PARROT: A Practical Runtime for Deterministic, Stable, and Reliable Threads

Heming Cui[+], Jiri Simsa[*], Yi-Hong Lin[+], Hao Li[+], Ben Blum[*], Xinan Xu[+],
Junfeng Yang[+], Garth A. Gibson[*], and Randal E. Bryant[*]
[+]Columbia University [*]Carnegie Mellon University

Abstract

Multithreaded programs are hard to get right. A key reason is that the contract between developers and runtimes grants exponentially many schedules to the runtimes. We present PARROT, a simple, practical runtime with a new contract to developers. By default, it orders thread synchronizations in the well-defined round-robin order, vastly reducing schedules to provide determinism (more precisely, deterministic synchronizations) and *stability* (i.e., robustness against input or code perturbations, a more useful property than determinism). When default schedules are slow, it allows developers to write intuitive *performance hints* in their code to switch or add schedules for speed. We believe this "meet in the middle" contract eases writing correct, efficient programs.

We further present an ecosystem formed by integrating PARROT with a model checker called DBUG. This ecosystem is more effective than either system alone: DBUG checks the schedules that matter to PARROT, and PARROT greatly increases the coverage of DBUG.

Results on a diverse set of 108 programs, roughly 10× more than any prior evaluation, show that PARROT is easy to use (averaging 1.2 lines of hints per program); achieves low overhead (6.9% for 55 real-world programs and 12.7% for all 108 programs), 10× better than two prior systems; scales well to the maximum allowed cores on a 24-core server and to different scales/types of workloads; and increases DBUG's coverage by 10^6–10^{19734} for 56 programs. PARROT's source code, entire benchmark suite, and raw results are available at `github.com/columbia/smt-mc`.

Categories and Subject Descriptors: D.4.5 [**Operating Systems**]: Threads, Reliability; D.2.4 [**Software Engineering**]: Software/Program Verification;

General Terms: Algorithms, Design, Reliability, Performance

Keywords: Deterministic Multithreading, Stable Multithreading, Software Model Checking, State Space Reduction

SOSP'13, Nov. 3–6, 2013, Farmington, Pennsylvania, USA.
ACM 978-1-4503-2388-8/13/11.
http://dx.doi.org/10.1145/2517349.2522735

1 Introduction

1.1 Background

Our accelerating computational demand and the rise of multicore hardware have made multithreaded programs pervasive and critical. Yet, these programs remain extremely difficult to write, test, analyze, debug, and verify. A key reason is that, for decades, the contract between developers and thread runtimes has favored performance over correctness. In this contract, developers use synchronizations to coordinate threads, while thread runtimes can use *any* of the exponentially many thread interleavings, or *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 missed schedule may surface in the least expected moment, causing critical failures [33, 37, 52, 68].

Several recent systems aim to flip this performance vs correctness tradeoff by reducing the number of allowed schedules. Among them, *deterministic multithreading* (DMT) systems [6, 9, 11, 12, 16–18, 34, 43] reduce schedules by mapping each input to only one schedule, so that executions of the same program on the same input always show the same behavior. DMT is commonly believed to simplify testing, debugging, record-replay, and replication of multithreaded programs.

However, we argue that determinism is not as useful as commonly perceived: it is neither sufficient nor necessary for reliability [69, 70]. It is not sufficient because it is quite narrow ("same input + same program = same behavior") and has no jurisdiction if the input or program changes however slightly. For instance, a perfectly deterministic system can map each input to an arbitrary schedule, so that flipping an input bit or adding a line of debug code leads to vastly different schedules, artificially reducing the program's robustness against input and program perturbations, or *stability*. Yet stability is crucial for reliability (e.g., after testing some inputs, developers often anticipate that the program work on many similar inputs). Determinism is not necessary for reliability because a nondeterministic system with a small set of schedules for all inputs can be made reliable by exhaustively checking all schedules.

We propose a better approach we call *stable multi-*

threading (StableMT) [69, 70] that reuses each schedule on a wide range of inputs, mapping all inputs to a dramatically reduced set of schedules. For instance, under most setups, StableMT reduces the number of schedules needed by parallel compression utility `PBZip2` down to *two* schedules for each different number of threads, regardless of the file contents [17]. By vastly shrinking the set of schedules, StableMT makes it extremely easy to find the schedules that cause concurrency bugs. Specifically, StableMT greatly increases the coverage of tools that systematically test schedules for bugs [22, 41, 57, 67]. It also greatly improves the precision and simplicity of program analysis [63], verification [63], and debugging, which can now focus on a much smaller set of schedules. Moreover, StableMT makes programs robust against small input or program perturbations, bringing stability into multithreading.

StableMT is not mutually exclusive with DMT. Grace [12], TERN [16], Determinator [6], PEREGRINE [17], and DTHREADS [34] may all be classified as both deterministic and stable. Prior work [6, 16, 17, 34], including ours [16, 17], conflated determinism and stability, but they are actually separate properties.

Figure 1 compares traditional multithreading, DMT, and StableMT. Figure 1a depicts traditional multithreading, a conceptual many-to-many mapping between inputs and schedules. Figure 1b depicts a DMT system that maps each input to an arbitrary schedule, artificially destabilizing program behaviors. Figures 1c and 1d depict two StableMT systems: the many-to-one mapping in Figure 1c is deterministic, while the many-to-few mapping in Figure 1d is nondeterministic. A many-to-few mapping improves performance by giving the runtime choices, but it increases the checking effort needed for reliability. Fortunately, the choice of schedules is minimal, so that tools can easily achieve high coverage.

1.2 Challenges

Although the latest advances are promising, two important challenges remain unaddressed. First, can the DMT and StableMT approaches consistently achieve good performance on a wide range of programs? For instance, we observed that a prior system, DTHREADS, had 5× to 100× slowdown on some programs. Second, can they be made simple and adoptable? These challenges are not helped much by the limited evaluation of prior 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 §8).

These challenges are intermingled. 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 (§7.3). To recover performance, one method is to count the instructions executed by each thread and select schedules that balance the instruction counts [9, 18, 43], 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 [16, 17], but it requires sophisticated program analysis, complicating deployment.

1.3 Contributions

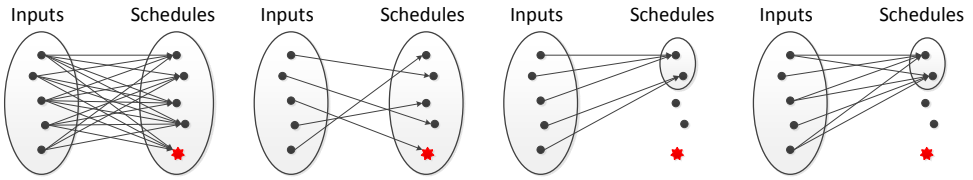
This paper makes three contributions. First, we present PARROT,[1] a simple, practical runtime that efficiently makes threads deterministic and stable by offering a new contract to developers. By default, it schedules synchronizations in each thread using round-robin, vastly reducing schedules and providing 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 prior systems, PARROT's contract reduces schedules to favor correctness over performance. Unlike prior systems, it allows advanced developers to optimize performance. We believe this practical "meet in the middle" contract eases writing correct, efficient programs.

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 (§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 [5, 19, 46].

Our PARROT implementation is Pthreads-compatible,

[1] We name our system after one of the most trainable birds.



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

Figure 1: *Different multithreading approaches.* Stars in red represent schedules that cause concurrency bugs.

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 (§6). PARROT's determinism is similar to Kendo's weak determinism [43], but PARROT offers stability which Kendo lacks.

Our second contribution is an ecosystem formed by integrating PARROT with DBUG [57], 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 [21, 23].

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 [13], OpenLDAP [45], Redis [54], MPlayer [39], all 33 parallel C++ STL algorithm implementations [59] which use OpenMP, and all 14 parallel image processing utilities (also OpenMP) in the ImageMagick [26] software suite. Further, they include *all* 53 programs in four widely used benchmark suites: PARSEC [2], Phoenix [53], SPLASH-2x [58], and NPB [42]. 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 [14], fork-join, map-reduce, pipeline, and workpile. To our knowledge, our evaluation uses roughly 10× more programs than any prior DMT or StableMT evaluation, and 4× more than all prior 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 prior systems DTHREADS [34] and COREDET [9] 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 results [1]. In the remaining of this paper, §2 contrasts PARROT with prior systems on an example and gives an overview of PARROT's architecture. §3 describes the performance hint abstractions PARROT provides, §4 the PARROT runtime, and §5 the PARROT-DBUG ecosystem. §6 discusses PARROT's determinism, §7 presents evaluation results, §8 discusses related work, and §9 concludes.

2 Overview

This section first compares two prior systems and PARROT using an example (§2.1), and then describes PARROT's architecture (§2.2).

2.1 An Example

Figure 2 shows the example, a simplified version of the parallel compression utility `PBZip2` [3]. It uses the common producer-consumer idiom: the producer (main)

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

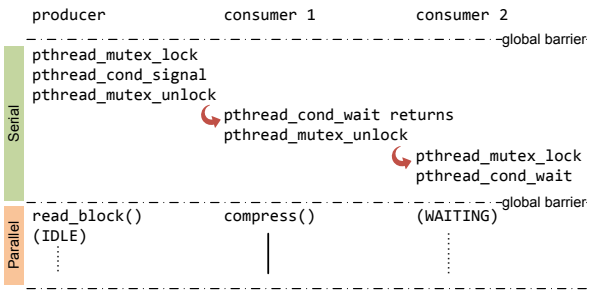


Figure 3: *A* DTHREADS *schedule.* All `compress` calls are serialized. `read_block` runs much faster than `compress`.

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

COREDET [9] represents DMT systems that balance load by counting instructions each thread has run [9, 10, 18, 25, 43]. 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 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.

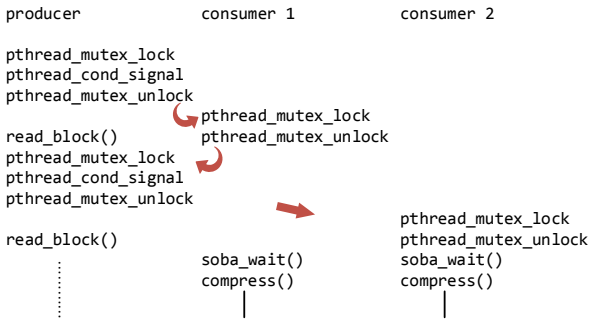


Figure 4: *A* PARROT *schedule with performance hints.*

DTHREADS [34] 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 with two threads, we observed the DTHREADS schedule in Figure 3. 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`

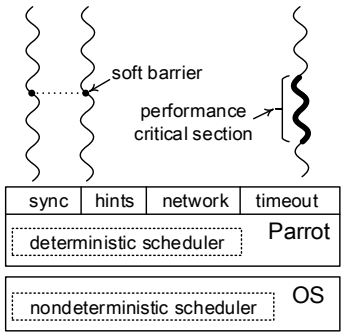


Figure 5: PARROT *architecture.*

threads, and line 23 is the starting point of coscheduling. With these hints, PARROT switched to the schedule in Figure 4 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`).

2.2 Architecture

Figure 5 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.

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.

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 which calls `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 (§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 [46]. 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 [19] for coscheduling purposes because schedulers cannot ignore a real barrier.

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.

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 [2] 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();`").

4 PARROT Runtime

The PARROT runtime contains implementation of the hint abstractions (§4.3) and a set of wrapper functions that intercept Pthreads (§4.2), network (§4.4), and timeout (§4.5) operations. The wrappers interpose dynamically loaded library calls via `LD_PRELOAD` and "trap" the calls into PARROT's deterministic scheduler (§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).

```
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 1: ***Scheduler primitives.***

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 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)`

```
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 6: *Wrappers of Pthreads mutex lock&unlock.*

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 §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 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. 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

```
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 7: *Wrapper of* `pthread_cond_wait`.

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.

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 6 shows the pseudo code of our Pthreads mutex lock and unlock wrappers. Both are 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 7 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.

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 §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.

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`).

4.5 Timeouts

Real-world programs frequently 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 (§7.6). Of course, PARROT's logical time corresponds loosely to real time, and may be less useful for real-time applications.[2]

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 [10, 67]. 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 common timed operations such 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 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 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 [31, 40, 65] to guide the exploration toward executions deemed more interesting, but heuristics have a risk of missing bugs. *State-space reduction* techniques [21–23] 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

[2] dOS [10] discussed the possibility of converting real time to logical time but did not present how.

check only one of them. Unfortunately, equivalence is rare and extremely challenging to find, especially for *implementation-level* model checkers which check implementations directly [22, 31, 40, 57, 65, 66]. This difficulty is reflected in the existence of only two main reduction techniques [21, 23] 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 efforts, state-space explosion remains the bane of model checking.

As discussed in §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 (§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.

5.1 The DBUG Model Checker

In principle, PARROT can be integrated with many model checkers. We chose DBUG [57] 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) [21], 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* [30], an online variant of Knuth's offline technique for tree size estimation [32]. 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.

5.2 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 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 (§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.

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 [6, 12, 34]. 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 [34]. Implementing it can leverage existing code [34].

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);`") [64] which require manual annotations anyway in some prior systems that make races deterministic [12, 34]. 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, prior work has shown that data races rarely occur if a synchronization schedule is enforced. For instance, PEREGRINE [17] 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 [49]). Similarly, we can detect the races by model checking a small set of schedules [41]. In short, by vastly reducing schedules, StableMT makes the problems caused by nondeterminism easy to solve.

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 [13]; OpenLDAP, a server implementing the Lightweight Directory Access Protocol [45]; Redis, a fast key-value data store server [54]; MPlayer, a popular media encoder, decoder, and player [39]; `PBZip2`, a parallel compression utility [3]; `pfscan`, a parallel `grep`-like utility [51]; `aget`, a parallel file download utility [4]; all 33 parallel C++ STL algorithm implementations [59] which use OpenMP; all 14 parallel image processing utilities (which also use OpenMP) in the ImageMagick software suite [26] 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 [2], 14 in Phoenix [53], 14 in SPLASH-2x [58], and 10 in NPB [42]. 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 prior DMT or StableMT evaluation, and 4× more than all prior 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` [8], 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 [64], 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` (§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 [1].)

The rest of this section focuses on six questions:

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 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.

§7.1: Is PARROT easy to use? How many hints are needed to make the programs with PARROT fast?
§7.2: Is PARROT fast? How effective are the hints?
§7.3: How does it compare to prior systems?
§7.4: How does its performance vary according to core counts and workload scales/types?
§7.5: Is it deterministic in the absence of data races?
§7.6: How much does it improve DBUG's coverage?

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 hints in `libgomp`, and 38 need program-specific soft barriers (Table 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 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` [60] and Linux `perf` [50] 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.

7.2 Performance

Figure 8 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 starting 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 (§4.1). `bodytrack-openmp` and `facesim` enjoyed a similar benefit. So did another 19 programs which had 10× fewer context switches with PARROT [1]. `linear-regression-pthread`'s stalled cycles were reduced by 10× with PARROT, and we speculate that PARROT's scheduler improved its affinity. (See [1] for all results on microarchitectural events.)

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 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.

Figure 9 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%

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 4: *Soft barrier successes and timeouts.*

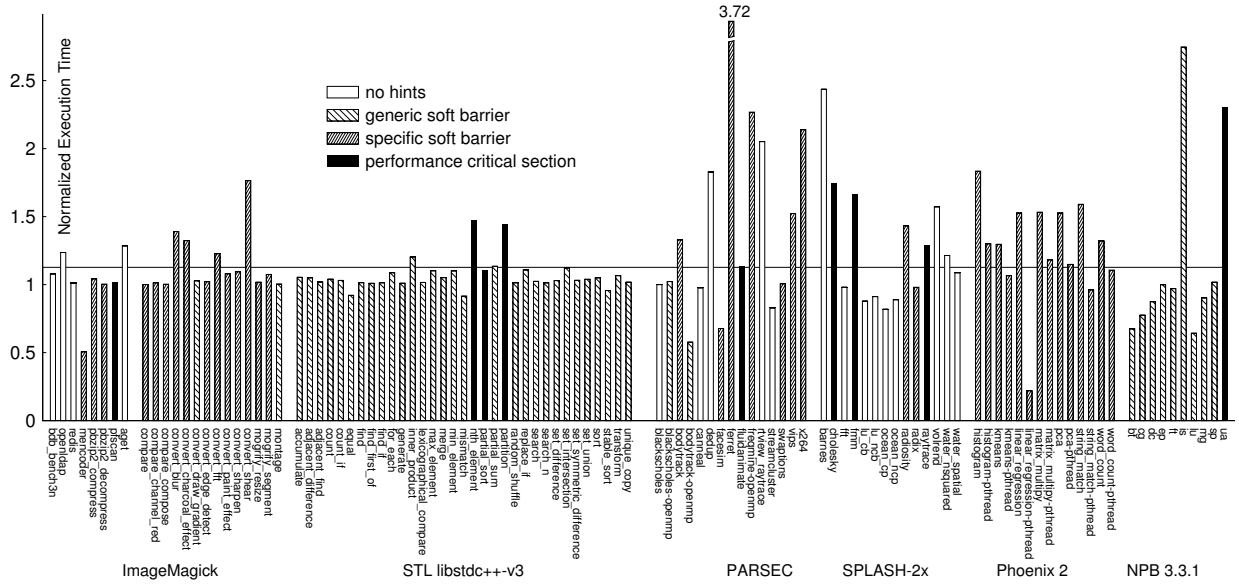


Figure 8: PARROT*'s performance normalized over nondeterministic execution.* The patterns of the bars show the types of the hints the programs need: no hints, generic soft barriers in `libgomp`, program-specific soft barriers, or performance critical sections. The mean overhead is 12.7% (indicated by the horizontal line).

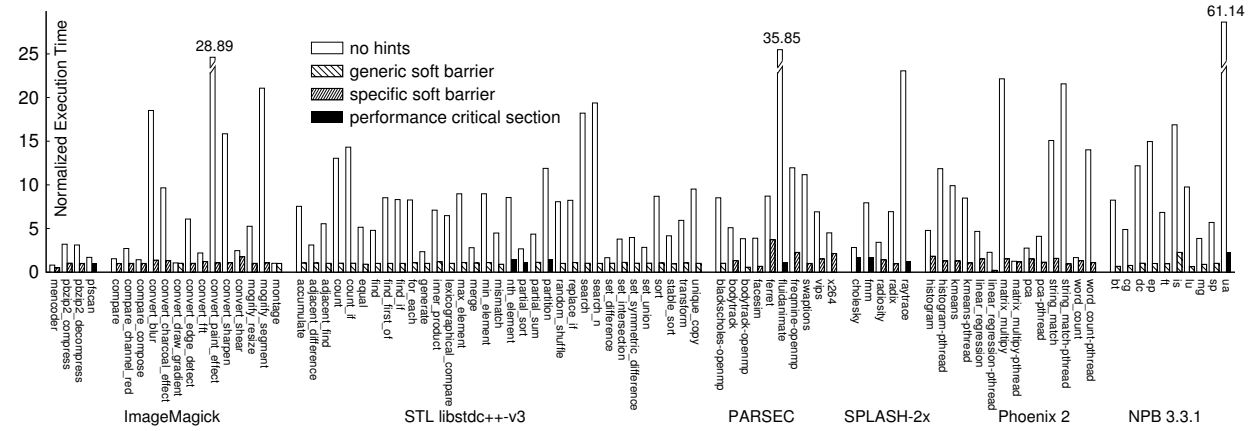


Figure 9: *Effects of performance hints.* They reduced PARROT's overhead from 510% to 11.9%.

after hints were added. The four lines of generic soft barrier hints in `libgomp` (Table 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 performance critical sections from 830% to 42.1% for 9 programs. Soft barriers timed out on 12 programs (Table 4), which affected neither determinism nor correctness. The `kmeans` experienced over 10% timeouts, causing higher overhead. `x264` experienced many timeouts but enjoyed partial coscheduling benefits (§3).

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,[3] 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 10 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`,

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

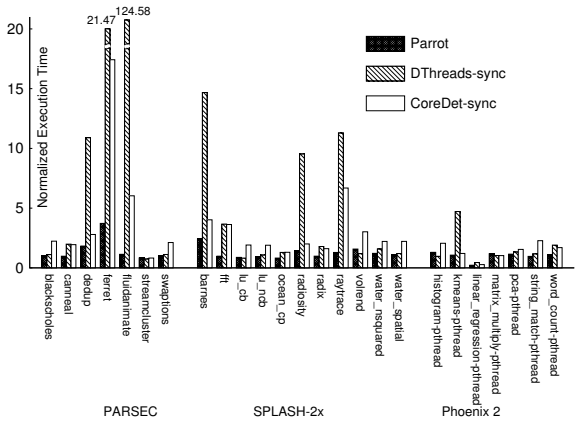


Figure 10: PARROT, DTHREADS, *and* COREDET *overhead.*

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.

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 [1] for detailed scalability results.)

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 [64] can solve this problem. We also evaluated PARROT's determinism using a modified version of `racey` [24] 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.

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

7.6 Model Checking Coverage

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` [38]). We enabled state-of-the-art DPOR [21] 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 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 6 shows the results for all 11 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

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	1,201s
`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 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.

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, 6 in Table 6). It increased the number of verified programs from 43 to 99 (95 programs in Table 5, 4 in Table 6).

8 Related Work

DMT and StableMT systems. Conceptually, prior work [6, 16, 17, 34], including ours [16, 17], conflated determinism and stability. To our knowledge, we are the first to distinguish these two separate properties [69, 70].

Implementation-wise, several prior systems are not backward-compatible because they require new hardware [18], new language [15], or new programming model and OS [6]. Among backward-compatible systems, some DMT systems, including Kendo [43], COREDET [9], and COREDET-related systems [10, 25], improve performance by balancing each thread's load with low-level instruction counts, so they are not stable.

Five systems can be classified as StableMT systems. Our TERN [16] and PEREGRINE [17] systems require sophisticated program analysis to determine input and schedule compatibility, complicating deployment. Bergan *et al* [11] 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 [12] and DTHREADS [34] ignore thread load imbalance, so they are prone to the serialization problem (§2.1). Grace also requires fork-join parallelism.

Compared to PARROT's evaluation, prior evaluations have several limitations. First, prior work has reported results on a narrow 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.[4]

Lastly, little prior work except ours [63] 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., [21–23]). Partial order reduction [21, 22] has been the main reduction technique for model checkers that check implementations directly [57, 67]. 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 [23] 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): (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) [27] assumes all shared memory is implicitly protected and allows advanced developers the flexibility to remove protection. It thus shares a 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 prior work on concurrency error detection [20, 35, 36, 55, 71, 72], diagnosis [47, 48, 56], and correction [28, 29, 61, 62]. By reducing schedules, it potentially benefits all these techniques.

9 Conclusion

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 prior 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`.

Acknowledgments

We thank Tom Bergan, Emery Berger, Luis Ceze, Bryan Ford (our shepherd), Shan Lu, Martin Vechev, Emmett Witchel, and the anonymous reviewers for their many helpful comments. This work was supported in part by AFRL FA8650-11-C-7190, FA8650-10-C-7024, and FA8750-10-2-0253; ONR N00014-12-1-0166; NSF CCF-1162021, CNS-1117805, CNS-1054906, and CNS-0905246; NSF CAREER; AFOSR YIP; Sloan Research Fellowship; Intel Science and Technology Center for Cloud Computing; ARO W911NF0910273; MSR-CMU Computational Thinking Center; and members of the CMU PDL Consortium.

[4] Three exceptions used more than 8 cores: [44] (ran a 12-line program on 48 cores), [7] (ran 9 selected programs from PARSEC, SPLASH-2x, and NPB on 32 cores), and [18] (emulated 16 cores).

References

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

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

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

[4] Aget. http://www.enderunix.org/aget/.

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

[6] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient system-enforced deterministic parallelism. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI '10)*, Oct. 2010.

[7] A. F. Aviram. *Deterministic OpenMP*. PhD thesis, Yale University, 2012.

[8] Bench3n. http://libdb.wordpress.com/3n1/.

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

[10] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble. Deterministic process groups in dOS. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI '10)*, Oct. 2010.

[11] T. Bergan, L. Ceze, and D. Grossman. Input-covering schedules for multithreaded programs. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '13)*, Oct. 2013.

[12] 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, Oct. 2009.

[13] Berkeley DB. http://www.sleepycat.com.

[14] O. A. R. Board. OpenMP application program interface version 3.0, May 2008.

[15] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel java. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '09)*, pages 97–116, Oct. 2009.

[16] H. Cui, J. Wu, C.-C. Tsai, and J. Yang. Stable deterministic multithreading through schedule memoization. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI '10)*, Oct. 2010.

[17] H. Cui, J. Wu, J. Gallagher, H. Guo, and J. Yang. Efficient deterministic multithreading through schedule relaxation. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, pages 337–351, Oct. 2011.

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

[19] A. C. Dusseau, R. H. Arpaci, and D. 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.

[20] 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.

[21] C. Flanagan and P. 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, Jan. 2005.

[22] 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, Jan. 1997.

[23] H. Guo, M. Wu, L. Zhou, G. Hu, J. Yang, and L. 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, Oct. 2011.

[24] M. D. Hill and M. Xu. Racey: A stress test for deterministic execution. `http://www.cs.wisc.edu/~markhill/racey.html`.

[25] N. Hunt, T. Bergan, , L. Ceze, and S. 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.

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

[27] M. Isard and A. 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.

[28] G. Jin, W. Zhang, D. Deng, B. Liblit, and S. Lu. Automated concurrency-bug fixing. In *Proceedings of the Tenth Symposium on Operating Systems Design and Implementation (OSDI '12)*, pages 221–236, 2012.

[29] 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.

[30] P. Kilby, J. K. Slaney, S. Thiébaux, and T. Walsh. Estimating search tree size. In *Proceedings of the 21st national conference on Artificial intelligence (AAAI '06)*, pages 1014–1019, 2006.

[31] C. Killian, J. W. Anderson, R. Jhala, and A. 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, Apr. 2007.

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

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

[34] T. Liu, C. Curtsinger, and E. D. Berger. DTHREADS: efficient deterministic multithreading. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, pages 327–336, Oct. 2011.

[35] 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.

[36] 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. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP '07)*, pages 103–116, 2007.

[37] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Thirteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '08)*, pages 329–339, Mar. 2008.

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

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

[40] M. Musuvathi, D. Y. Park, A. Chou, D. R. Engler, and D. 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, Dec. 2002.

[41] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the Eighth Symposium on Operating Systems Design and Implementation (OSDI '08)*, pages 267–280, Dec. 2008.

[42] NASA Parallel Benchmarks. `http://www.nas.nasa.gov/publications/npb.html`.

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

[44] M. Olszewski, J. Ansel, and S. Amarasinghe. Scaling deterministic multithreading. In *The 2nd Workshop on Determinism and Correctness in Parallel Programming (WODET '11)*, Mar. 2011.

[45] OpenLDAP. `http://www.openldap.org/`.

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

[47] 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.

[48] 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.

[49] 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, Oct. 2009.

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

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

[52] K. Poulsen. Software bug contributed to blackout. `http://www.securityfocus.com/news/8016`, Feb. 2004.

[53] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. 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.

[54] Redis. `http://redis.io/`.

[55] 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.

[56] 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.

[57] J. Simsa, G. Gibson, and R. 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.

[58] SPLASH-2x. `http://parsec.cs.princeton.edu/parsec3-doc.htm`.

[59] STL Parallel Mode. `http://gcc.gnu.org/onlinedocs/libstdc++/manual/parallel_mode.html`.

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

[61] 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.

[62] J. Wu, H. Cui, and J. Yang. Bypassing races in live applications with execution filters. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI '10)*, Oct. 2010.

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

[64] W. Xiong, S. Park, J. Zhang, Y. Zhou, and Z. Ma. Ad hoc synchronization considered harmful. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI '10)*, Oct. 2010.

[65] J. Yang, P. Twohey, D. Engler, and M. 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, Dec. 2004.

[66] J. Yang, C. Sar, and D. 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, Nov. 2006.

[67] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. 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, Apr. 2009.

[68] J. Yang, A. Cui, S. Stolfo, and S. Sethumadhavan. Concurrency attacks. In *the Fourth USENIX Workshop on Hot Topics in Parallelism (HOTPAR '12)*, June 2012.

[69] J. Yang, H. Cui, and J. Wu. Determinism is overrated: What really makes multithreaded programs hard to get right and what can be done about it? In *the Fifth USENIX Workshop on Hot Topics in Parallelism (HOTPAR '13)*, June 2013.

[70] J. Yang, H. Cui, J. Wu, Y. Tang, and G. Hu. Determinism is not enough: Making parallel programs reliable with stable multithreading. *Communications of the ACM*, 2014.

[71] Y. Yu, T. Rodeheffer, and W. 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, Oct. 2005.

[72] 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, Mar. 2010.