Stable Deterministic Multithreading through Schedule Memoization

Heming Cui, Jingyue Wu, Chia-che Tsai, Junfeng Yang
{*heming, jingyue, ct2459, junfeng*}*@cs.columbia.edu*
Computer Science Department
Columbia University
New York, NY 10027

Abstract

A deterministic multithreading (DMT) system eliminates nondeterminism in thread scheduling, simplifying the development of multithreaded programs. However, existing DMT systems are unstable; they may force a program to (ad)venture into vastly different schedules even for slightly different inputs or execution environments, defeating many benefits of determinism. Moreover, few existing DMT systems work with server programs whose inputs arrive continuously and nondeterministically.

TERN is a *stable* DMT system. The key novelty in TERN is the idea of *schedule memoization* that memoizes past working schedules and reuses them on future inputs, making program behaviors stable across different inputs. A second novelty in TERN is the idea of *windowing* that extends schedule memoization to server programs by splitting continuous request streams into windows of requests. Our TERN implementation runs on Linux. It operates as user-space schedulers, requiring no changes to the OS and only a few lines of changes to the application programs. We evaluated TERN on a diverse set of 14 programs (*e.g.*, Apache and MySQL) with real and synthetic workloads. Our results show that TERN is easy to use, makes programs more deterministic and stable, and has reasonable overhead.

1 Introduction

Multithreaded programs are difficult to write, test, and debug. A key reason is nondeterminism: different runs of a multithreaded program may show different behaviors, depending on how the threads interleave [35].

Two main factors make threads interleave nondeterministically. The first is *scheduling*, how the OS and hardware schedule threads. Scheduling nondeterminism is not essential and can be eliminated without affecting correctness for most programs. The second is *input*, what data (*input data*) arrives at what time (*input timing*). Input nondeterminism sometimes is essential because major changes in inputs require different schedules. However, frequently input nondeterminism is not essential and the same schedule can be used to process many different inputs (§2.2). We believe nonessential nondeterminism should be eliminated in favor of determinism.

Deterministic multithreading (DMT) systems [13, 22, 41] make threads more deterministic by eliminating scheduling nondeterminism. Specifically, they constrain a multithreaded program such that it always uses the same thread schedule for the same input. By doing so, these systems make program behaviors repeatable, increase testing confidence, and ease bug reproduction.

Unfortunately, though existing DMT systems eliminate scheduling nondeterminism, they do not reduce input nondeterminism. In fact, they may aggravate the effects of input nondeterminism because of their design limitation: when scheduling the threads to process an input, they consider only this 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 many benefits of determinism. We call this the *instability* problem. This problem is confirmed by our results (§8.2.1) from an existing DMT system [13].

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.

This paper presents TERN, a schedule-centric, stateful DMT system. It addresses the instability problem using an 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 sched-

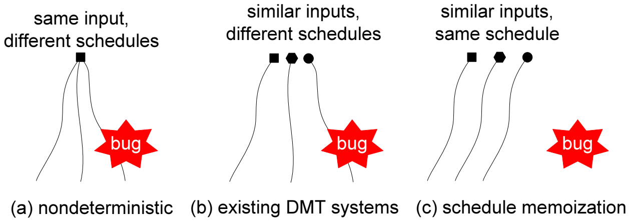

Figure 1: *Advantage of schedule memoization.* Each solid shape represents an input, and each curved line a schedule. Schedule memoization reuses schedules when possible, avoiding bugs in unknown schedules and making program behaviors repeatable across similar inputs.

ule. 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. By reusing schedules, TERN avoids potential errors in unknown schedules. This advantage is illustrated in Figure 1.

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 [2].

A second advantage of schedule memoization is that it 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 [17], a symbolic execution engine. Our implementation is software-only, works with general C/C++ programs using threads, 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 server programs Apache [10] and MySQL [4], a parallel compression utility PBZip2 [5], and 11 scientific programs in SPLASH2 [6]. 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 adapt them to 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 [13] made three bugs inconsistently occur or disappear depending on minor input changes, TERN always 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 §7.

Our main conceptual contributions are that we identified the instability problem in existing DMT systems and proposed two ideas, schedule memoization and windowing, to mitigate input nondeterminism. Our engineering contributions include the TERN system and its evaluation of real programs. To the best of our knowledge, TERN is the first stable DMT system, the first to mitigate input timing nondeterminism, and the first shown to work on programs as large, complex, and nondeterministic as Apache and MySQL. TERN demonstrates that DMT has the potential to be deployed today.

This paper is organized as follows. We first present a background (§2) and an overview of TERN (§3). We then describe TERN's interface (§4), schedule memoization for batch programs (§5), and windowing to extend TERN to server programs (§6). We then present refinements we made to optimize TERN (§7). Lastly, we show our experimental results (§8), discuss related work (§9), and conclude (§10).

2 Background

This section presents a background of TERN. We explain the instability problem of existing DMT systems (§2.1), our choice of schedule representation in TERN (§2.2), and why we can reuse schedules across inputs (§2.3).

2.1 The Instability Problem

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 [41]). 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 [41]). 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.

2.2 Schedule Representation and Determinism

Previous DMT systems have considered two types of schedules: (1) a deterministic order of shared memory accesses [13, 22] and (2) a synchronization order (*i.e.*, a total order of synchronization operations) [41]. The first type of schedules are truly deterministic even if there are races, but they are costly to enforce on commodity hardware (*e.g.*, up to 10 times overhead [13]). The second type can be efficiently enforced (*e.g.*, 16% overhead [41]) 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 [41, 46].

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 [39]. These errors can be deterministically reproduced or avoided using only synchronization orders.

Program	Input Constraints for Schedule Reuse
PBZip2	Same number of file blocks (`NumBlocks` or `-b`) and threads (`-p`)
Apache	For groups of typical HTTP GET requests, same cache status and response sizes
fft	Same number of threads (`-p`)
lu	Same number of threads (`-p`), size of the matrix (`-n`), and block size (`-b`)
barnes	Same number of threads (`NPROC`) and values of variables `dtime` and `tstop`

Table 1: *Input constraints of five programs to reuse schedules.* Identifiers without a dash are configuration variables, and those with a dash are command line options.

Although data races may still make runs which reuse schedules nondeterministic, TERN is less prone to this problem than existing DMT systems [41] 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.

2.3 Why Can We Reuse Schedules?

This subsection presents an intuitive and an empirical argument to support our insight that we can frequently reuse schedules for many programs/workloads. Intuitively, synchronization operations map to developer intents of inter-thread control flow. By enforcing the same synchronization order, we fix the same inter-thread "path," but still allow many different inputs to flow down this path. (This observation is similarly made for sequential paths [11, 12, 26].)

To empirically validate our insight, we studied the input constraints to reuse schedules for five programs, including a parallel compression utility PBZip2; the Apache web server; and three scientific programs fft, lu, and barnes in SPLASH2. Table 1 shows the results for all programs studied. We found that the input constraints were often general, allowing frequent reuses of schedules. For instance, PBZip2 can use the same schedule to compress many different files, as long as the number of threads and the number of file blocks remain the same.

3 Overview

Our design of TERN adheres to the following goals:

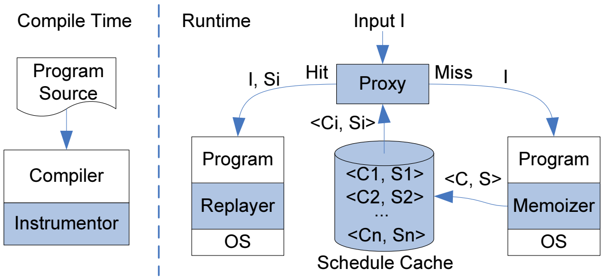

Figure 2: TERN *architecture.* Its components are shaded.

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 TERN's architecture (§3.1), workflow (§3.2), deployment scenarios (§3.3), and limitations (§3.4).

3.1 Architecture

Figure 2 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 §4). The instrumentor runs as a plugin to LLVM [3], 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 §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.

```
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: Simplified PBZip2 code.

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 [17]. 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.

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.

3.2 Workflow and An Example

We illustrate how TERN works using PBZip2 as an example. Figure 3 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 threads, splits the file

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

Figure 4: 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 5: Input constraints of a PBZip2 run.

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 4. It also collects constraints as shown in Figure 5, 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 4) to compress the file, even though the file data may differ completely.

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.

3.4 Limitations

Determinism. TERN aims for best-effort determinism for reasons discussed in §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 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 §7.4 for how TERN reduces this overhead and §8.1 for an evaluation of this overhead). This overhead may be further reduced using simple static analysis.

4 Interface

Table 2 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 (§6); (3) wrap-

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

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. Often used to divide the executions of threads in a pool into separate tasks (§6).

lock_wrapper(*l*)
unlock_wrapper(*l*)
	Developer or TERN	Synchronization wrappers. The memoizer intercepts these operations for memoizing schedules, and the replayer intercepts them for reusing schedules.

before_blocking()
after_blocking()
	TERN	Inserted before and after blocking system calls. The memoizer logs the order of these calls. The replayer opportunistically enforces the same order of these calls.

Table 2: 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 (§5).

pers 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.

Currently TERN hooks 28 pthread 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.

5 Schedule Memoization

This section presents the idea of schedule memoization in the context of batch programs. We describe how TERN memoizes schedules (§5.1), tracks input constraints (§5.2), merges a schedule into the schedule cache (§5.3), and reuses schedules (§5.4).

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 wrapper functions in Table 2. Figure 6 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).

```
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(!pthread_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 6: *The memoizer's round-robin scheduling algorithm.*

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 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 non-determinism, 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 6. 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 (§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.

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 §7.4.

TERN leverages KLEE [17], 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.

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 calls `symbolic()` twice.

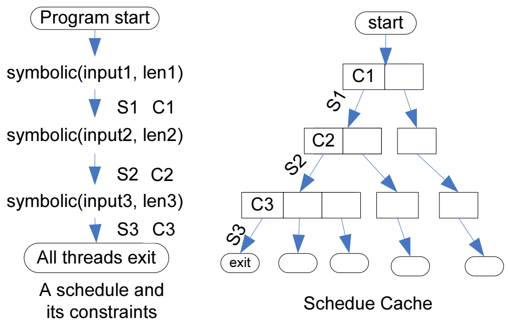

Figure 7: *Decision tree of* TERN*'s schedule cache.*

Figure 7 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.

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. 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.

```
1 :  pthread_mutex_lock_wrapper(mutex) {
2 :     down(sem[self()]); // wait for our turn
3 :     pthread_mutex_lock(mutex);
4 :     next = shift schedule; // find next thread in schedule
5 :     up(sem[next]); // wake up next thread
6 :  }
```

Figure 8: *Pseudo code of the replayer.*

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 8 illustrates semaphore relay using the replayer's `pthread_mutex_lock()` wrapper.

We note several subtleties of the pseudo code in Figure 8. First, we do not use non-blocking lock operations (line 3) as in Figure 6 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.

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 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.

7 Refinements

This section describes four refinements we made, one for determinism (§7.1) and three for speed (§7.2-§7.4).

7.1 Detecting Data Races

As discussed in §2.2, if a memoized schedule allows data races, runs reusing this schedule may become nondeterministic. Thus, for determinism, we would like to de-

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

Figure 9: *A conventional race, not a schedule race.*

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

Figure 10: *A symbolic race that occurs only when $i = j$.*

tect 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 [46] (call them *schedule races*). Figure 9 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 10 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 10, 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 10, 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.

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 (§8.3).

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.

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 [8] 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 (§8.3). However, we note that our analysis is unsound because it ignores data dependencies. Thus, we plan to implement a sound slicing algorithm [21] in our future work.

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: *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.

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 SPLASH2.[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 (§8.1)?
2. Does TERN make multithreaded programs stable across different inputs (§8.2)?
3. Does TERN incur high overhead (§8.3)?
4. Does TERN make multithreaded programs deterministic on the same input (§8.4)?

8.1 Ease of Use

Table 3 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 (§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 SPLASH2 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.

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

Table 4: *Bug stability results on SPLASH2* `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.

8.2 Stability

We evaluated TERN's stability via two sets of experiments. The first set compares it to an existing DMT system (§8.2.1), the second quantifies how frequently it can reuse schedules on real and synthetic workloads (§8.2.2).

8.2.1 Bug Stability

We compared TERN to COREDET [13] 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 SPLASH2 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.

We present only the fft results; the results of the other programs are similar. Table 4 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

[2] The version of the SPLASH2 [36] 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-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 5: TERN *stability.* Column **Schedules** indicates the number of schedules in the schedule cache.

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.

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 [7] 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.

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 5 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 (§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.

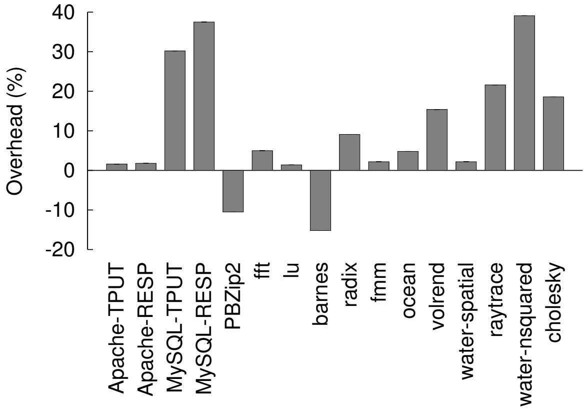

Figure 11: *Relative overhead of the replayer over nondeterministic execution.* Negative overhead means speedup.

8.3 Overhead

We used the following workloads to evaluate TERN's overhead. For Apache, we used ApacheBench [1] 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 SPLASH2 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. Table 11 shows the relative overhead of the replayer over nondeterministic 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 (§7.2). Figure 12 shows the results. This optimization significantly reduced the replayer's overhead for four programs. Specifically, it

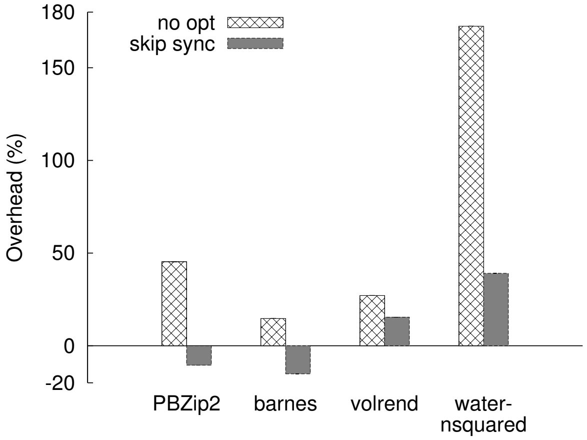

Figure 12: *Overhead reduction by skipping unnecessary synchronizations.* “no opt” indicates the baseline overhead.

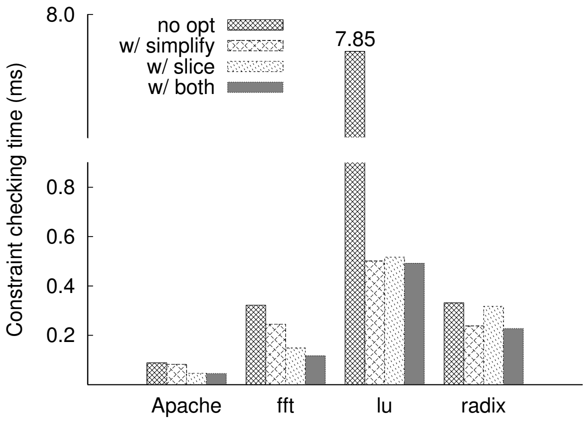

Figure 13: *Optimizations to speed up constraint checking.* Note the y-axis is broken. “no opt” indicates the baseline constraint checking time. “simplify” refers to the optimization in §7.3. “slice” refers to the optimization in §7.4.

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 6: *Overhead of the memoizer.*

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 7: *Concurrency errors used in evaluation.*

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.

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 (§7.3 and §7.4). Figure 13 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 6 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 [16], which we plan to implement in our future work.

8.4 Determinism

We evaluated TERN’s determinism via three sets of experiments. The first set checked the memoized schedules for races (§8.4.1). The second evaluated TERN’s ability to deterministically reproduce or avoid bugs (§8.4.2). The third measured how deterministic memory accesses are with and without TERN (§8.4.3).

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 free of schedule races and symbolic races with respect to the symbolic data we annotated (§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 §7.1.

8.4.2 Bug Determinism

We also evaluated how deterministically TERN could reproduce or avoid bugs. Table 7 lists five real concurrency bugs we used. We selected them because they were frequently used in previous studies [37, 39, 43, 44] and we could reproduce them on our evaluation machine. To measure bug determinism, we first memoized schedules for programs listed in Table 7. We then manually inserted `usleep()` to these programs to get alternate schedules.

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

Table 8: *Memory access determinism.* We traced memory accessed only from PBZip2, not the external BZip2 library.

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.

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 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 [52]. 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 8 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.

9 Related Work

Deterministic Execution TERN differs from existing DMT systems [13, 22, 41] by making threads stable, *i.e.*, repeating familiar behaviors across different inputs. Another difference is that TERN reduces timing nondeterminism for server programs through windowing.

The closest system to TERN in this category is Kendo [41], a software-only DMT system that also enforces synchronization orders instead of memory access orders for efficiency. COREDET [13] 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 SPLASH2. 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 [22], a hardware DMT system .

Grace [14] 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 [45] and a Cilk [15] benchmark. Unlike Grace, TERN aims to make general multithreaded programs, not just fork-join programs, deterministic and stable.

Deterministic Replay Deterministic replay [9, 23, 24, 27, 31, 33, 34, 40, 44, 50, 51] 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 [34]; Capo, which defines a novel software-hardware interface and a set of abstractions for efficient replay [40]; PRES and ODR, which systematically search for a complete execution based on a partial one [9, 44]; and SMP-ReVirt, which uses clever page protection trick for recording the order of conflicting memory accesses [24].

Concurrency Errors The complexity in developing multithreaded programs has led to many concurrency errors [39]. A significant number of them are not data races, but atomicity and order errors [39], which can be deterministically reproduced or avoided using only synchronization orders.

Much work exists on concurrency error detection [25, 37, 38, 47, 55, 56], diagnosis [42, 43, 48], and correction [32, 53]. 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 [16–18, 20, 28–30, 49, 54], block malicious inputs [21], and preserve privacy in error reports [19]. Compared to these systems, TERN applies symbolic execution to a new domain: tracking input constraints to reuse schedules.

10 Conclusion

We have presented TERN, the first DMT 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. TERN is the first DMT system shown to work on applications as large, complex, and nondeterministic as MySQL and Apache. It demonstrates that DMT has the potential to be deployed today.

Acknowledgement

We thank Cristian Cadar, John Gallagher, Michael Kester, Emery Berger (our shepherd), and the anonymous reviewers for their tremendous feedback and comments, which have substantially improved the content and presentation of this paper. We thank Shan Lu for providing some of the concurrency errors used in our evaluation. Michael Kester wrote the script for replaying the HTTP trace from the Columbia CS website.

This work was supported by the National Science Foundation (NSF) through Contract CNS-1012633 and CNS-0905246 and the Air Force Research Laboratory (AFRL) through Contract FA8650-10-C-7024. Opinions, findings, conclusions, and recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the US Government.

References

[1] ab - Apache HTTP server benchmarking tool. `http://httpd.apache.org/docs/2.2/programs/ab.html`.

[2] Artici Terns - Wikipedia. `http://en.wikipedia.org/wiki/Arctic_Tern`.

[3] The LLVM Compiler Framework. `http://llvm.org`.

[4] MySQL Database. `http://www.mysql.com/`.

[5] Parallel BZIP2 (PBZIP2). `http://compression.ca/pbzip2/`.

[6] Stanford Parallel Applications for Shared Memory (SPLASH). `http://www-flash.stanford.edu/apps/SPLASH/`.

[7] SysBench: a system performance benchmark. `http://sysbench.sourceforge.net`.

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

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

[10] Apache Web Server. `http://www.apache.org`.

[11] T. Ball and J. R. Larus. Branch prediction for free. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 300–313, 1993.

[12] T. Ball and J. R. Larus. Efficient path profiling. In *MICRO 29: Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 46–57, 1996.

[13] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. Coredet: a compiler and runtime system for deterministic multithreaded execution. In *Fifteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '10')*, pages 53–64, 2010.

[14] 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) 2009*, 2009.

[15] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. *J. Parallel Distrib. Comput.*, 37(1):55–69, 1996.

[16] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. 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, Oct.–Nov. 2006.

[17] C. Cadar, D. Dunbar, and D. 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, Dec. 2008.

[18] G. Candea, S. Bucur, and C. Zamfir. Automated software testing as a service. In *Proceedings of the 1st Symposium on Cloud Computing (SOCC '10)*, 2010.

[19] M. Castro, M. Costa, and J.-P. Martin. Better bug reporting with better privacy. In *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 319–328, 2008.

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

[21] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado. Bouncer: securing software by blocking bad input. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 117–130, 2007.

[22] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. Dmp: deterministic shared memory multiprocessing. In *Fourteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '09)*, pages 85–96, 2009.

[23] G. Dunlap, S. T. King, S. Cinar, M. Basrat, and P. 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)*, Dec. 2002.

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

[25] D. Engler and K. 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, Oct. 2003.

[26] J. A. Fisher and S. M. Freudenberger. Predicting conditional branch directions from previous runs of a program. In *ASPLOS-V: Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*, pages 85–95, 1992.

[27] D. Geels, G. Altekarz, P. Maniatis, T. Roscoey, and I. Stoicaz. Friday: Global comprehension for distributed replay. In *Proceedings of the Fourth Symposium on Networked Systems Design and*

Implementation (NSDI '07), Apr. 2007.

[28] P. Godefroid, N. Klarlund, and K. 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.

[29] P. Godefroid, A. Kiezun, and M. 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.

[30] P. Godefroid, M. Levin, and D. Molnar. Automated whitebox fuzz testing. In *NDSS '08: Proceedings of 15th Network and Distributed System Security Symposium*, Feb. 2008.

[31] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. 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, Dec. 2008.

[32] H. Jula, D. Tralamazza, Z. Cristian, and C. 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, Dec. 2008.

[33] R. Konuru, H. Srinivasan, and J.-D. 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.

[34] O. Laadan, N. Viennot, and J. Nieh. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In *Proceedings of the 2010 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 2010.

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

[36] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. 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.

[37] S. Lu, J. Tucek, F. Qin, and Y. 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, Oct. 2006.

[38] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. Muvi: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. *SIGOPS Oper. Syst. Rev.*, 41(6):103–116, 2007.

[39] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 329–339, 2008.

[40] P. Montesinos, M. Hicks, S. T. King, and J. 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, 2009.

[41] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: efficient deterministic multithreading in software. In *Fourteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '09)*, pages 97–108, 2009.

[42] C.-S. Park and K. 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, Nov. 2008.

[43] S. Park, S. Lu, and Y. 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, Mar. 2009.

[44] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. 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, 2009.

[45] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24, 2007.

[46] M. Ronsse and K. De Bosschere. Recplay: a fully integrated practical record/replay system. *ACM Trans. Comput. Syst.*, 17 (2):133–152, 1999.

[47] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multithreaded programming. *ACM Transactions on Computer Systems*, pages 391–411, Nov. 1997.

[48] K. 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.

[49] K. Sen, D. Marinov, and G. 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, Sept. 2005.

[50] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. 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.

[51] VMWare Virtual Lab Automation. `http://www.vmware.com/solutions/vla/`.

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

[53] Y. Wang, T. Kelly, M. Kudlur, S. Lafortune, and S. 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, Dec. 2008.

[54] J. Yang, C. Sar, P. Twohey, C. Cadar, and D. Engler. Automatically generating malicious disks using symbolic execution. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy (SP '06)*, pages 243–257, May 2006.

[55] Y. Yu, T. Rodeheffer, and W. Chen. RaceTrack: efficient detection of data race conditions via adaptive tracking. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 221–234, 2005.

[56] W. Zhang, C. Sun, and S. 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, 2010.