRINE is one of the first stable and fully deterministic multithreading system with good efficiency. Leveraging the insight that races are rare, PEREGRINE combines sync-schedules and mem-schedules into hybrid schedules, getting the benefits of both. PEREGRINE reuses schedules across different inputs, amortizing the cost of computing hybrid schedules and making program behaviors repeatable across inputs. It further improves efficiency using two new techniques: determinism-preserving slicing to generalize a schedule to more inputs while preserving determinism, and schedule-guided simplification to precisely analyze a program according to a dynamic schedule. Our evaluation on a diverse set of programs shows that PEREGRINE is both deterministic and efficient, and can frequently reuse schedules for half of the evaluated programs.

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

Chapter 5

Making StableMT Simple, Fast, and Deployable

The last two chapters have presented our two systems, Tern and Peregrine, with each addressing a distinct challenge on building StableMT. Other researchers have also been building and applying StableMT systems [8, 15, 66] to improve software reliability. However, despite these latest advances, existing StableMT systems either require sophisticated program analysis (e.g., our two previous systems), or incur prohibitive performance overhead (e.g., a previous system [66] incurred 30× slowdown with many programs in our evaluation in §5.7), causing StableMT difficult to be widely deployed. Thus, a third challenge on building StableMT arises: how to make StableMT simple, fast, and deployable? To address this challenge, this chapter presents Parrot, a simple, deployable StableMT runtime system, and its novel programming abstraction called *performance hints* that make Parrot's schedules run fast.

5.1 Introduction

As described in Chapter 2, a root cause that makes multithreading extremely difficult to get right is that, for decades, the contract between developers and thread runtimes has favored performance over correctness and grants exponentially many possible schedules for all inputs. In this contract, developers use synchronizations to coordinate threads, while thread runtimes can use *any* of the exponentially many schedules, compliant with the synchronizations. This large number of possible

schedules make it more likely to find an efficient schedule for a workload, but ensuring that all schedules are free of concurrency bugs is extremely challenging, and a single unchecked schedule may surface in the least expected moment, causing critical failures and vulnerabilities [65, 71, 92, 121].

Two recent techniques aim to flip this performance vs. correctness tradeoff by reducing the number of allowed schedules. First, StableMT [8, 31, 32, 66], which is created by my collaborators and me, aims to reduce the number possibles schedules on all inputs. Other researchers have also been building and applying StableMT systems [8, 15, 66] to improve software reliability. These work have shown to greatly improve the reliability of multithreaded programs, including: (1) making reproducing concurrency bugs much easier [31, 32], (2) improving the precision of program analysis [32, 115], leading to the detection of several new harmful data races in heavily-tested programs, and (3) computing a small set of schedules to cover all or most inputs [15]. Second, DMT [12, 13, 16, 34, 80] addresses the nondeterminism problem, and it focuses on reducing the number of possible schedules on each input down to one. DMT is especially useful in testing and debugging multithreaded programs, however, we have previously stated in Chapter 2 that DMT is not as useful as commonly perceived, and StableMT is better for improving reliability of multithreaded programs. StableMT is complementary to DMT, and several multithreading systems (e.g., [8, 31, 32, 66]) are both stable and deterministic.

However, despite these recent advances, it remains an open challenge that whether StableMT can be made simple, fast, and deployable on a wide range of multithreaded programs. This challenge is not helped much by the limited evaluation of previous systems which often used (1) synthetic benchmarks, not real-world programs, from incomplete benchmark suites; (2) one workload per program; and (3) at most 8 cores (with three exceptions; see §5.8). For instance, while a previous system DThreads [66] achieves reasonable performance overhead on 14 scientific benchmark programs, we observed that this system incurred 30× slowdown with many other programs in our evaluation (§5.7).

This open challenge comes from the design choices of existing StableMT systems. Reducing schedules improves correctness but trades performance because the schedules left may not balance each thread's load well, causing some threads to idle unnecessarily. Our experiments show that ignoring load imbalance as in DThreads can lead to pathological slowdown if the order of operations enforced by a schedule *serializes* the intended parallel computations (§5.7.3). To recover

performance, one method is to count the instructions executed by each thread and select schedules that balance the instruction counts [12, 34, 80], but this method is not stable because input or program perturbations easily change the instruction counts. The other method (we proposed) lets the nondeterministic OS scheduler select a reasonably fast schedule and reuses the schedule on compatible inputs [31, 32], but it requires sophisticated program analysis, complicating deployment.

To tackle this open challenge, this chapter presents Parrot, our third StableMT (and also DMT) system, with three contributions. First, Parrot is a simple, practical runtime that efficiently makes threads deterministic and stable by offering a new contract to developers. By default, it schedules synchronizations in each thread using round-robin, vastly reducing schedules and providing broad repeatability. When default schedules are slow, it allows advanced developers to add intuitive *performance hints* to their code for speed. Developers discover where to add hints through profiling as usual, and Parrot simplifies performance debugging by deterministically reproducing the bottlenecks. The hints are robust to developer mistakes as they can be safely ignored without affecting correctness. Like previous systems, Parrot's contract reduces schedules to favor correctness over performance. Unlike previous systems, it allows advanced developers to optimize performance. We believe this practical "meet in the middle" contract eases writing correct, efficient programs. For this reason, we name this system Parrot, one of the most trainable birds.

Parrot provides two performance hint abstractions. A *soft barrier* encourages the scheduler to coschedule a group of threads at given program points. It is for performance only, and operates as a barrier with deterministic timeouts in Parrot. Developers use it to switch to faster schedules without compromising determinism when the default schedules serialize parallel computations (§5.2.1). A *performance critical section* informs the scheduler that a code region is a potential bottleneck, encouraging the scheduler to get through the region fast. When a thread enters a performance critical section, Parrot delegates scheduling to the nondeterministic OS scheduler for speed. Performance critical sections may trade some determinism for performance, so they should be applied only when the schedules they add are thoroughly checked by tools or advanced developers. These simple abstractions let Parrot run fast on all programs evaluated, and may benefit other DMT or StableMT systems and classic nondeterministic schedulers [3, 37, 83].

Our Parrot implementation is Pthreads-compatible, simplifying deployment. It handles many diverse constructs real-world programs depend upon such as network operations and timeouts.

PARROT makes synchronizations outside performance critical sections deterministic but allows nondeterministic data races. Although it is straightforward to make data races deterministic in PARROT, we deemed it not worthwhile because the cost of doing so outweighs the benefits (§5.6). PARROT's determinism is similar to Kendo's weak determinism [80], but PARROT offers stability which Kendo lacks.

Our second contribution is an ecosystem formed by integrating PARROT with DBUG [101], an open source model checker for distributed and multithreaded Linux programs that systematically checks possible schedules for bugs. This PARROT-DBUG ecosystem is more effective than either system alone: DBUG checks the schedules that matter to PARROT and developers (e.g., schedules added by performance critical sections), and PARROT greatly increases DBUG's coverage by reducing the schedules DBUG has to check (the *state space*). Our integration is transparent to DBUG and requires only minor changes to PARROT. It lets PARROT effectively leverage advanced model checking techniques [39, 48].

Third, we quantitatively show that PARROT achieves good performance and high model checking coverage on a diverse set of 108 programs. The programs include 55 real-world programs, such as Berkeley DB [17], OpenLDAP [82], Redis [94], MPlayer [75], all 33 parallel C++ STL algorithm implementations [105] which use OpenMP, and all 14 parallel image processing utilities (also OpenMP) in the ImageMagick [52] software suite. Further, they include *all* 53 programs in four widely used benchmark suites: PARSEC [88], Phoenix [93], SPLASH-2x [102], and NPB [79]. We used complete software or benchmark suites to avoid biasing our results. The programs together cover many different parallel programming models and idioms such as threads, OpenMP [20], fork-join, map-reduce, pipeline, and workpile. To our knowledge, our evaluation uses roughly 10× more programs than any previous DMT or StableMT evaluation, and 4× more than all previous evaluations combined. Our experiments show:

1. PARROT is easy to use. It averages only 1.2 lines of hints per program to get good performance, and adding hints is fast. Of all 108 programs, 18 need no hints, 81 need soft barriers which do not affect determinism, and only 9 programs need performance critical sections to trade some determinism for speed.

2. PARROT has low overhead. At the maximum allowed (16–24) cores, PARROT's geometric mean overhead is 6.9% for 55 real-world programs, 19.0% for the other 53 programs, and

12.7% for all.

3. On 25 programs that two previous systems DThreads [66] and CoreDet [12] can both handle, Parrot's overhead is 11.8% whereas DThreads's is 150.0% and CoreDet's 115.1%.

4. Parrot scales well to the maximum allowed cores on our 24-core server and to at least three different scales/types of workloads per program.

5. Parrot-dbug offers exponential coverage increase compared to dbug alone. Parrot helps dbug reduce the state space by 10^6–10^{19734} for 56 programs and increase the number of verified programs from 43 to 99 under our test setups. These verified programs include all 4 real-world programs out of the 9 programs that need performance critical sections, so they enjoy both speed and reliability. These quantitative reliability results help potential Parrot adopters justify the overhead.

We have released Parrot's source code, entire benchmark suite, and raw evaluation results at `github.com/columbia/smt-mc`. In the remaining of this chapter, §5.2 contrasts Parrot with previous systems on an example and gives a high-level design of Parrot. §5.3 describes the performance hint abstractions Parrot provides, §5.4 the Parrot runtime, and §5.5 the Parrot-dbug ecosystem. §5.6 discusses Parrot's determinism, §5.7 presents evaluation results, §5.8 discusses related work, and §5.9 concludes.

5.2 High-level Design

This section first compares two previous systems and Parrot using an example (§5.2.1), and then describes Parrot's architecture design (§5.2.2).

5.2.1 An Example

Figure 5.1 shows the example, a simplified version of the parallel compression utility `PBZip2` [89]. It uses the common producer-consumer idiom: the producer (main) thread reads file blocks, and multiple consumer threads compress them in parallel. Once the number of threads and the number of blocks are given, one synchronization schedule suffices to compress *any* file, regardless of file content or size. Thus, this program appears easy to make deterministic and stable. However,

```
1 : int main(int argc, char *argv[]) {
2 :   ...
3 :   soba_init(nthreads); /* performance hint */
4 :   for (i = 0; i < nthreads; ++i)
5 :     pthread_create(..., NULL, consumer, NULL);
6 :   for (i = 0; i < nblocks; ++i) {
7 :     char *block = read_block(i);
8 :     pthread_mutex_lock(&mu);
9 :     enqueue(q, block);
10:     pthread_cond_signal(&cv);
11:     pthread_mutex_unlock(&mu);
12:   }
13:   ...
14: }
15: void *consumer(void *arg) {
16:   while(1) {
17:     pthread_mutex_lock(&mu);
18:     while (empty(q)) // termination logic elided for clarity
19:       pthread_cond_wait(&cv, &mu);
20:     char *block = dequeue(q);
21:     pthread_mutex_unlock(&mu);
22:     ...
23:     soba_wait(); /* performance hint */
24:     compress(block);
25:   }
26: }
```

Figure 5.1: *Simplified* `PBZip2`*.* It uses the producer-consumer idiom to compress a file in parallel.

previous systems suffer from various problems doing so, illustrated below using two representative, open-source systems.

CoreDet [12] represents DMT systems that balance load by counting instructions each thread has run [12, 13, 34, 51, 80]. While the schedules computed may have reasonable overhead, minor input or program changes perturb the instruction counts and subsequently the schedules, destabilizing program behaviors. When running the example with CoreDet on eight different files, we observed five different synchronization schedules. This instability is counterintuitive and raises

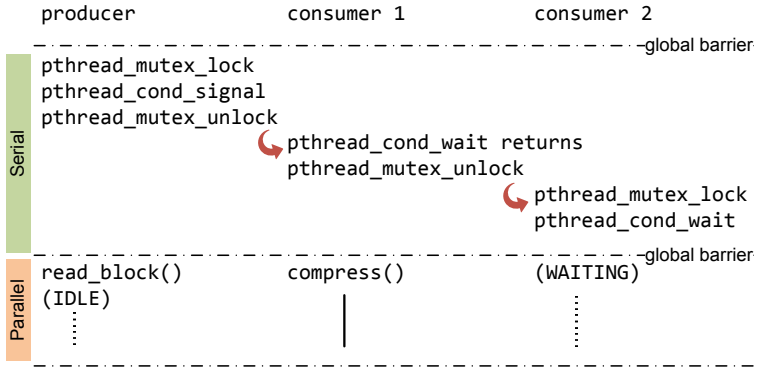


Figure 5.2: *A* DThreads *schedule.* All `compress` calls are serialized. `read_block` runs much faster than `compress`.

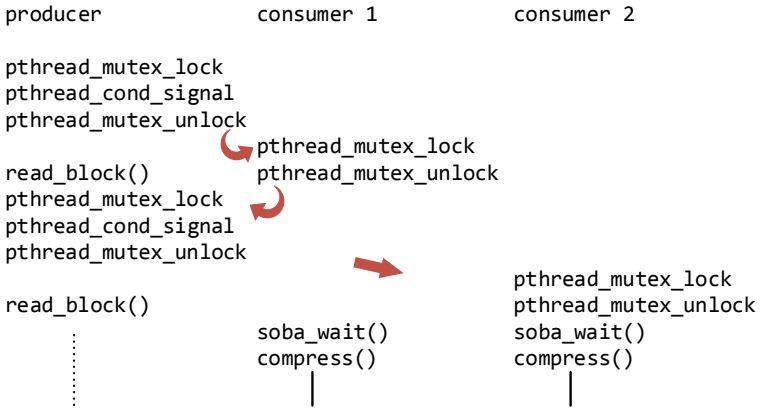


Figure 5.3: *A* Parrot *schedule with performance hints.*

new reliability challenges. For instance, testing one input provides little assurance for very similar inputs. Reproducing a bug may require every bit of the bug-inducing input, including the data a user typed, environment variables, shared libraries, etc. Missing one bit may deterministically hide the bug. CoreDet also relies on static analysis to detect and count shared memory load and store instructions, but the inherent imprecision of static analysis causes it to instrument unnecessary accesses, resulting in high overhead. On this example, CoreDet causes a 4.2× slowdown over nondeterministic execution with a 400 MB file and 16 threads.

DThreads [66] represents StableMT systems that ignore load imbalance among threads. It works by alternating between a serial and a parallel phase, separated by global barriers. In a serial phase, it lets each thread do one synchronization in order. In a parallel phase, it lets threads run until all of them are about to do synchronizations. A parallel phase lasts as long as the slowest thread, and is oblivious to the execution times of the other threads. When running the example

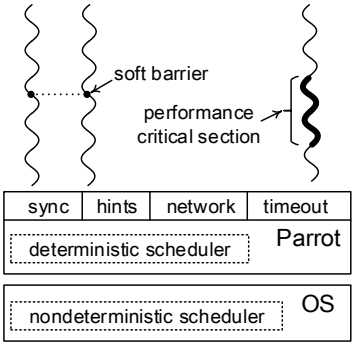


Figure 5.4: PARROT *architecture.*

with two threads, we observed the DTHREADS schedule in Figure 5.2. This schedule is stable because it can compress any file, but it is also very slow because it serializes all `compress` calls. We observed 7.7× slowdown with 16 threads; and more threads give bigger slowdowns.

This *serialization* problem is not specific to only DTHREADS. Rather, it is general to all StableMT systems that ignore load imbalance.

Running the example with PARROT is easy; users do

```
$ LD_PRELOAD=./parrot.so program args...
```

During the execution, PARROT intercepts Pthreads synchronizations. Without the hints at lines 3 and 23, PARROT schedules the synchronizations using round-robin. This schedule also serializes the `compress` calls, yielding the same slowdown as DTHREADS. Developers can easily detect this performance problem with sample runs, and PARROT simplifies performance debugging by deterministically reproducing the problem and reporting synchronizations excessively delayed by round-robin (e.g., the return of `pthread_cond_wait` here).

To solve the serialization problem, we added a soft barrier at lines 3 and 23. Line 3 informs PARROT that the program has a coscheduling group involving `nthreads` threads, and line 23 is the starting point of coscheduling. With these hints, PARROT switched to the schedule in Figure 5.3 which ran `compress` in parallel, achieving 0.8% overhead compared to nondeterministic execution. A soft barrier is different from classic synchronizations and can be safely ignored without affecting correctness. For instance, if the file blocks cannot be evenly divided among the threads, the soft barrier will time out on the last round of input blocks. Moreover, for reasonable performance, we need to align only time-consuming computations (e.g., `compress`, not `read_block`).

5.2.2 Architecture

Figure 5.4 shows PARROT's architecture. We designed PARROT to be simple and deployable. It consists of a deterministic user-space scheduler, implementation of hints, a set of wrapper functions for intercepting Pthreads, network, and timeout operations. For simplicity, the scheduler schedules only synchronizations, and delegates everything else, such as assigning threads to CPU cores, to the OS scheduler. The wrapper functions typically call into the scheduler for round-robin scheduling, then delegate the actual implementation to Pthreads or the OS. Synchronizations in performance critical sections and inherently nondeterministic operations (e.g., `recv`) are scheduled by the OS scheduler.

5.3 Performance Hint Abstractions

PARROT provides two performance-hint abstractions: a *soft barrier* and a *performance critical section*. This section describes these abstractions and their usage.

5.3.1 Soft Barrier

A *soft barrier* encourages the scheduler to coschedule a group of threads at given program points. It is for performance only, and a scheduler can ignore it without affecting correctness. It operates as a barrier with deterministic timeouts in PARROT, helping PARROT switch to faster schedules that avoid serializing parallel computations. The interface is

```
void soba_init(int group_size, void *key, int timeout);
void soba_wait(void *key);
```

One thread calls `soba_init(N, key, timeout)` to initialize the barrier named `key`, logically indicating that a group of `N` threads will be spawned. Subsequently, any thread calling `soba_wait(key)` will block until either (1) `N`-1 other threads have also called `soba_wait(key)` or (2) `timeout` time has elapsed since the first thread arrived at the barrier. This timeout is made deterministic by PARROT (§5.4.1). `soba_init` can be called multiple times: if the number of coscheduled threads varies but is known at runtime, the soft barrier can be initialized before each use. Both `key` and `timeout` in `soba_init` are optional. An absent `key` refers to a unique anonymous barrier. An absent `timeout` initializes the barrier with the default timeout.

A soft barrier may help developers express coscheduling intent to classic nondeterministic schedulers [83]. One advantage is that it makes the group of threads and program points explicit. It is more robust to developer mistakes than a real barrier [37] for coscheduling purposes because schedulers cannot ignore a real barrier.

5.3.2 Performance Critical Section

A *performance critical section* identifies a code region as a potential bottleneck and encourages the scheduler to get through the region fast. When a thread enters a performance critical section, PARROT removes the thread from the round-robin scheduling and delegates it to the OS scheduler for nondeterministic execution. PARROT thus gains speed from allowing additional schedules. The interface is

```
void pcs_enter();
void pcs_exit();
```

The `pcs_enter` function marks the entry of a performance critical section and `pcs_exit` the exit.

5.3.3 Usage of the Two Hints

Soft barrier. Developers should generally use soft barriers to align high-level, time-consuming parallel computations, such as the `compress` calls in `PBZip2`. A generic method is to use performance debugging tools or PARROT's logs to detect synchronizations excessively delayed by PARROT's round-robin scheduling, then identify the serialized parallel computations.

A second method is to add soft barriers based on parallel computation patterns. Below we describe how to do so based on four parallel computation patterns we observed from the 108 evaluated programs.

- *Data partition.* Data is partitioned among worker threads, and each worker thread computes on a partition. This pattern is the most common; 86 out of the 108 programs follow this pattern, including the programs with fork-join parallelism. Most programs with this pattern need no soft barriers. In rare cases when soft barriers are needed, developers can add `soba_wait` before each worker's computation. These soft barriers often work extremely well.

- *Pipeline.* The threads are split into stages of a pipeline, and each item in the workload flows through the pipeline stages. `ferret`, `dedup`, `vips`, and `x264` from PARSEC [88] follow this pattern. These programs often need soft barriers because threads have different roles and thus do different synchronizations, causing default schedules to serialize computations. The methodology is to align the most time-consuming stages of the pipeline.

- *Map-reduce.* Programs with this pattern use both data partition and pipeline, so the methodology follows both: align the map function and, if the reduce function runs for roughly the same amount of time as the map function, align reduce with map.

- *Workpile.* The workload consists of a pile of independent work items, processed by worker threads running in parallel. Among the programs we evaluated, Berkeley DB, OpenLDAP, Redis, `pfscan`, and `aget` fall in this category. These programs often need no soft barriers because it typically takes similar times to process most items.

Performance critical section. Unlike a soft barrier, a performance critical section may trade some determinism for performance. Consequently, it should be applied with caution, only when (1) a code region imposes high performance overhead on deterministic execution and (2) the additional schedules have been thoroughly checked by tools or advanced developers. Fortunately, both conditions are often easy to meet because the synchronizations causing high performance overhead are often low-level synchronizations (e.g., lock operations protecting a shared counter), straightforward to analyze with local reasoning or model checkers.

Of all 108 evaluated programs, only 9 need performance critical sections for reasonable performance; all other 99 programs need not trade determinism for performance. Moreover, PARROT-DBUG verified all schedules in all 4 real-world programs that need performance critical sections, providing high assurance.

Developers can identify where to add performance critical sections also using performance debugging tools. For instance, frequent synchronizations with medium round-robin delays are often good candidates for a performance critical section. Developers can also focus on such patterns as synchronizations in a tight loop, synchronizations inside an abstraction boundary (e.g., `lock()` inside a custom memory allocator), and tiny critical sections (e.g., "`lock(); x++; unlock();`").

```
void get_turn(void);
void put_turn(void);
 int wait(void *addr, int timeout);
void signal(void *addr);
void broadcast(void *addr);
void nondet_begin(void);
void nondet_end(void);
```

Table 5.1: *Scheduler primitives.*

5.4 Parrot Runtime

The Parrot runtime contains implementation of the hint abstractions (§5.4.3) and a set of wrapper functions that intercept Pthreads (§5.4.2), network (§5.4.4), and timeout (§5.4.5) operations. The wrappers interpose dynamically loaded library calls via `LD_PRELOAD` and "trap" the calls into Parrot's deterministic scheduler (§5.4.1). Instead of reimplementing the operations from scratch, these wrappers leverage existing runtimes, greatly simplifying Parrot's implementation, deployment, and inter-operation with code that assumes standard runtimes (e.g., debuggers).

5.4.1 Scheduler

The scheduler intercepts synchronization calls and releases threads using the well-understood, deterministic round-robin algorithm: the first thread enters synchronization first, the second thread second, ..., and repeat. It does not control non-synchronization code, often the majority of code, which runs in parallel. It maintains a queue of runnable threads (*run queue*) and another queue of waiting threads (*wait queue*), like typical schedulers. Only the head of the run queue may enter synchronization next. Once the synchronization call is executed, Parrot updates the queues accordingly. For instance, for `pthread_create`, Parrot appends the new thread to the tail of the run queue and rotates the head to the tail. By maintaining its own queues, Parrot avoids nondeterminism in the OS scheduler and the Pthreads library.

To implement operations in the Parrot runtime, the scheduler provides a monitor-like internal interface, shown in Table 5.1. The first five functions map one-to-one to functions of a typical monitor, except the scheduler functions are deterministic. The last two are for selectively reverting

to nondeterministic execution. The rest of this subsection describes these functions.

The `get_turn` function waits until the calling thread becomes the head of the run queue, i.e., the thread gets a "turn" to do a synchronization. The `put_turn` function rotates the calling thread from the head to the tail of the run queue, i.e., the thread gives up a turn. The `wait` function is similar to `pthread_cond_timedwait`. It requires that the calling thread has the turn. It records the address the thread is waiting for and the timeout (see next paragraph), and moves the calling thread to the tail of the wait queue. The thread is moved to the tail of the run queue when (1) another thread wakes it up via `signal` or `broadcast` or (2) the timeout has expired. The `wait` function returns when the calling thread gets a turn again. Its return value indicates how the thread was woken up. The `signal(void *addr)` function appends the first thread waiting for `addr` to the run queue. The `broadcast(void *addr)` function appends all threads waiting for `addr` to the run queue in order. Both `signal` and `broadcast` require the turn.

The `timeout` in the `wait` function does not specify real time, but relative *logical time* that counts the number of turns executed since the beginning of current execution. In each call to the `get_turn` function, PARROT increments this logical time and checks for timeouts. (If all threads block, PARROT keeps the logic time advancing with an idle thread; see §5.4.5.) The `wait` function takes a relative timeout argument. If current logical time is t_l, a timeout of 10 means waking up the thread at logical time $t_l + 10$. A `wait(NULL, timeout)` call is a logical sleep, and a `wait(addr, 0)` call never times out.

The last two functions in Table 5.1 support performance critical sections and network operations. They set the calling thread's execution mode to nondeterministic or deterministic. PARROT always schedules synchronizations of deterministic threads using round-robin, but it lets the OS scheduler schedule nondeterministic threads. Implementation-wise, the `nondet_begin` function marks the calling thread as nondeterministic and simply returns. This thread will be lazily removed from the run queue by the thread that next tries to pass the turn to it. (Next paragraph explains why the lazy update.) The `nondet_end` function marks the calling thread as deterministic and appends it to an additional queue. This thread will be lazily appended to the run queue by the next thread getting the turn.

We have optimized the multicore scheduler implementation for the most frequent operations: `get_turn`, `put_turn`, `wait`, and `signal`. Each thread has an integer flag and condition variable.

```
int wrap_mutex_lock(pthread_mutex_t *mu){
  scheduler.get_turn();
  while(pthread_mutex_trylock(mu))
    scheduler.wait(mu, 0);
  scheduler.put_turn();
  return 0; /* error handling is omitted for clarity. */
}
int wrap_mutex_unlock(pthread_mutex_t *mu){
  scheduler.get_turn();
  pthread_mutex_unlock(mu);
  scheduler.signal(mu);
  scheduler.put_turn();
  return 0; /* error handling is omitted for clarity. */
}
```

Figure 5.5: *Wrappers of Pthreads mutex lock&unlock.*

The `get_turn` function spin-waits on the current thread's flag for a while before block-waiting on the condition variable. The `wait` function needs to get the turn before it returns, so it uses the same combined spin- and block-wait strategy as the `get_turn` function. The `put_turn` and the `signal` functions signal both the flag and the condition variable of the next thread. In the common case, these operations acquire no lock and do not block-wait. The lazy updates above simplify the implementation of this optimization by maintaining the invariant that only the head of the run queue can modify the run and wait queues.

5.4.2 Synchronizations

PARROT handles all synchronizations on Pthreads mutexes, read-write locks, condition variables, semaphores, and barriers. It also handles thread creation, join, and exit. It need not implement the other Pthreads functions such as thread ID operations, another advantage of leveraging existing Pthreads runtimes. In total, PARROT has 38 synchronization wrappers. They ensure a total (round-robin) order of synchronizations by (1) using the scheduler primitives to ensure that at most one wrapper has the turn and (2) executing the actual synchronizations only when the turn is held.

Figure 5.5 shows the pseudo code of our Pthreads mutex lock and unlock wrappers. Both are

```
int wrap_cond_wait(pthread_cond_t *cv,pthread_mutex_t *mu){
  scheduler.get_turn();
  pthread_mutex_unlock(mu);
  scheduler.signal(mu);
  scheduler.wait(cv, 0);
  while(pthread_mutex_trylock(mu))
    scheduler.wait(mu, 0);
  scheduler.put_turn();
  return 0; /* error handling is omitted for clarity. */
}
```

Figure 5.6: *Wrapper of* `pthread_cond_wait`.

quite simple; so are most other wrappers. The lock wrapper uses the try-version of the Pthreads lock operation to avoid deadlock: if the head of run queue is blocked waiting for a lock before giving up the turn, no other thread can get the turn.

Figure 5.6 shows the `pthread_cond_wait` wrapper. It is slightly more complex than the lock and unlock wrappers for two reasons. First, there is no try-version of `pthread_cond_wait`, so Parrot cannot use the same trick to avoid deadlock as in the lock wrapper. Second, Parrot must ensure that unlocking the mutex and waiting on the conditional variable are atomic (to avoid the well-known lost-wakeup problem). Parrot solves these issues by implementing the wait with the scheduler's `wait` which atomically gives up the turn and blocks the calling thread on the wait queue. The wrapper of `pthread_cond_signal` (not shown) calls the scheduler's `signal` accordingly.

Thread creation is the most complex of all wrappers for two reasons. First, it must deterministically assign a logical thread ID to the newly created thread because the system's thread IDs are nondeterministic. Second, it must also prevent the new thread from using the logical ID before the ID is assigned. Parrot solves these issues by synchronizing the current and new threads with two semaphores, one to make the new thread wait for the current thread to assign an ID, and the other to make the current thread wait until the child gets the ID.

5.4.3 Performance Hints

PARROT implements performance hints using the scheduler primitives. It implements the soft barrier as a reusable barrier with a deterministic timeout. It implements the performance critical section by simply calling `nondet_begin()` and `nondet_end()`.

One tricky issue is that deterministic and nondeterministic executions may interfere. Consider a deterministic thread t_1 trying to lock a mutex that a nondeterministic t_2 is trying to unlock. Nondeterministic thread t_2 always "wins" because the timing of t_2's unlock directly influences t_1's lock regardless of how hard PARROT tries to run t_1 deterministically. An additional concern is deadlock: PARROT may move t_1 to the wait queue but never wake t_1 up because it cannot see t_2's unlock.

To avoid the above interference, PARROT requires that synchronization variables accessed in nondeterministic execution are isolated from those accessed in deterministic execution. This *strong isolation* is easy to achieve based on our experiments because, as discussed in §5.3, the synchronizations causing high overhead on deterministic execution tend to be low-level synchronizations already isolated from other synchronizations. To help developers write performance critical sections that conform to strong isolation, PARROT checks this property at runtime: it tracks two sets of synchronization variables accessed within deterministic and nondeterministic executions, and emits a warning when the two sets overlap. Strong isolation is considerably stronger than necessary: to avoid interference, it suffices to forbid deterministic and nondeterministic sections from *concurrently* accessing the same synchronization variables. We have not implemented this *weak isolation* because strong isolation works well for all programs evaluated.

5.4.4 Network Operations

To handle network operations, PARROT leverages the `nondet_begin` and `nondet_end` primitives. Before a blocking operation such as `recv`, it calls `nondet_begin` to hand the thread to the OS scheduler. When the operation returns, PARROT calls `nondet_end` to add the thread back to deterministic scheduling. PARROT supports 33 network operations such as `send`, `recv`, `accept`, and `epoll_wait`. This list suffices to run all evaluated programs that require network operations (Berkeley DB, OpenLDAP, Redis, and `aget`).

5.4.5 Timeouts

Real-world programs use timeouts (e.g., `sleep`, `epoll_wait`, and `pthread_cond_timedwait`) for periodic activities or timed waits. Not handling them can lead to nondeterministic execution and deadlocks. One deadlock example in our evaluation was running `PBZip2` with DThreads: DThreads ignores the timeout in `pthread_cond_timedwait`, but `PBZip2` sometimes relies on the timeout to finish.

Parrot makes timeouts deterministic by proportionally converting them to a logical timeout. When a thread registers a relative timeout that fires Δt_r later in real time, Parrot converts Δt_r to a relative logical timeout $\Delta t_r / R$ where R is a configurable conversion ratio. (R defaults to 3 μs, which works for all evaluated programs.) Proportional conversion is better than a fixed logical timeout because it matches developer intents better (e.g., important activities run more often). A nice fallout is that it makes some non-terminating executions terminate for model checking (§5.7.6). Of course, Parrot's logical time corresponds loosely to real time, and may be less useful for real-time applications.[1]

When all threads are on the wait queue, Parrot spawns an idle thread to keep the logical time flowing. The thread repeatedly gets the turn, sleeps for time R, and gives up the turn. An alternative to idling is fast-forwarding [13, 120]. Our experiments show that using an idle thread has better performance than fast-forwarding because the latter often wakes up threads prematurely before the pending external events (e.g., receiving a network packet) are done, wasting CPU cycles.

Parrot handles all such common operations as `sleep` and `pthread_cond_timedwait`, enough for all five evaluated programs that require timeouts (`PBZip2`, Berkeley DB, MPlayer, ImageMagick, and Redis). Pthreads timed synchronizations use absolute time, so Parrot provides developers a function `set_base_time` to pass in the base time. It uses the delta between the base time and the absolute time argument as Δt_r.

5.5 Parrot-dbug Ecosystem

Model checking is a formal verification technique that systematically explores possible executions of a program for bugs. These executions together form a *state space* graph, where states are snapshots

[1] dOS [13] discussed the possibility of converting real time to logical time but did not present how.

of the running program and edges are nondeterministic events that move the execution from one state to another. This state space is typically very large, impossible to completely explore—the so-called *state-space explosion* problem. To mitigate this problem, researchers have created many heuristics [57, 76, 117] to guide the exploration toward executions deemed more interesting, but heuristics have a risk of missing bugs. *State-space reduction* techniques [39, 46, 48] soundly prune executions without missing bugs, but the effectiveness of these techniques is limited. They work by discovering equivalence: given that execution e_1 is correct if and only if e_2 is, we need check only one of them. Unfortunately, equivalence is rare and extremely challenging to find, especially for *implementation-level* model checkers which check implementations directly [46, 57, 76, 101, 117, 118]. This difficulty is reflected in the existence of only two main reduction techniques [39, 48] for these implementation-level model checkers. Moreover, as a checked system scales, the state space after reduction still grows too large to fully explore. Despite decades of effort, state-space explosion remains the bane of model checking.

As discussed in §5.1, integrating StableMT and model checking is mutually beneficial. By reducing schedules, StableMT offers an extremely simple, effective way to mitigate and sometimes completely solve the state-space explosion problem without requiring equivalence. For instance, Parrot enables dbug to verify 99 programs, including 4 programs containing performance critical sections (§5.7.6). In return, model checking helps check the schedules that matter for Parrot and developers. For instance, it can check the default schedules chosen by Parrot, the faster schedules developers choose using soft barriers, or the schedules developers add using performance critical sections.

5.5.1 The dbug Model Checker

In principle, Parrot can be integrated with many model checkers. We chose dbug [101] for three reasons. First, it is open source, checks implementations directly, and supports Pthreads synchronizations and Linux socket calls. Second, it implements one of the most advanced state-space reduction techniques—dynamic partial order reduction (DPOR) [39], so the further reduction Parrot achieves is more valuable. Third, dbug can estimate the size of the state space based on the executions explored, a technique particularly useful for estimating the reduction Parrot can achieve when the state space explodes.

Specifically, DBUG represents the state space as an *execution tree* where nodes are states and edges are choices representing the operations executed. A path leading from the root to a leaf encodes a unique test execution as a sequence of nondeterministic operations. The total number of such paths is the state space size. To estimate this size based on a set of explored paths, DBUG uses the *weighted backtrack estimator* [56], an online variant of Knuth's offline technique for tree size estimation [60]. It treats the set of explored paths as a sample of all paths assuming uniform distribution over edges, and computes the state space size as the number of explored paths divided by the aggregated probability they are explored.

5.5.2 Integrating Parrot and dbug

A key integration challenge is that both PARROT and DBUG control the order of nondeterministic operations and may interfere, causing difficult-to-diagnose false positives. A naïve solution is to replicate PARROT's scheduling algorithm inside DBUG. This approach is not only labor-intensive, but also risky because the replicated algorithm may diverge from the real one, deviating the checked schedules from the actual ones.

Fortunately, the integration is greatly simplified because performance critical sections make nondeterminism explicit, and DBUG can ignore operations that PARROT runs deterministically. PARROT's strong-isolation semantics further prevent interference between PARROT and DBUG. Our integration uses a nested-scheduler architecture similar to Figure 5.4 except the nondeterministic scheduler is DBUG. This architecture is transparent to DBUG, and requires only minor changes (243 lines) to PARROT. First, we modified `nondet_begin` and `nondet_end` to turn DBUG on and off. Second, since DBUG explores event orders only after it has received the full set of concurrent events, we modified PARROT to notify DBUG when a thread transitions between the run queue and the wait queue in PARROT. These notifications help DBUG accurately determine when all threads in the system are waiting for DBUG to make a scheduling decision.

We found two pleasant surprises in the integration. First, soft barriers speed up DBUG executions. Second, PARROT's deterministic timeout (§5.4.5) prevents DBUG from possibly having to explore infinitely many schedules. Consider the "`while(!done) sleep(30);`" loop which can normally nondeterministically repeat any number of times before making progress. This code has only one schedule with PARROT-DBUG because PARROT makes the `sleep` return deterministically.

5.6 Determinism Discussion

PARROT's determinism is relative to three factors: (1) external input (data and timing), (2) performance critical sections, and (3) data races w.r.t. the enforced synchronization schedules. Factor 1 is inherently nondeterministic, and PARROT mitigates it by reusing schedules on inputs. Factor 2 is developer-intended. Factor 3 can be easily eliminated, but we deemed it not worthwhile. Below we explain how to make data races deterministic in PARROT and why it is not worthwhile.

We designed a simple memory commit protocol to make data races deterministic in PARROT, similar to those in previous work [8, 16, 66]. Each thread maintains a private, copy-on-write mapping of shared memory. When a thread has the turn, it commits updates and fetches other threads' updates by merging its private mapping with shared memory. Since only one thread has the turn, all commits are serialized, making data races deterministic. (Threads running nondeterministically in performance critical sections access shared memory directly as intended.) This protocol may also improve speed by reducing false sharing [66]. Implementing it can leverage existing code [66].

We deemed the effort not worthwhile for three reasons. First, making data races deterministic is often costly. Second, many races are *ad hoc synchronizations* (e.g., "`while(flag);`") [116] which require manual annotations anyway in some previous systems that make races deterministic [16, 66]. Third, most importantly, we believe that stability is much more useful for reliability than full determinism: once the set of schedules is much reduced, we can afford the nondeterminism introduced by a few data races. Specifically, previous work has shown that data races rarely occur if a synchronization schedule is enforced. For instance, PEREGRINE [32] reported at most 10 races in millions of shared memory accesses within an execution. To reproduce failures caused by the few races, we can search through a small set of schedules (e.g., fewer than 96 for an `Apache` race caused by a real workload [87]). Similarly, we can detect the races by model checking a small set of schedules [77]. In short, by vastly reducing schedules, StableMT makes the problems caused by nondeterminism easy to solve.

5.7 Evaluation

We evaluated PARROT on a diverse set of 108 programs. This set includes 55 real-world programs: Berkeley DB, a widely used database library [17]; OpenLDAP, a server implementing the

Lightweight Directory Access Protocol [82]; Redis, a fast key-value data store server [94]; MPlayer, a popular media encoder, decoder, and player [75]; `PBZip2`, a parallel compression utility [89]; `pfscan`, a parallel `grep`-like utility [91]; `aget`, a parallel file download utility [1]; all 33 parallel C++ STL algorithm implementations [105] which use OpenMP; all 14 parallel image processing utilities (which also use OpenMP) in the ImageMagick software suite [52] to create, edit, compose, or convert bitmap images. The set also includes all 53 programs in four widely used benchmark suites including 15 in PARSEC [88], 14 in Phoenix [93], 14 in SPLASH-2x [102], and 10 in NPB [79]. The Phoenix benchmark suite provides two implementations per algorithm, one using regular Pthreads (marked with `-pthread` suffix) and the other using a map-reduce library atop Pthreads. We used complete software or benchmark suites to avoid biasing our results. The programs together cover a good range of parallel programming models and idioms such as threads, OpenMP, data partition, fork-join, pipeline, map-reduce, and workpile. To the best of our knowledge, our evaluation of Parrot represents 10× more programs than any previous DMT or StableMT evaluation, and 4× more than all previous evaluations combined.

Our evaluation machine was a 2.80 GHz dual-socket hex-core Intel Xeon with 24 hyper-threading cores and 64 GB memory running Linux 3.2.14. Unless otherwise specified, we used the maximum number of truly concurrent threads allowed by the machine and programs. For 83 out of the 108 programs, we used 24. For 13 programs, we used 16 because they require the number of threads be a power of two. For `ferret`, we used 18 because it requires the number of threads to be $4n+2$. For MPlayer, we used 8, the max it takes. For the other 10 programs, we used 16 because they reach peak performance with this thread count. In scalability experiments, we varied the number of threads from 4 to the max.

Unless otherwise specified, we used the following workloads. For Berkeley DB, we used a popular benchmark `bench3n` [11], which does fine-grained, highly concurrent transactions. For both OpenLDAP and Redis, we used the benchmarks the developers themselves use, which come with the code. For MPlayer, we used its utility `mencoder` to transcode a 255 MB video (OSDI '12 keynote) from MP4 to AVI. For `PBZip2`, we compressed and decompressed a 145 MB binary file. For `pfscan`, we searched for the keyword `return` in all 16K files in `/usr/include` on our evaluation machine. For `aget`, we downloaded a 656 MB file. For all ImageMagick programs, we used a 33 MB JPG. For all 33 parallel STL algorithms, we used integer vectors with 4G elements. For PARSEC,

SPLASH-2x, and Phoenix, we used the largest workloads because they are considered "real" by the benchmark authors. For NPB, we used the second largest workloads because the largest workloads are intended for supercomputers. In workload sensitivity experiments, we used workloads of 3 or 4 different scales per program, typically with a 10× difference between scales. We also tried 15 different types of workloads for Redis and 5 for MPlayer. All workloads ran from a few seconds to about 0.5 hour, using 100 or 10 repetitions respectively to bring the standard error below 1%. All overhead means are geometric.

We compiled all programs using `gcc -O2`. To support OpenMP programs such as parallel STL algorithms, we used the GNU `libgomp`. When evaluating PARROT on the client program `aget` and the server programs OpenLDAP and Redis, we ran both endpoints on the same machine to avoid network latency. 5 programs use ad hoc synchronization [116], and we added a `sched_yield` to the busy-wait loops to make the programs work with PARROT. 5 programs use Pthreads timed operations, and we added `set_base_time` (§5.4.5) to them. We set the spin-wait of PARROT's scheduler to 10^5 cycles. We used the default soft barrier timeout of 20 except 3,000 for `ferret`. Some Phoenix programs read large files, so we ran them with a warm file cache to focus on measuring their computation time. (Cold-cache results are unusable due to large variations [84].)

The rest of this section focuses on six questions:

§5.7.1: Is PARROT easy to use? How many hints are needed to make the programs with PARROT fast?

§5.7.2: Is PARROT fast? How effective are the hints?

§5.7.3: How does it compare to previous systems?

§5.7.4: How does its performance vary according to core counts and workload scales/types?

§5.7.5: Is it deterministic in the absence of data races?

§5.7.6: How much does it improve DBUG's coverage?

5.7.1 Ease of Use

Of all 108 programs, 18 have reasonable overhead with default schedules, requiring no hints. 81 programs need a total of 87 lines of soft barrier hints: 43 need only 4 lines of generic soft barrier

Program	Lines
`mencoder`, `vips`, `swaptions`, `freqmine`, `facesim`, `x264`, `radiosity`, `radix`, `kmeans`, `linear-regression-pthread`, `linear-regression`, `matrix-multiply-pthread`, `matrix-multiply`, `word-count-pthread`, `string-match-pthread`, `string-match`, `histogram-pthread`, `histogram`	2 each
`PBZip2`, `ferret`, `kmeans-pthread`, `pca-pthread`, `pca`, `word-count`	3 each
`libgomp`, `bodytrack`	4 each
ImageMagick (12 programs)	25 total

Table 5.2: *Stats of soft barrier hints.* 81 programs need soft barrier hints. The hints in `libgomp` benefit all OpenMP programs including ImageMagick, STL, and NPB.

hints in `libgomp`, and 38 need program-specific soft barriers (Table 5.2). These programs enjoy both determinism and reasonable performance. Only 9 programs need a total of 22 lines of performance critical section hints to trade some determinism for performance (Table 5.3). On average, each program needs only 1.2 lines.

In our experience, adding hints was straightforward. It took roughly 0.5–2 hours per program despite unfamiliarity with the programs. We believe the programs' developers would spend much less time adding better hints. Parrot helped us deterministically reproduce the bottlenecks and identify the synchronizations delayed by round-robin. We used Intel `VTune` [109] and Linux `perf` [90] performance counter-based tools to identify time-consuming computations, and usually needed to align only the top two or three computations. For instance, `ferret` uses a pipeline of six stages, all serialized by the Parrot's default schedules. We aligned only two of them to bring the overhead down to a reasonable level. Aligning more stages did not help.

5.7.2 Performance

Figure 5.7 compares Parrot's performance to nondeterministic execution. Even with the maximum number of threads (16–24), the mean overhead is small: 6.9% for real-world programs, 19.0% for benchmark programs, and 12.7% for all programs. Only seven programs had over 100% overhead. The `ferret`, `freqmine`, and `is` benchmarks had dynamic load imbalance even with the start-

Program	Lines	Nondet Sync Var
pfscan	2	matches_lock
partition	2	__result_lock
fluidanimate	6	mutex[i][j]
fmm	2	lock_array[i]
cholesky	2	tasks[i].taskLock
raytrace	2	ridlock
ua	6	tlock[i]

Table 5.3: *Stats of performance critical section hints.* 9 programs need performance critical section hints. The hints in `partition` are generic for three STL programs `partition`, `nth_element`, and `partial_sort`. The last column shows the synchronization variables whose operations are made nondeterministic.

ing points of the computations aligned with soft barrier hints. `ua` also had load imbalance even after performance critical section hints are added. `x264` is a pipeline program, and its overhead comes from the soft barrier timeouts during the pipeline startup and teardown. `rtview_raytrace` and `barnes` have low-level synchronizations in tight loops, and their overhead may be further reduced with performance critical sections. Four programs, `mencoder`, `bodytrack-openmp`, `facesim`, and `linear-regression-pthread`, enjoyed big speedups, so we analyzed their executions with profiling tools. We found that the number of `mencoder`'s context switches due to synchronization decreased from 1.9M with nondeterministic executions to 921 with Parrot. The reason of the context switch savings was that Parrot's round-robin scheduling reduced contention and its synchronizations use a more efficient wait that combines spin- and block-waits (§5.4.1). `bodytrack-openmp` and `facesim` enjoyed a similar benefit. So did another 19 programs which had 10× fewer context switches with Parrot [84]. `linear-regression-pthread`'s stalled cycles were reduced by 10× with Parrot, and we speculate that Parrot's scheduler improved its affinity. (See [84] for all results on microarchitectural events.)

Figure 5.8 compares Parrot's performance with and without hints. For all the 90 programs that have hints, their mean overhead was reduced from 510% to 11.9% after hints were added. The four lines of generic soft barrier hints in `libgomp` (Table 5.2) reduced the mean overhead from 500% to 0.8% for 43 programs, program-specific soft barriers from 460% to 19.1% for 38 programs, and

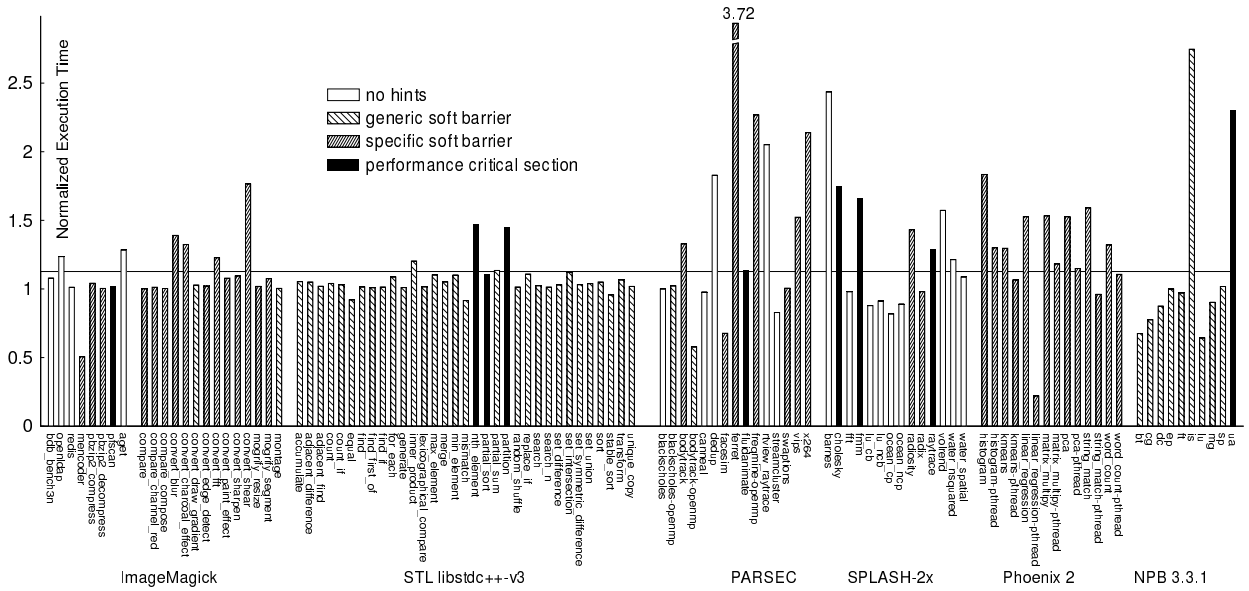


Figure 5.7: PARROT*'s performance normalized over nondeterministic execution.* The patterns of the bars show the types of the hints the programs need: no hints, generic soft barriers in `libgomp`, program-specific soft barriers, or performance critical sections. The mean overhead is 12.7% (indicated by the horizontal line).

performance critical sections from 830% to 42.1% for 9 programs. Soft barriers timed out on 12 programs (Table 5.4), which affected neither determinism nor correctness. The `kmeans` experienced

Program	Success	Timeout
`convert_shear`	725	1
`bodytrack`	60,071	2,611
`ferret`	699	2
`vips`	3,311	6
`x264`	39,480	148,470
`radiosity`	200,316	7,266
`histogram`	167	1
`kmeans`	1,470	196
`pca`	119	2
`pca-pthread`	84	1
`string-match`	64	1
`word-count`	15,468	11

Table 5.4: *Soft barrier successes and timeouts.*

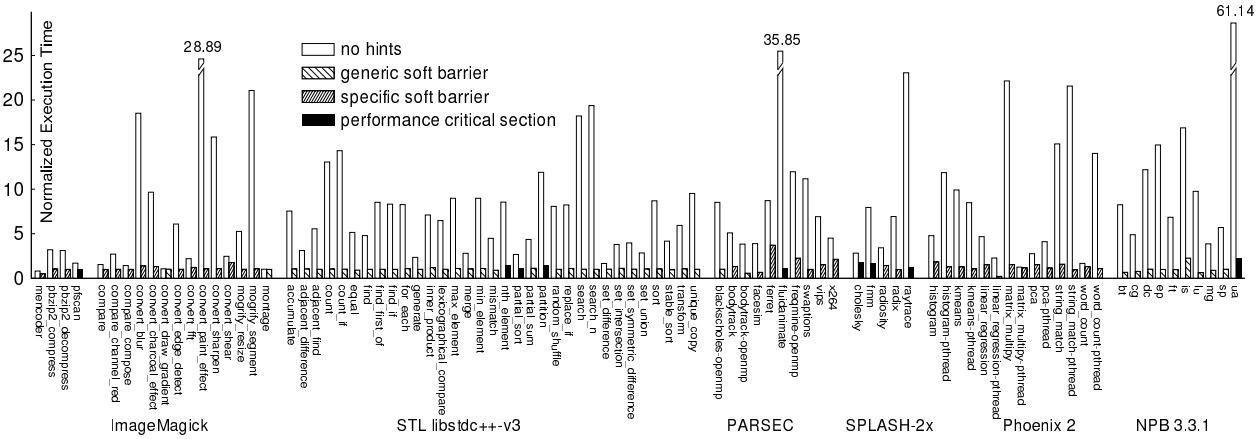


Figure 5.8: *Effects of performance hints.* They reduced PARROT's overhead from 510% to 11.9%.

over 10% timeouts, causing higher overhead. `x264` experienced many timeouts but enjoyed partial coscheduling benefits (§5.3).

5.7.3 Comparison to Prior Systems

We compared PARROT's performance to DTHREADS and COREDET. We configured both to provide the same determinism guarantee as PARROT,[2] so their overhead measured only the overhead to make synchronizations deterministic. One caveat is that neither system is specially optimized for this purpose. We managed to make only 25 programs work with both systems because not both of them support programming constructs such as read-write locks, semaphores, thread local storage, network operations, and timeouts. These programs are all benchmarks, not real-world programs.

Figure 5.9 shows the comparison results. PARROT's mean overhead is 11.8%, whereas DTHREADS's is 150.0% and COREDET's is 115.1%. DTHREADS's overhead is mainly from serializing parallel computations. `dedup`, `ferret`, `fluidanimate`, `barnes`, `radiosity`, and `raytrace` have over 500% overhead. `fluidanimate` is the slowest, whose threads wasted 59.3% of their time waiting for the other threads to do synchronizations. Without `fluidanimate`, DTHREADS's overhead is still 112.5%. (Performance hints may also help DTHREADS mitigate the serialization problem.) COREDET's overhead is mainly from counting instructions. `ferret`, `fluidanimate`, `barnes`, and `raytrace` have over 300% overhead.

[2]While Kendo's determinism guarantee is closest to PARROT's, we tried and failed to acquire its code.

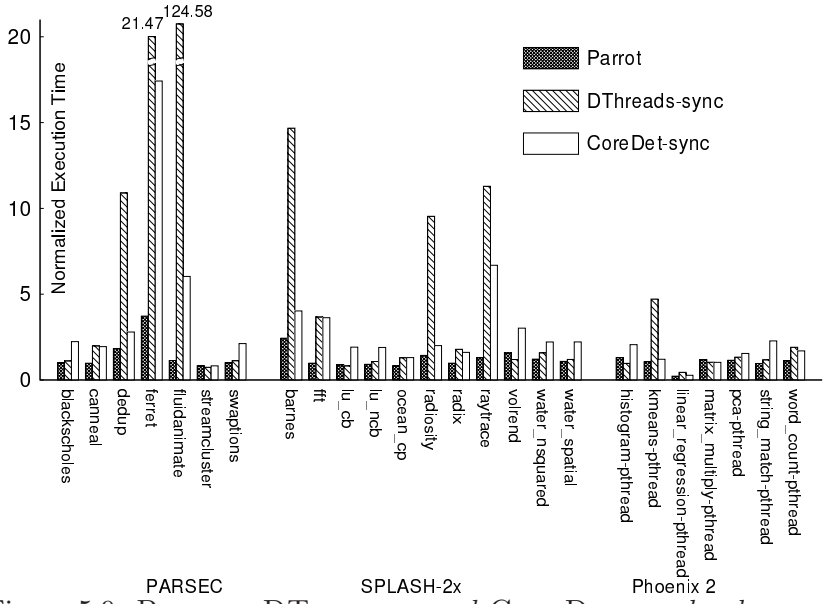


Figure 5.9: PARROT, DTHREADS, *and* CORECET *overhead.*

5.7.4 Scalability and Sensitivity

We measured PARROT's scalability on our 24-core machine. All programs varied within 40.0% from each program's mean overhead across different core counts except `ferret` (57.4%), `vips` (46.7%), `volrend` (43.1%), and `linear-regression-pthread` (57.1%). Some of these four programs use pipelines, so more threads lead to more soft barrier timeouts during pipeline startup and teardown. We also measured PARROT's scalability on three or four different workload scales as defined by the benchmark authors. All programs varied within 60% from each program's mean overhead across different scales except 14 programs, of which 9 varied from 60%–100%, 3 from 100%–150%, and 2 above 150%. The 2 programs, `partition` and `radiosity`, went above 150% because their smaller workloads run too short. For instance, `radiosity`'s native workload runs for over 200s, but its large workload runs for less than 3s and medium and small workloads for less than 0.4s. We also ran Redis on 15 types of workloads, and `mencoder` on 5. The overhead did not vary much. To summarize, PARROT's performance is robust to core count and workload scale/type. (See [84] for detailed scalability results.)

5.7.5 Determinism

We evaluated PARROT's determinism by verifying that it computed the same schedules given the same input. For all programs except those with performance critical sections, ad hoc synchronizations, and network operations, PARROT is deterministic. Our current way of marking ad hoc synchronization causes nondeterminism; annotations [116] can solve this problem. We also evaluated PARROT's determinism using a modified version of `racey` [50] that protects each shared memory access with a lock. In `racey`, each different schedule leads to a different result with high probability. We executed our modified `racey` 1,000 times without PARROT, and saw 1,000 different results. With PARROT, it always computed the same result.

5.7.6 Model Checking Coverage

Bin	# of Programs	State Space Size with dbug
A	27	$1 \sim 14$
B	18	$28 \sim 47,330$
C	25	$3.99 \times 10^{6} \sim 1.06 \times 10^{473}$
D	25	$4.75 \times 10^{511} \sim 2.10 \times 10^{19734}$

Table 5.5: *Estimated* DBUG*'s state space sizes on programs with no performance critical section nor network operation.*

To evaluate coverage, we used small workloads and two threads per workload. Otherwise, the time and space overhead of DBUG, or model checking in general, becomes prohibitive. Consequently, PARROT's reduction measured with small state spaces is a conservative estimate of its potential. Two programs, `volrend` and `ua`, were excluded because they have too many synchronization operations (e.g., 132M for `ua`), causing DBUG to run out of memory. Since model checking requires a closed (no-input) system, we paired `aget` with lightweight web server `Mongoose` [73]). We enabled state-of-the-art DPOR [39] to evaluate how much more PARROT can reduce the state space. We checked each program for a maximum of one day or until the checking session finished. We then compared the estimated state space sizes.

Table 5.5 bins all 95 programs that contain (1) no network operations and (2) either no hints or only soft barriers. For each program, PARROT-DBUG reduced the state space down to just one

schedule and finished in 2 seconds. DBUG alone could finish only 43 (out of 45 in bin A and B) within the time limit.

Table 5.6 shows the results for all 11 multithreaded programs containing network operations or performance critical sections. For all four real-world programs `pfscan`, `partition`, `nth_element`, and `partial_sort`, PARROT-DBUG effectively explored all schedules in seven hours or less, providing a strong reliability guarantee. These results also demonstrate the power of PARROT: the programs can use the checked schedules at runtime for speed.

To summarize, PARROT reduced the state space by 10^6–10^{19734} for 56 programs (50 programs in Table 5.5, 6 in Table 5.6). It increased the number of verified programs from 43 to 99 (95 programs in Table 5.5, 4 in Table 5.6).

5.8 Related Work

StableMT and DMT systems. Implementation-wise, several previous systems are not backward-compatible because they require new hardware [34], new language [21], or new programming model and OS [8]. Among backward-compatible systems, some DMT systems, including Kendo [80], COREDET [12], and COREDET-related systems [13, 51], improve performance by balancing each

Program	dbug	Parrot-dbug	Time
OpenLDAP	2.40×10^{2795}	5.70×10^{1048}	No
Redis	1.26×10^{8}	9.11×10^{7}	No
`pfscan`	2.43×10^{2117}	$32,268$	$1,201$s
`aget`	2.05×10^{17}	5.11×10^{10}	No
`nth_element`	1.35×10^{7}	$8,224$	309s
`partial_sort`	1.37×10^{7}	$8,194$	307s
`partition`	1.37×10^{7}	$8,194$	307s
`fluidanimate`	2.72×10^{218}	2.64×10^{218}	No
`cholesky`	1.81×10^{371}	5.99×10^{152}	No
`fmm`	1.25×10^{78}	2.14×10^{54}	No
`raytrace`	1.08×10^{13863}	3.68×10^{13755}	No

Table 5.6: *Estimated state space sizes for programs containing performance critical sections.* PARROT-DBUG finished 4 real-world programs (time in last column), and DBUG none.

thread's load with low-level instruction counts, so they are not stable.

Five systems can be classified as StableMT systems. Our TERN (Chapter 3) and PEREGRINE (Chapter 4) systems require sophisticated program analysis to determine input and schedule compatibility, complicating deployment. Bergan *et al* [15] built upon the ideas in TERN and PEREGRINE to statically compute a small set of schedules covering all inputs, an undecidable problem in general. Grace [16] and DTHREADS [66] ignore thread load imbalance, so they are prone to the serialization problem (§5.2.1). Grace also requires fork-join parallelism.

Compared to PARROT's evaluation, previous evaluations have several limitations. First, previous work has reported results on a narrower set of programs, typically less than 15. The programs are mostly synthetic benchmarks, not real-world programs, from incomplete suites. Second, the experimental setups are limited, often with one workload per program and up to 8 cores.[3] Lastly, little previous work except ours [31, 32, 115] has demonstrated how the approaches benefit testing or reported any quantitative results on improving reliability, making it difficult for potential adopters to justify the overhead.

State-space reduction. PARROT greatly reduces the state space of model checking, so it bears similarity to *state-space reduction techniques* (e.g., [39, 46, 48]). Partial order reduction [39, 46] has been the main reduction technique for model checkers that check implementations directly [101, 120]. It detects permutations of independent events, and checks only one permutation because all should lead to the same behavior. Recently, we proposed dynamic interface reduction [48] that checks loosely coupled components separately, avoiding expensive global exploration of all components. However, this technique has yet to be shown to work well for tightly coupled components such as threads communicating via synchronizations and shared memory.

PARROT offers three advantages over reduction techniques (§5.5): (1) it is much simpler because it does not need equivalence to reduce state space; (2) it remains effective as the checked system scales; and (3) it works transparently to reduction techniques, so it can be combined with them for further reduction. The disadvantage is that PARROT has runtime overhead.

Concurrency. Automatic mutual exclusion (AME) [53] assumes all shared memory is implicitly protected and allows advanced developers the flexibility to remove protection. It thus shares a

[3]Three exceptions used more than 8 cores: [81] (ran a 12-line program on 48 cores), [9] (ran 9 selected programs from PARSEC, SPLASH-2x, and NPB on 32 cores), and [34] (emulated 16 cores).

similar high-level philosophy with PARROT, but the differences are obvious. We are unaware of any publication describing a fully implemented AME system. PARROT is orthogonal to much previous work on concurrency error detection [38, 69, 70, 97, 123, 126], diagnosis [85, 86, 99], and correction [54, 55, 111, 114]. By reducing schedules, it potentially benefits all these techniques.

5.9 Summary

We have presented PARROT, a simple, practical Pthreads-compatible system for making threads deterministic and stable. It offers a new contract to developers. By default, it schedules synchronizations using round-robin, vastly reducing schedules. When the default schedules are slow, it allows developers to write performance hints for speed. We believe this contract eases writing correct, efficient programs. We have also presented an ecosystem formed by integrating PARROT with model checker DBUG, so that DBUG can thoroughly check PARROT's schedules, and PARROT can greatly improve DBUG's coverage. Results on a diverse set of 108 programs, roughly 10× more than any previous evaluation, show that PARROT is easy to use, fast, and scalable; and it improves DBUG's coverage by many orders of magnitude. We have released PARROT's source code, entire benchmark suite, and raw results at `github.com/columbia/smt-mc`.

Chapter 6

Conclusion

Multithreading is notoriously difficult to get right, and a root cause is that a multithreaded program may run into exponentially many possible schedules for all inputs at runtime, which brings a series of significant reliability and security challenges on understanding, testing, debugging, analyzing, and verification of multithreaded programs.

To make multithreading easier to get right, we have invented a new idea called StableMT that reuses each schedule on a wide range of inputs, greatly reducing the number of possible schedules for all inputs. Through building three StableMT systems, Tern, Peregrine, and Parrot, with each addressing a distinct research challenge, we have shown that StableMT is simple, fast, and deployable. Through applying StableMT to make reproducing concurrency bugs easier, to improve the precision of static program analysis, and to increase the coverage of model checking tools, we have quantitatively demonstrated StableMT's advantages on improving software reliability. StableMT has attracted the research community's interests, and some techniques and ideas in our previous systems have been leveraged by University of Washington researchers to compute a small set of schedules to cover all or most inputs of multithreaded programs. All the source code, benchmarks, and raw evaluation results of Parrot, our latest StableMT system, are available at `github.com/columbia/smt-mc`.

By addressing the root cause that makes multithreading difficult to get right, StableMT has broad applications on software reliability and security. In the future, we plan to apply StableMT to make replication and verification of multithreaded programs easier, and to defend against security attacks that leverage concurrency bugs.

Bibliography

[1] `http://www.enderunix.org/aget/`.

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

[3] Jeannie Albrecht, Christopher Tuttle, Alex C. Snoeren, and Amin Vahdat. Loose synchronization for large-scale networked systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX '06)*, pages 28–28, 2006.

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

[5] ab - Apache server benchmark. `http://httpd.apache.org/docs/2.2/programs/ab.html`, 2014.

[6] Apache web server. `http://www.apache.org`, 2012.

[7] Arctic Terns - Wikipedia. `http://en.wikipedia.org/wiki/Arctic_Tern`.

[8] Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. Efficient system-enforced deterministic parallelism. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI '10)*, October 2010.

[9] Amittai F. Aviram. *Deterministic OpenMP*. PhD thesis, Yale University, 2012.

[10] Dzintars Avots, Michael Dalton, V. Benjamin Livshits, and Monica S. Lam. Improving software

security with a C pointer analysis. In *Proceedings of the 27th International Conference on Software Engineering (ICSE '05)*, pages 332–341, May 2005.

[11] `http://libdb.wordpress.com/3n1/`.

[12] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan 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, March 2010.

[13] Tom Bergan, Nicholas Hunt, Luis Ceze, and Steven D. Gribble. Deterministic process groups in dOS. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI '10)*, October 2010.

[14] Tom Bergan, Joseph Devietti, Nicholas Hunt, and Luis 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)*, March 2011.

[15] Tom Bergan, Luis Ceze, and Dan Grossman. Input-covering schedules for multithreaded programs. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '13)*, October 2013.

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

[17] `http://www.sleepycat.com`.

[18] Sanjay Bhansali, Wen-Ke Chen, Stuart de Jong, Andrew Edwards, Ron Murray, Milenko Drinić, Darek Mihocka, and Joe Chau. Framework for instruction-level tracing and analysis of program executions. In *Proceedings of the 2nd International Conference on Virtual Execution Environments (VEE '06)*, pages 154–163, June 2006.

[19] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. *J. Parallel Distrib. Comput.*, 37(1):55–69, 1996.

[20] OpenMP Architecture Review Board. OpenMP application program interface version 3.0, May 2008.

[21] Robert L. Bocchino, Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen 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, October 2009.

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

[23] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: automatically generating inputs of death. In *Proceedings of the 13th ACM conference on Computer and communications security (CCS '06)*, pages 322–335, October–November 2006.

[24] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the Eighth Symposium on Operating Systems Design and Implementation (OSDI '08)*, pages 209–224, December 2008.

[25] George Candea, Stefan Bucur, and Cristian Zamfir. Automated software testing as a service. In *Proceedings of the 1st Symposium on Cloud Computing (SOCC '10)*, 2010.

[26] Miguel Castro, Manuel Costa, and Jean-Philippe Martin. Better bug reporting with better privacy. In *Thirteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '08)*, pages 319–328, March 2008.

[27] V. Chipounov, V. Georgescu, C. Zamfir, and G. Candea. Selective Symbolic Execution. In *Fifth Workshop on Hot Topics in System Dependability (HotDep '09)*, 2009.

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

[29] Peter Collingbourne, Cristian Cadar, and Paul H.J. Kelly. Symbolic crosschecking of floating-point and SIMD code. In *Proceedings of the 2011 ACM European Conference on Computer Systems (EUROSYS '11)*, pages 315–328, April 2011.

[30] Manuel Costa, Miguel Castro, Lidong Zhou, Lintao Zhang, and Marcus Peinado. Bouncer: securing software by blocking bad input. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP '07)*, pages 117–130, October 2007.

[31] Heming Cui, Jingyue Wu, Chia-Che Tsai, and Junfeng Yang. Stable deterministic multithreading through schedule memoization. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI '10)*, October 2010.

[32] Heming Cui, Jingyue Wu, John Gallagher, Huayang Guo, and Junfeng Yang. Efficient deterministic multithreading through schedule relaxation. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, pages 337–351, October 2011.

[33] Heming Cui, Jiri Simsa, Yi-Hong Lin, Hao Li, Ben Blum, Xinan Xu, Junfeng Yang, Garth A. Gibson, and Randal E. Bryant. Parrot: a practical runtime for deterministic, stable, and reliable threads. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, November 2013.

[34] Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. DMP: deterministic shared memory multiprocessing. In *Fourteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '09)*, pages 85–96, March 2009.

[35] George Dunlap, Samuel T. King, Sukru Cinar, Murtaza Basrat, and Peter Chen. ReVirt: enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI '02)*, pages 211–224, December 2002.

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

[37] Andrea C. Dusseau, Remzi H. Arpaci, and David E. Culler. Effective distributed scheduling of parallel workloads. In *Proceedings of the 1996 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '96)*, pages 25–36, May 1996.

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

[39] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32nd Annual Symposium on Principles of Programming Languages (POPL '05)*, pages 110–121, January 2005.

[40] Pedro Fonseca, Cheng Li, and Rodrigo Rodrigues. Finding complex concurrency bugs in large multi-threaded applications. In *Proceedings of the 2011 ACM European Conference on Computer Systems (EUROSYS '11)*, pages 215–228, April 2011.

[41] Qi Gao, Wenbin Zhang, Zhezhe Chen, Mai Zheng, and Feng Qin. 2ndStrike: towards manifesting hidden concurrency typestate bugs. In *Sixteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '11)*, pages 239–250, March 2011.

[42] Dennis Geels, Gautam Altekar, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Friday: global comprehension for distributed replay. In *Proceedings of the Fourth Symposium on Networked Systems Design and Implementation (NSDI '07)*, April 2007.

[43] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI '05)*, pages 213–223, June 2005.

[44] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based whitebox fuzzing. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 206–215, 2008.

[45] Patrice Godefroid, Michael Levin, and David Molnar. Automated whitebox fuzz testing. In *NDSS '08: Proceedings of 15th Network and Distributed System Security Symposium*, February 2008.

[46] P. Godefroid. Model checking for programming languages using verisoft. In *Proceedings of the*

24th Annual Symposium on Principles of Programming Languages (POPL '97), pages 174–186, January 1997.

[47] Zhenyu Guo, Xi Wang, Jian Tang, Xuezheng Liu, Zhilei Xu, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. R2: An application-level kernel for record and replay. In *Proceedings of the Eighth Symposium on Operating Systems Design and Implementation (OSDI '08)*, pages 193–208, December 2008.

[48] Huayang Guo, Ming Wu, Lidong Zhou, Gang Hu, Junfeng Yang, and Lintao Zhang. Practical software model checking via dynamic interface reduction. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, pages 265–278, October 2011.

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

[50] Mark D. Hill and Min Xu. Racey: A stress test for deterministic execution. `http://www.cs.wisc.edu/~markhill/racey.html`, 2009.

[51] Nicholas Hunt, Tom Bergan, , Luis Ceze, and Steven Gribble. DDOS: Taming nondeterminism in distributed systems. In *Eighteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '13)*, pages 499–508, 2013.

[52] `http://www.imagemagick.org/script/index.php`.

[53] Michael Isard and Andrew Birrell. Automatic mutual exclusion. In *Proceedings of the 11th USENIX workshop on Hot topics in operating systems (HOTOS '07)*, pages 3:1–3:6, 2007.

[54] Guoliang Jin, Wei Zhang, Dongdong Deng, Ben Liblit, and Shan Lu. Automated concurrency-bug fixing. In *Proceedings of the Tenth Symposium on Operating Systems Design and Implementation (OSDI '12)*, pages 221–236, 2012.

[55] Horatiu Jula, Daniel Tralamazza, Zamfir Cristian, and Candea George. Deadlock immunity: Enabling systems to defend against deadlocks. In *Proceedings of the Eighth Symposium on Operating Systems Design and Implementation (OSDI '08)*, pages 295–308, December 2008.

[56] Philip Kilby, John K. Slaney, Sylvie Thiébaux, and Toby Walsh. Estimating search tree size. In *Proceedings of the 21st national conference on Artificial intelligence (AAAI '06)*, pages 1014–1019, 2006.

[57] Charles Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. In *Proceedings of the Fourth Symposium on Networked Systems Design and Implementation (NSDI '07)*, pages 243–256, April 2007.

[58] Taesoo Kim, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Intrusion recovery using selective re-execution. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI '10)*, pages 1–9, October 2010.

[59] James C. King. A new approach to program testing. In *Proceedings of the 1975 International Conference on Reliable Software*, pages 228–233, June 1975.

[60] Donald E. Knuth. Estimating the Efficiency of Backtrack Programs. *Mathematics of Computation*, 29(129):121–136, 1975.

[61] Ravi Konuru, Harini Srinivasan, and Jong-Deok Choi. Deterministic replay of distributed Java applications. In *Proceedings of the 14th International Symposium on Parallel and Distributed Processing (IPDPS '00)*, pages 219–228, May 2000.

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

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

[64] Dongyoon Lee, Benjamin Wester, Kaushik Veeraraghavan, Satish Narayanasamy, Peter M. Chen, and Jason Flinn. Respec: efficient online multiprocessor replayvia speculation and external determinism. In *Fifteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '10)*, pages 77–90, March 2010.

[65] N. G. Leveson and C. S. Turner. An investigation of the Therac-25 accidents. *Computer*, 26(7):18–41, 1993.

[66] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. DTHREADS: efficient deterministic multithreading. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, pages 327–336, October 2011.

[67] The LLVM compiler framework. `http://llvm.org`, 2013.

[68] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. Bugbench: Benchmarks for evaluating bug detection tools. In *Proceedings of the first Workshop on the Evaluation of Software Defect Detection Tools (BUGS '05)*, June 2005.

[69] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. AVIO: detecting atomicity violations via access interleaving invariants. In *Twelfth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '06)*, pages 37–48, October 2006.

[70] Shan Lu, Soyeon Park, Chongfeng Hu, Xiao Ma, Weihang Jiang, Zhenmin Li, Raluca A. Popa, and Yuanyuan Zhou. Muvi: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP '07)*, pages 103–116, 2007.

[71] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan 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, March 2008.

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

[73] `https://code.google.com/p/mongoose/`.

[74] Pablo Montesinos, Matthew Hicks, Samuel T. King, and Josep Torrellas. Capo: a software-hardware interface for practical deterministic multiprocessor replay. In *Fourteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '09)*, pages 73–84, March 2009.

[75] `http://www.mplayerhq.hu/design7/news.html`.

[76] Madanlal Musuvathi, David Y.W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. CMC: A pragmatic approach to model checking real code. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI '02)*, pages 75–88, December 2002.

[77] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian 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, December 2008.

[78] MySQL Database. `http://www.mysql.com/`, 2014.

[79] `http://www.nas.nasa.gov/publications/npb.html`.

[80] Marek Olszewski, Jason Ansel, and Saman 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, March 2009.

[81] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Scaling deterministic multithreading. In *The 2nd Workshop on Determinism and Correctness in Parallel Programming (WODET '11)*, March 2011.

[82] `http://www.openldap.org/`.

[83] J. K. Ousterhout. Scheduling Techniques for Concurrent Systems. In *Proceedings of Third International Conference on Distributed Computing Systems (ICDCS '82)*, pages 22–30, 1982.

[84] Complete source code, benchmark suite, and raw results of the PARROT thread runtime. `https://github.com/columbia/smt-mc`.

[85] Chang-Seo Park and Koushik Sen. Randomized active atomicity violation detection in concurrent programs. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '08/FSE-16)*, pages 135–145, November 2008.

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

[87] Soyeon Park, Yuanyuan Zhou, Weiwei Xiong, Zuoning Yin, Rini Kaushik, Kyu H. Lee, and Shan Lu. PRES: probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, pages 177–192, October 2009.

[88] The Princeton application repository for shared-memory computers (PARSEC). `http://parsec.cs.princeton.edu/`, 2010.

[89] Parallel BZIP2 (PBZIP2). `http://compression.ca/pbzip2/`, 2011.

[90] `https://perf.wiki.kernel.org/index.php/Main_Page/`.

[91] `http://ostatic.com/pfscan`.

[92] Kevin Poulsen. Software bug contributed to blackout. `http://www.securityfocus.com/news/8016`, February 2004.

[93] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture (HPCA '07)*, pages 13–24, 2007.

[94] `http://redis.io/`.

[95] Michiel Ronsse and Koen De Bosschere. RecPlay: a fully integrated practical record/replay system. *ACM Trans. Comput. Syst.*, 17(2):133–152, 1999.

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

[97] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas E. Anderson. Eraser: A dynamic data race detector for multithreaded programming. *ACM Transactions on Computer Systems*, pages 391–411, November 1997.

[98] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference held jointly with the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-13)*, pages 263–272, September 2005.

[99] Koushik Sen. Race directed random testing of concurrent programs. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI '08)*, pages 11–21, June 2008.

[100] Jiri Simsa, Randy Bryant, and Garth Gibson. dBug: Systematic Evaluation of Distributed Systems. In *5th International Workshop on Systems Software Verification (SSV '10), co-located with 9th USENIX Symposium On Operating Systems Design and Implementation (OSDI '10)*, October 2010.

[101] Jiri Simsa, Garth Gibson, and Randy Bryant. dBug: Systematic Testing of Unmodified Distributed and Multi-Threaded Systems. In *The 18th International SPIN Workshop on Model Checking of Software (SPIN'11)*, pages 188–193, 2011.

[102] `http://parsec.cs.princeton.edu/parsec3-doc.htm`.

[103] Stanford parallel applications for shared memory (SPLASH). `http://www-flash.stanford.edu/apps/SPLASH/`, 2007.

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

[105] `http://gcc.gnu.org/onlinedocs/libstdc++/manual/parallel_mode.html`.

[106] SysBench: a system performance benchmark. `http://sysbench.sourceforge.net`, 2004.

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

[108] `http://www.vmware.com/solutions/vla/`.

[109] `http://software.intel.com/en-us/intel-vtune-amplifier-xe/`.

[110] Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, 1974.

[111] Yin Wang, Terence Kelly, Manjunath Kudlur, Stephane Lafortune, and Scott Mahlke. Gadara: Dynamic deadlock avoidance for multithreaded programs. In *Proceedings of the Eighth Symposium on Operating Systems Design and Implementation (OSDI '08)*, pages 281–294, December 2008.

[112] John Whaley. bddbddb Project. `http://bddbddb.sourceforge.net`.

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

[114] Jingyue Wu, Heming Cui, and Junfeng Yang. Bypassing races in live applications with execution filters. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI '10)*, October 2010.

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

[116] Weiwei Xiong, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, and Zhiqiang Ma. Ad hoc synchronization considered harmful. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI '10)*, October 2010.

[117] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors. In *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation (OSDI '04)*, pages 273–288, December 2004.

[118] Junfeng Yang, Can Sar, and Dawson Engler. Explode: a lightweight, general system for finding serious storage system errors. In *Proceedings of the Seventh Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 131–146, November 2006.

[119] Junfeng Yang, Can Sar, Paul Twohey, Cristian Cadar, and Dawson Engler. Automatically generating malicious disks using symbolic execution. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P '06)*, pages 243–257, May 2006.

[120] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. MODIST: Transparent model checking of unmodified distributed systems. In *Proceedings of the Sixth Symposium on Networked Systems Design and Implementation (NSDI '09)*, pages 213–228, April 2009.

[121] Junfeng Yang, Ang Cui, Sal Stolfo, and Simha Sethumadhavan. Concurrency attacks. In *the Fourth USENIX Workshop on Hot Topics in Parallelism (HOTPAR '12)*, June 2012.

[122] Junfeng Yang, Heming Cui, Jingyue Wu, Yang Tang, and Gang Hu. Determinism is not enough: Making parallel programs reliable with stable multithreading. *Communications of the ACM*, 2014.

[123] Yuan Yu, Tom Rodeheffer, and Wei Chen. RaceTrack: efficient detection of data race conditions via adaptive tracking. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 221–234, October 2005.

[124] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. SherLog: error diagnosis by connecting clues from run-time logs. In *Fifteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '10)*, pages 143–154, March 2010.

[125] Cristian Zamfir and George Candea. Execution synthesis: a technique for automated software debugging. In *Proceedings of the 2010 ACM European Conference on Computer Systems (EUROSYS '10)*, pages 321–334, April 2010.

[126] Wei Zhang, Chong Sun, and Shan Lu. ConMem: detecting severe concurrency bugs through

an effect-oriented approach. In *Fifteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '10)*, pages 179–192, March 2010.

[127] Wei Zhang, Junghee Lim, Ramya Olichandran, Joel Scherpelz, Guoliang Jin, Shan Lu, and Thomas Reps. ConSeq: detecting concurrency bugs through sequential errors. In *Sixteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '11)*, pages 251–264, March 2011.