Semantics-Aware Scheduling Policies for Synchronization Determinism

Qi Zhao
Department of Computer Science
North Carolina State University
qzhao6@ncsu.edu

Zhengyi Qiu
Department of Computer Science
North Carolina State University
zqiu2@ncsu.edu

Guoliang Jin
Department of Computer Science
North Carolina State University
guoliang_jin@ncsu.edu

Abstract

A common task for all deterministic multithreading (DMT) systems is to enforce synchronization determinism. However, synchronization determinism has not been the focus of existing DMT research. Instead, most DMT systems focused on how to order data races remained after synchronization determinism is enforced. Consequently, existing scheduling policies for synchronization determinism all have limitations. They may either require performance annotations to achieve good performance or fail to provide schedule stability.

In this paper, we argue that synchronization determinism is more fundamental to DMT systems than existing research suggests and propose efficient and effective scheduling policies. Our key insight is that synchronization operations actually encode programmers' intention on how inter-thread communication should be done and can be used as hints while scheduling synchronization operations. Based on this insight, we have built QiThread, a synchronization-determinism system with semantics-aware scheduling policies. Results of a diverse set of 108 programs show that QiThread is able to achieve comparable low overhead as state-of-the-art synchronization-determinism systems without the limitations associated with them.

CCS Concepts • **Software and its engineering** → **Scheduling**; **Synchronization**; **Multithreading**; *Software performance*; Software reliability; • **Theory of computation** → *Program semantics*; • **Computing methodologies** → Parallel computing methodologies.

Keywords synchronization determinism, synchronization scheduling, semantics-aware policies, deterministic multithreading, stable multithreading

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

PPoPP '19, February 16–20, 2019, Washington, DC, USA

© 2019 Association for Computing Machinery.

ACM ISBN 978-1-4503-6225-2/19/02…$15.00
https://doi.org/10.1145/3293883.3295731

1 Introduction

Multithreaded programs are now a necessity to fully utilize the computing power of multi-core processors. Unlike sequential ones, multithreaded programs execute nondeterministically, as they can follow different schedules and produce different output even if they are given the same exact input. Nondeterminism makes debugging difficult, as developers may have to reproduce the exact buggy schedule from a vast number of possible ones. Nondeterminism also complicates testing and deployment, as it is difficult, if not impossible, for in-house testing to cover all schedules that are feasible after in-field deployment.

To address the nondeterminism issue of multithreaded programs, there has been a stream of research into *deterministic multithreading (DMT)* systems. By enforcing the same schedule for the same input, DMT systems eliminate nondeterminism from multithreaded programs. These DMT systems greatly simplify debugging, testing, deployment, replication, and record-replay of multithreaded programs [18, 21, 25].

A common task for all DMT systems is to enforce *synchronization determinism* under which the order of synchronization operations is deterministic. During the process of adding synchronization operations, programmers usually strive to exclude only buggy schedules and retain as many non-buggy schedules as possible to maximize runtime performance. DMT systems view synchronization from a different angle, and they essentially consider synchronization as a form of nondeterministic inter-thread communication that programmers intend to carry out. Due to the nondeterministic nature of synchronization, DMT systems all deterministically schedule synchronization operations.

All existing synchronization-determinism techniques can be considered as a combination of the same turn-based *mechanism* and one of two scheduling *policies*: round-robin or logical-clock-based [49]. The *turn*-based mechanism ensures that (1) at any given time only one thread can have the turn and (2) a thread can execute a synchronization operation only if the thread has the turn. The scheduling policies decide (1) when a thread can get the turn and (2) which thread should get the next turn.

While the mechanism is standard and itself has little-to-none overhead [54], the policies are far from perfect, and they each have their own limitations. The round-robin policy allows threads to execute synchronization operations in a

round-robin fashion, and it works well only if threads perform synchronization operations at the same rate. To achieve a good performance in practice, Parrot [26], the state-of-the-art synchronization-determinism system, has shown that some program annotations are necessary. The logical-clock-based policy counts the numbers of instructions different threads have executed as logical clocks, and it obtains a deterministic total order by allowing the thread with the global minimum logical clock to execute a synchronization operation. While this policy achieves good performance without program annotations, the schedules generated by this policy for similar inputs may differ significantly, i.e., not *stable*. It is also not trivial to come up with a well-balanced logical clock that can accurately reflect the physical time, and large slowdowns can still be observed.

Despite the fact that synchronization determinism is a common component for all DMT systems and the limitations mentioned above, synchronization determinism has not been the focus of existing DMT systems research. This is because systems enforcing synchronization determinism alone [26, 49] only guarantee determinism for race-free programs, and such systems are commonly referred to as *weak determinism* systems. To guarantee determinism even for programs with data races, some other DMT systems [17, 27–30, 34, 42, 43, 45, 46] further enforce *memory-access determinism* that the order of shared memory accesses is also deterministic, and they are commonly referred to as *strong determinism* systems. Currently, there are more strong determinism systems than weak determinism systems. As enforcing memory-access determinism usually introduces a large performance overhead, much research effort has been devoted to improving the efficiency by exploring different deterministic multi-core architectures [29, 30, 34] and memory consistency models [17, 29, 30, 43, 45, 46].

However, we argue that the importance of synchronization determinism is overlooked, and results from several recent works suggest that synchronization determinism is just as, if not more, important as memory-access determinism.

- After synchronization determinism is enforced, the program interleaving space is more constrained, and this can significantly reduce the number of remaining races. For instance, Peregrine [28] reported at most 10 races in millions of shared memory accesses within an execution after synchronization determinism is enforced. However, the overhead of enforcing memory-access determinism remains the same regardless of the number of races.
- The state-of-the-art synchronization-determinism system, Parrot [26], has shown that it constraints the interleaving space to the extent that a model checker, dBug [56], can thoroughly check the remaining schedules after synchronization determinism is enforced.
- Even if strong determinism is really desired, recent work [43, 45, 46] has demonstrated that enforcing memory-access determinism does not require extra program serialization on top of the serialization caused by synchronization determinism. As a result, one can consider synchronization determinism and memory-access determinism as two orthogonal tasks, and advancements in synchronization determinism can directly benefit strong determinism systems.

In this paper, we propose scheduling policies that are efficient, stable, and require no performance annotations. Our key insight is that synchronization operations actually encode programmers' intention on how inter-thread communication should be done and can be used as hints to better align threads while scheduling. For example, considering a program in which one thread wakes up multiple other threads in a loop and the threads to be waken up share a similar code structure, scheduling all the wake-up operations as a whole can help balance synchronization across multiple threads.

Based on this insight, we have built QiThread[1], a runtime synchronization-determinism system with semantics-aware scheduling policies. We use the round-robin policy as our base policy, and we then apply semantics-aware scheduling policies on top of it to help mitigate the limitations of the round-robin policy without performance annotations. Results of a diverse set of 108 programs show that QiThread is able to achieve comparable low overhead as state-of-the-art DMT systems without the limitations associated with them.

The primary contributions of this paper are:

- To the best of our knowledge, we are the first to (1) elaborate the importance of synchronization determinism and (2) propose new scheduling policies that leverage the semantics of synchronization operations to address the limitations of existing policies.
- We prototype our approach in QiThread, consisting of a runtime library as a drop-in replacement library for pthreads and a user-space scheduler.
- We evaluate our prototype on a diverse set of 108 real-world programs, and our results show that QiThread is able to achieve performance and scalability comparable to those of state-of-the-art synchronization-determinism systems without their limitations.

2 Limitations of Existing Policies

There are different sources that can lead to nondeterminism during the execution of multithreaded programs, and they can be attributed to internal factors and external factors. Internal factors are inherited from nondeterministic inter-thread communication, which manifests as (1) nondeterministic orders of synchronization operations and (2) races on

[1]Qi in QiThread is the Hanyu Pinyin, which is one type of romanization, of the Chinese character "齐" with a meaning of aligned and balanced.

```c
1  int producer(int argc, char *argv[]){
2    ....
3    for (i = 0; i<nthreads; ++i)
4      pthread_create(..., consumer, ...);
5    ...
6    for (i = 0; i<nblocks;++i){
7      char *block = read_block(i);
8      pthread_mutex_lock(&m);
9      enqueue(q, block);
10     pthread_mutex_unlock(&m);
11     pthread_cond_signal(&cv);
12   }
13   ...
14 }
15
16 void *consumer(void *arg) {
17   ...
18   while(1) {
19     pthread_mutex_lock(&m);
20     while (empty(q))
21       pthread_cond_wait(&cv, &m);
22     char *block = dequeue(q);
23     pthread_mutex_unlock(&m);
24     ...
25     compress(block);
26   }
27   ...
28 }
```

(a) Simplified pbzip2 code

Turn	producer	consumer 1	consumer 2
1	L4: `create()`		
2		L16: `thread_begin()`	
3	L4: `create()`		
4		L19: `lock(&m)`	
5			L16: `thread_begin()`
6	L8: `lock(&m) blocks`		
7		L21: `wait(&cv, &m) blocks`	
8			L19: `lock(&m)`
9			L21: `wait(&cv, &m) blocks`
10	L8: `lock(&m) returns`		
11	L10: `unlock(&m)`		
12	L11: `signal(&cv)`		
13		L21: `wait(&cv, &m) returns`	
14	L8: `lock(&m) blocks`		
15		L23: `unlock(&m)`	
16	L8: `lock(&m) returns`	L25: `compress()`	
17		L19: `lock(&m) blocks`	
18	L10: `unlock(&m)`		
19		L19: `lock(&m) returns`	
20	L11: `signal(&cv)`		
21		L23: `unlock(&m)`	
22		L25: `compress()`	L21: `wait(&cv, &m) returns`
23	L8: `lock(&m) blocks`		
24		L19: `lock(&m) blocks`	
25			L21: `wait(&cv, &m) blocks`

(b) The first 25 synchronization operations and the `compress` functions

Figure 1. Simplified pbzip2 program and the synchronization operations scheduled in the first 25 turns

shared program variables [32, 47] and shared library resources [17, 49]. External factors include (1) program inputs concerning both data and timing and (2) nondeterminism in the execution environment, which can be in compilers, libraries, operating systems, and hardware. Depending on what sources a DMT system eliminates, different DMT systems have different focuses and employ different determinism-enforcement strategies.

While there are DMT systems focusing on external factors, we focus on DMT systems that eliminate internal nondeterminism. As argued in Section 1, synchronization determinism is important for all DMT systems eliminating internal nondeterminism. Next, we use an example to illustrate the limitations of existing scheduling policies for synchronization determinism.

An Example. Figure 1a shows a code snippet simplified from the pbzip2 application, which is a parallel compression/decompression utility program. It uses the common producer-consumer paradigm, where the `main` function creates multiple consumer threads using the `consumer` function as the start routine and then acts as the producer itself. The producer reads blocks of data from the input file, and multiple consumers compress them in parallel. After the producer thread reads a data block, it calls `pthread_mutex_lock` on a shared mutex, enqueues the data block, releases the shared mutex, and finally wakes up a consumer thread by calling `pthread_cond_signal`. The consumer thread being woken up first calls `pthread_mutex_lock` and then checks if the data-block queue has data to work on. If not, it would call `pthread_cond_wait` to wait for the data to be ready. Otherwise, the consumer threads dequeue one data block and work on it. Usually, the `compress` function call in the consumer threads takes much longer than the call to `read_block(i)` in the producer thread.

The Limitations of the Round-Robin Policy. All DMT systems currently use the same turn-based mechanism to enforce synchronization determinism, and they either use round-robin or logical clocks to pass the turn around threads. With the round-robin policy, all threads take turns to execute synchronization operations. The waiting time depends on the structure of program synchronization. When the numbers and frequency of synchronization operations are balanced across all threads, the waiting time can be minimized and program execution can achieve a good performance. Otherwise, the waiting time can be long, and the program execution can be serialized in the worst case.

Figure 1b shows the resulting schedule with the round-robin policy if we let the producer thread create two consumer threads with the simplified code shown in Figure 1a. Each row shows the synchronization operation executed by the running thread that has the turn, and each cell contains the line number prefixed with 'L' as in Figure 1a and the

abbreviated synchronization name. If a synchronization operation gets blocked, it will return only if it gets the turn again after being woken up. The function `thread_begin` is not called explicitly by the program, but it is usually added by DMT systems to ensure deterministic initialization of data structures related to child threads. For clarity, Figure 1b does not show synchronization operations during the initialization of the producer thread, and it just shows the first 25 turns. From the schedule, we can see the blocks are processed in a serialized way by the same thread.

Parrot [26] is the state-of-the-art system implementing round-robin scheduling for synchronization determinism, and it provides a *soft-barrier* interface to restore parallelism. The software barrier performs similarly as a barrier, and it encourages the scheduler to co-schedule a group of threads at program points where it is added. For the code shown in Figure 1a, if one adds a software barrier before calling the `compress` function in line 25, the serialization problem will be solved.

However, these performance hints could require significant programmer effort, as the ideal placement for such hints may not be easy to determine. Moreover, after a developer inserts soft barriers correctly, they become a part of program code that needs to be maintained and tested each time the application is modified, which creates another burden for the developers. Pegasus [31] can automate the process of inserting soft barriers by profiling program execution and analyzing execution logs. However, it requires a repeated trial-and-error process to finalize the best position to insert soft barriers.

The Limitations of the Logical-Clock-Based Policy. The other scheduling policy based on logical clocks is more resilient to the synchronization imbalance problem by design, as it allows a thread to execute synchronization operations only if the thread has the lowest instruction count. However, this policy suffers from the instability problem that minor input or code changes can perturb instruction counts and subsequently the schedules. It has been reported that pbzip2 running on top of CoreDet [17], a DMT system using the logical-clock-based scheduling policy, uses five different schedules to process eight different files [26]. As a result, testing one input provides little insight into how the program and system work on other inputs. Further, this policy can still have the imbalance problem as instruction counts may not perfectly reflect the physical time, and large overhead can also be observed [26].

3 Scheduling Policies and Illustrations

We strive to develop scheduling policies for synchronization determinism that can achieve good performance and schedule stability without programmer intervention.

We use round robin as our base policy due to its stability. To address the performance slowdown of the round-robin policy, we design semantics-aware scheduling policies that let the scheduler break the vanilla round-robin policy when it recognizes certain synchronization operations, and the goal is to achieve better synchronization alignment among threads with our policies.

We start our design with programs that show significant performance improvement after applying Parrot soft barriers on top of the vanilla round-robin policy. By comparing schedules before and after applying Parrot soft barriers, we come up with patterns of imbalanced schedules and design semantics-aware policies to compensate these imbalances. The design process stops when we are able to achieve comparable performance as Parrot with soft barriers on most programs in our evaluation. We end up with five policies, where each policy is designed by analyzing a very small number of programs, which is one in some cases. Our evaluation will show that these five policies designed based on a small number of programs can benefit many other programs. Below, we describe our policies and elaborate the rationales behind them.

3.1 BoostBlocked: Prioritizing Blocked Threads

Like typical scheduling policies, we maintain queues for threads with different states. QiThread maintains three queues: one *run queue* for running threads, one *wait queue* for waiting threads that are blocked while executing some synchronization operations, and one *wake-up queue* for threads that were in the wait queue and just being woken up.

This BoostBlocked policy prioritizes threads that were previously blocked, by putting threads just being woken up into the high priority wake-up queue and scheduling threads in the wake-up queue before those in the run queue.

This is based on two rationales: (1) when a resource becomes available as being indicated by the unblocking operation, we want to have the resource utilized as soon as possible, and (2) other threads that share a similar role as the woken-up thread may have executed more synchronization operations, prioritizing these previously blocked threads can help re-balance synchronization in all threads.

3.2 CreateAll: `pthread_create` Loops Create All

Multithreaded programs usually create multiple child threads inside a loop, and the `pthread_create` call is a synchronization operation that needs to be deterministically ordered. With a round-robin policy, the thread executing the `pthread_create` loop needs to wait for each running thread to execute one synchronization operation before it can execute the next `pthread_create` call. Depending on what other threads are doing, there is a high chance that the next call to `pthread_create` will be delayed. This will also make the synchronization operations in child threads be unbalanced, as thread created one iteration earlier has executed one more synchronization operation.

Parent Thread *Child Thread*

```
void parent_main(...) {           void child_main(...) {
  for (...)                         ... // all computation
    pthread_create(...,             ... // no synchronization
        child_main, ...);         }

  child_main();
  ...
}
```

Figure 2. A `pthread_create` loop creates all threads

Figure 2 shows a simplified real-world example, where the child threads do not explicitly execute any synchronization operations, and all the child threads are serialized after applying the round-robin policy.

This CreateAll policy allows a thread to finish the whole `pthread_create` loop if the loop does not contain other synchronization operations. This does not delay these child threads much, as the time taken to execute a `pthread_create` loop is short. Further, this better balances synchronization operations among all child threads, and this can balance child threads and the parent thread, if the parent thread also invokes the same function as the child threads.

3.3 CSWhole: Critical Sections Scheduled as a Whole

Critical sections are formed by first acquiring a mutex and then releasing the mutex. For a mutex with multiple threads contending for it, a round-robin policy will deterministically schedule one thread to acquire the mutex, go through all other threads contending for the mutex and block these threads, and schedule the thread that acquired the mutex to release the mutex. After the first thread finishes executing a critical section, the scheduler deterministically wakes up the remaining threads in a chained fashion to acquire and release the mutex. During this process, if there are many threads contending for the mutex, the overhead of putting threads into the wait queue and then putting them into the wake-up queue can be significant, and it can greatly slow down the program if the critical sections themselves are short.

This CSWhole policy schedules a critical section as a whole. This policy can greatly improve performance if there are many threads competing for the same mutex and the lengths of critical sections are short. When the critical sections become longer, the benefit of the CSWhole policy diminishes, but no extra overhead will be introduced as other threads cannot proceed anyway because of the mutex.

However, if threads are acquiring different mutexes, the CSWhole policy can unnecessarily serialize critical sections and may lead to poor performance. As we will see later, programs used in our evaluation do not run into this situation. Two characteristics of our programs make the negative impact of CSWhole small: (1) critical sections are often short,

The Wait Thread *Multiple Post Threads*

```
sem_wait(s);          pthread_mutex_lock(&m);
                      n--;
                      if (n == 0) // The last thread posts
                        sem_post(s);
                      pthread_mutex_unlock(&m);
```

Figure 3. Branched unblocking

and (2) there is some computation between critical sections. As a result, each thread can finish executing the critical section while other threads are doing computation, under which case CSWhole does not introduce much delay.

3.4 WakeAMAP: Wake Up as Many as Possible

To enforce order relationships among threads, programmers usually let some threads wait on condition variables or semaphores and let some other threads wake up these blocked threads. When there are multiple threads to be woken up, the thread waking up others and the blocked threads usually have very different code structures and workloads. If we schedule these threads in a round-robin fashion, the unblocking side may need to wait for those threads just woken up to execute synchronization operations.

Figure 1a is a real-world example where the program gets serialized because of the way how the unblocking side, which is the producer, gets scheduled with the round-robin policy. Note in this case, even if we apply the CreateAll policy, the program execution will be similarly serialized as in Figure 1b.

This WakeAMAP policy will let a thread executing an unblocking operation continue its execution until there are no more threads to be woken up on the same condition variable or semaphore or the unblocking thread itself gets blocked. The WakeAMAP policy can help resolve the serialization problem shown in Figure 1. This policy reduces the possibility of synchronization imbalance, as these woken-up threads usually have a similar code structure. Although the threads that get woken up need to wait for the unblocking thread to finish the unblocking loop to do any synchronization, this may not be a problem as the unblocking side usually does not do time-consuming computation before being descheduled. Further, our BoostBlocked policy can help compensate the waiting time by prioritizing threads just being woken up.

3.5 BranchedWake: Branched Unblocking

It is common to have synchronization operations in branches, and this is one source of synchronization imbalance, as not all threads are going to execute these synchronization operations. Figure 3 shows a simplified real-world example, where the wait thread is blocked on `sem_wait` and the last thread decrementing `n` wakes up the wait thread. With round-robin scheduling, the last thread sees `n` being 0 cannot immediately execute `sem_post`, as it needs to wait for other threads to

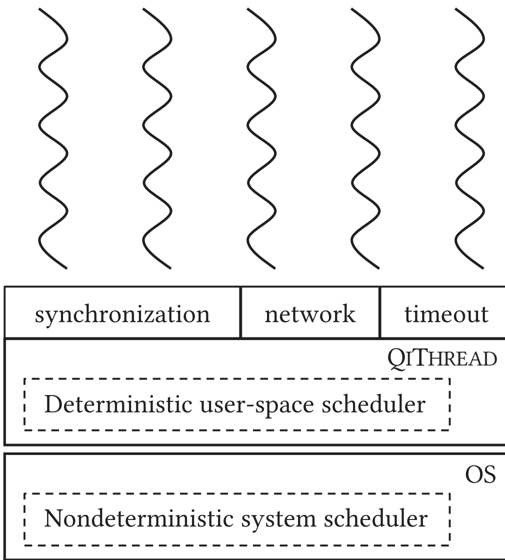

Figure 4. The components of Parrot system architecture leveraged by QiThread

execute a synchronization operation. This delays the wait thread unnecessarily.

This BranchedWake policy instruments the program with a dummy synchronization operation on branches where an unblocking operation is skipped. With the dummy synchronization operation, these multiple potential post threads become re-aligned.

In this particular example, the BoostBlocked policy also helps further balance threads, as it allows the wait thread to quickly return from the `sem_wait`. If the wait thread and these post threads are performing the same task later, this helps balance all these threads.

4 Implementation

We implement our synchronization-aware scheduling policies in QiThread, and the implementation is based on Parrot. In this section, we first present some necessary background of Parrot, focusing on its architecture and interface that are leveraged by QiThread, and then we describe how we implement QiThread on top of the Parrot infrastructure. Although we implement our scheduling policies in a software DMT system, the policies can also be applied to hardware DMT systems.

4.1 Parrot Architecture and Interface

Figure 4 shows the components of Parrot system architecture leveraged by QiThread, including a deterministic user-space scheduler and a set of wrapper functions for intercepting pthreads, network, and timeout operations. The user-space scheduler only schedules synchronization operations and delegates everything else to the OS scheduler. The wrapper functions interpose function calls to dynamically loaded libraries via `LD_PRELOAD`, "trap" the calls into the user-space scheduler where different scheduling policies can be implemented, and then delegate the actual implementation to pthreads or the OS. These design decisions greatly simplify the implementation and deployment of Parrot and QiThread. Parrot also provides an implementation of performance hints that QiThread gets rid of.

Table 1. Primitives for manipulating scheduler queues

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

The Parrot user-space scheduler employs the round-robin policy and maintains two queues: one run queue for running threads and one wait queue for waiting threads. To achieve round-robin scheduling, Parrot allows a thread to execute a synchronization operation only if the thread is the head of the run queue. After successfully executing the synchronization operation, the queues are updated accordingly.

To manipulate the queues and schedule synchronization operations, the scheduler provides an interface, and the primitives used by QiThread are shown in Table 1. The `get_turn` function waits until the calling thread gets the "turn" to execute synchronization operations. The `put_turn` function gives up the turn after the calling thread has executed some synchronization operations. The remaining three functions, `wait`, `signal`, and `broadcast`, all require the calling thread to have the turn. The `wait` function is similar to `pthread_cond_timedwait`, and it blocks the calling thread and moves it to the tail of the wait queue. The thread is moved out of the wait queue when (1) another thread wakes it up via `signal` or `broadcast` on the same `addr` or (2) the `timeout` specified in the `wait` call has expired. The return value of `wait` indicates how the thread was woken up. The `signal(void *addr)` function wakes up the first thread waiting on `addr`. The `broadcast(void *addr)` function wakes up all threads waiting on `addr` in order. Since QiThread and Parrot have different scheduler internals, the implementations for these primitives are not exactly the same while the interface remains unchanged.

Note that the `timeout` in the `wait` function is a relative *logical time* that counts the number of turns executed since the beginning of current execution, and Parrot makes timeouts deterministic by proportionally converting them to a logical timeout. A `wait(NULL, timeout)` call is a logical sleep, and a `wait(addr, 0)` call never times out. QiThread

```
int lock_wrapper(pthread_mutex_t *m) {
  scheduler.get_turn();
  while(pthread_mutex_trylock(m))
    scheduler.wait(m, 0);
  if(CSWhole_policy_not_on())
    scheduler.put_turn();
  return 0;
}
int unlock_wrapper(pthread_mutex_t *m) {
  if(CSWhole_policy_not_on())
    scheduler.get_turn();
  pthread_mutex_unlock(m);
  scheduler.signal(m);
  scheduler.put_turn();
  return 0;
}
```

Figure 5. Wrappers for mutex lock and unlock. The code for handling nested critical sections is not included. Error handling code is omitted.

does not change how these timeout operations, as well as network operations, are implemented in Parrot. We omit the details here, and please refer to the Parrot paper [26] for detailed description of their implementation.

4.2 QiThread Implementation

QiThread follows the architecture of Parrot, and it has 38 synchronization wrappers in total. It handles thread creation, start, exit, and join. It also handles all synchronization operations on mutexes, read-write locks, condition variables, semaphores, and barriers. All wrappers execute a synchronization operation only if the calling thread has the turn. Since at most one wrapper can have the turn at any time, it ensures a total order of all synchronization operations.

To implement our proposed semantics-aware scheduling policies in QiThread, the BoostBlocked policy requires changes to the queue-manipulation primitives, the CSWhole and WakeAMAP policies are implemented in synchronization wrappers, and the CreateAll and BranchedWake policies require some static analysis.

4.2.1 BoostBlocked Implementation

To implement the priority queue for threads just being woken up, QiThread maintains three queues, a run queue, a wait queue, and a wake-up queue. With these three queues, the primitives in Table 1 are implemented as follows. The `get_turn` function lets the calling thread get the "turn" and execute synchronization operations if the calling thread is (1) the head of the wake-up queue or (2) the head of the run queue when the wake-up queue is empty. The `put_turn` function puts the calling thread to the tail of the run queue, regardless of whether it was previously in the wake-up queue or the run queue. The `wait` function remains the same as described in Section 4.1. The `signal(void *addr)` function appends the first thread waiting for `addr` to the wake-up queue. The `broadcast(void *addr)` function appends all

```
int signal_wrapper(pthread_cond_t *cv) {
  scheduler.get_turn();
  scheduler.signal(cv);
  if (WakeAMAP_policy_is_on ()) {
    if (cv_wait_map[cv] > 0)
      cv_wait_map[cv]--;
    if (cv_wait_map[cv] == 0)
      scheduler.put_turn();
  }
  else
    scheduler.put_turn();
}

int wait_wrapper(pthread_cond_t *cv,
                 pthread_mutex_t *m) {
  scheduler.get_turn();
  if (WakeAMAP_policy_is_on ())
    cv_wait_map[cv]++;
  pthread_mutex_unlock(m);
  scheduler.signal(m);
  scheduler.wait(cv, 0);
  while(pthread_mutex_trylock(m))
    scheduler.wait(m, 0);
  scheduler.put_turn();
  return 0;
}
```

Figure 6. Wrappers for condition variable signal and wait. Error handling code is omitted.

threads waiting for `addr` to the wake-up queue in the same order as they were in the wait queue.

4.2.2 CSWhole and WakeAMAP Implementation

Figure 5 shows the simplified pseudo code of the mutex lock and unlock wrappers. To avoid deadlock, the `lock_wrapper` function uses the `pthread_mutex_trylock` operation, as if the calling thread is blocked waiting for a lock before giving up the turn, no other thread can get the turn. To enable the CSWhole policy, the `lock_wrapper` function does not give up the turn before returning and the `unlock_wrapper` function does not need to get the turn.

Figure 6 shows the pseudo code of the condition variable signal and wait wrappers. Once the WakeAMAP policy is turned on, the wrappers count the number of threads waiting on each condition variable with the map structure `cv_wait_map`, `signal_wrapper` decrements the counter and gives up the turn until no thread is blocked on the condition variable, and `wait_wrapper` increments the counter. To ensure the correctness of the `wait_wrapper`, i.e., (1) there is no deadlock and (2) the release of the mutex and the wait on the condition variable are atomic, it is implemented with the scheduler's `wait` which atomically gives up the turn and blocks the calling thread on the wait queue.

4.2.3 CreateAll and BranchedWake Implementation

To implement the CreateAll and BranchedWake policies, we need to instrument the programs. Figure 7a shows an example of instrumentation for a `pthread_create` loop. A new

```
for (i=0; i<nTheads; i++) {
  if (i+1<nThreads)
    keep_turn();
  pthread_create(...);
}
```

(a) CreateAll instrumentation

```
if (n == 0)
  sem_post(s);
else
  dummy_synchronization();
```

(b) BranchedWake instrumentation

Figure 7. Instrumentations for CreateAll and BranchedWake

primitive, keep_turn, is used to inform the scheduler not to give up the turn after executing the next synchronization operation from the calling thread, and it is involved only if the thread is going to create more child threads. Figure 7b shows an example of instrumentation for dummy synchronization. A call to dummy_sync is added on the branch where an unblocking synchronization operation is skipped, and this function simply calls get_turn and then put_turn.

Since there are only a small number of places where pthread_create and unblocking operations on condition variables and semaphores are invoked, we currently add such instrumentation manually, and a static analysis can be developed if manual instrumentation turns out to be infeasible in some programs.

5 Evaluation

We evaluated QiThread with all 108 programs used to evaluate Parrot. This set of programs covers a good range of parallel programming models and idioms, and it has 10× more programs than any other DMT system evaluation. Specifically, these programs are: 14 benchmarks in SPLASH-2x [3]; nine benchmarks in NPB [10]; 15 benchmarks in PARSEC [4]; 14 benchmarks in Phoenix [52], where each algorithm has two implementations, one using pthreads and the other using a map-reduce library built on top of pthreads; PBZip2, a parallel compression/decompression utility [5]; aget, a parallel file download utility [1]; pfscan, a parallel file scanner [2]; Berkeley DB, a database library [8]; OpenLDAP, a lightweight directory access protocol server [11]; MPlayer, a media encoder, decoder, and player [6]; Redis, a key-value data store server [12]; 14 image processing utilities using OpenMP in the ImageMagick software suite [9]; 33 parallel C++ STL algorithm implementations [7] that also use OpenMP. The use of complete software or benchmark suites avoids potential biases in the evaluation.

All our experiments were conducted on a machine with dual Intel Xeon E5-2670 CPUs. Each CPU has eight cores and 16 threads, running at 2.60 GHz. The machine has a total of 32 logical CPUs and 128GB memory, and it is running Linux kernel version 4.40.

We mimic many settings in Parrot evaluation. For programs with both a client side and a server side, we run both endpoints on the same machine. For the five programs that use ad hoc synchronization [58], a call of sched_yield is added to the busy-wait synchronization loops. We also add a call of set_base_time to programs that use timed pthreads operations. All programs are compiled with -O2. We use the GNU libgomp to support OpenMP programs. While Parrot used libgomp included in GCC 4.5.4, we migrate the setup to the more recent GCC 5.4.0.

Our evaluation focuses on three questions: (1) How does QiThread compare with Parrot on performance? (2) How helpful are different policies, and what policies should users enable? (3) How does the performance of QiThread vary with different thread counts?

5.1 Performance

To compare the performance of QiThread and Parrot, we measure the program execution times under QiThread and Parrot and normalize them to the times of nondeterministic execution. We mirrored the setup from how Parrot evaluated its performance with these programs to get a fair comparison between QiThread and Parrot. For most programs, we used the same workloads and number of threads as described in the Parrot paper to compare QiThread and Parrot. One exception is ImageMagick, where Parrot used a 16k resolution image as an input file to measure their performance results. We were unable to find that exact same file. We used an 8k resolution image described in their scalability evaluation and also compared QiThread results with Parrot's using the 8k image. We omit the detailed workload description for other programs here.

For Parrot performance, we include both the numbers reported in the original paper and the numbers measured by us. For all programs that Parrot applies soft barrier hints originally, we apply the same hints in the same places in our own measurement. Some Parrot numbers reported in the paper are measured with the *performance critical section (PCS)* hints applied to the programs, which allow code in these performance critical sections to execute nondeterministically, and we measure the Parrot performance both with and without PCS hints for most of these programs. The exceptions are programs that use the libgomp library, as one PCS hint is applied to the library but the relevant synchronization code enclosed in the PCS hint only exists in the version from GCC 4.5.4 but not in the version from GCC 5.4.0 that we use, and we only measure the Parrot performance without PCS hints for these programs.

For QiThread performance, we turn on the policies described in Section 3 in the order of BoostBlocked, CreateAll, CSWhole, WakeAMAP, and BranchedWake, and measure the performance under different policy configurations. We do not try all different possible policy combinations due to the large size of search space given the program number. While comparing performance under different configurations, we are particularly interested in coming up with a default configuration applicable to most programs.

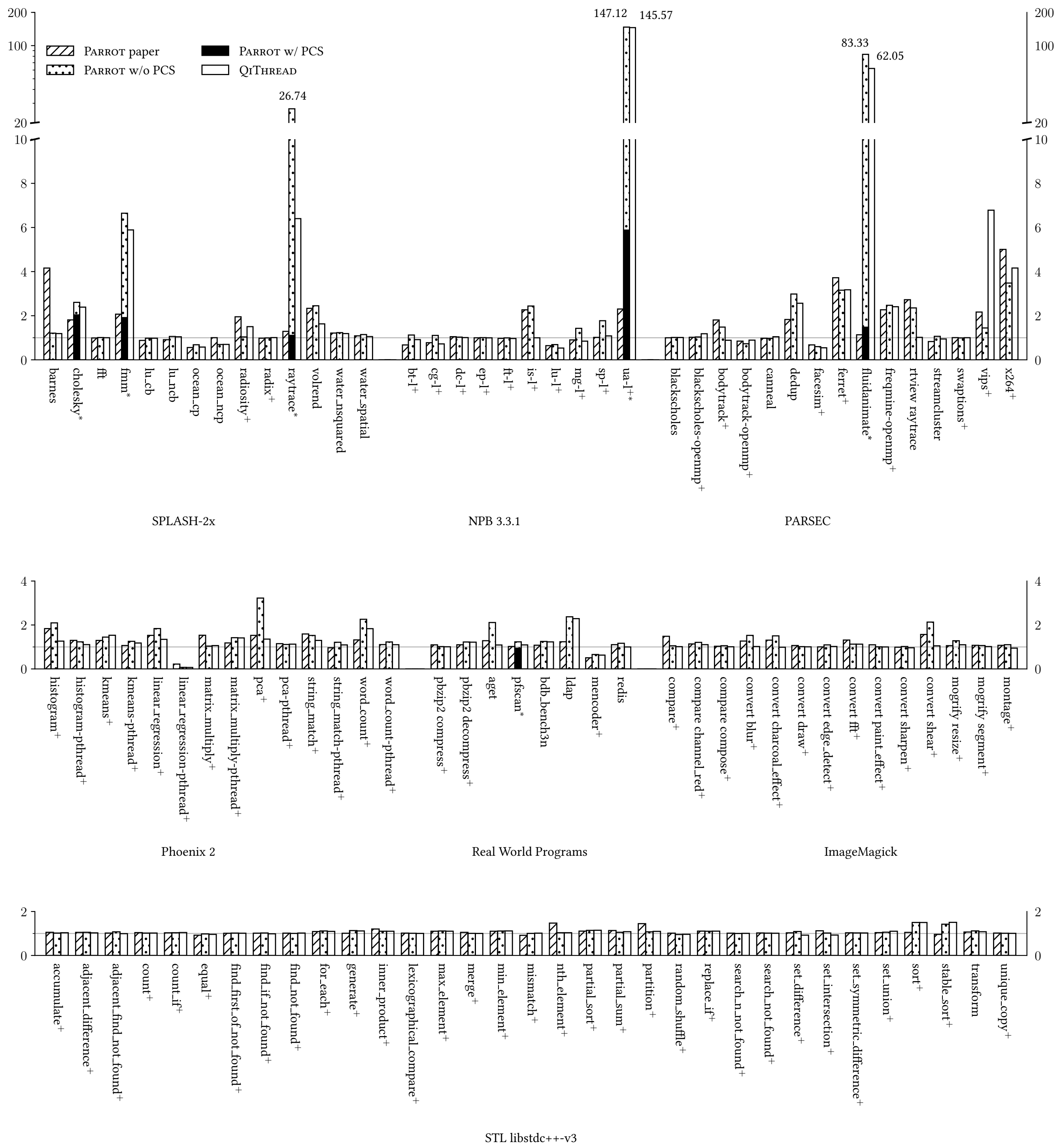

Figure 8. Execution times under QiThread and Parrot with nondeterministic execution times normalized to 1. "Parrot paper" bars are results from the Parrot paper. "Parrot w/ PCS" bars are our best effort to reproduce Parrot results. "Parrot w/o PCS" bars are our measured performance with only soft barrier hints inserted but not PCS hints to ensure determinism. "QiThread" bars are the results with all policies enabled. Programs with applicable soft barrier hints are marked with '+', and programs with applicable PCS hints are marked with '*'.

As we will detail in Section 5.2, turning on a policy rarely hurts program performance in our evaluation, and we use enabling all policies as the default policy configuration for QiThread due to its simplicity. The performance numbers reported in this subsection are measured with this default all-policy-enabled configuration.

Figure 8 compares the performance of QiThread with Parrot. The height of each bar is normalized to the non-deterministic execution time. For each program, we report the Parrot performance presented in the original paper (Parrot paper), our measured Parrot performance, and QiThread performance (QiThread). Our measured Parrot performance numbers have two versions when PCS hints are applicable, i.e., one with PCS hints applied (Parrot w/ PCS) and one without (Parrot w/o PCS), and they are all measured with applicable soft barrier hints applied. Again, QiThread performance numbers are measured under the default configuration with all policies enabled.

We first compare the Parrot paper performance numbers and our measured Parrot numbers with PCS hints. For 93 programs, we were able to reproduce performance results within 130% of the numbers reported by the Parrot paper, 72 of which are within 110%. This represents our best effort to resolve the performance difference, including contacting the Parrot authors.

We then compare QiThread numbers with our measured Parrot numbers. As QiThread always guarantees synchronization determinism, we compare QiThread performance with Parrot performance without PCS. QiThread is able to achieve a comparable performance, i.e., 110% or less, as our measured Parrot performance without PCS hints for 103 programs. For 30 programs, the QiThread execution times are smaller than 90% of the corresponding Parrot execution times, which can be considered as non-negligible speedups. QiThread takes more than 110% of the no-PCS Parrot numbers on five programs, which are radiosity from SPLASH-2x and blackscholes-openmp, bodytrack-openmp, vips, and x264 from PARSEC. Among these five programs, QiThread benefits four of them but not vips, and vips has the largest slowdown comparing QiThread with Parrot.

There are five programs that QiThread has an overhead of more than 400%, i.e., fmm, raytrace, ua, fluidanimate, and vips. Parrot applied PCS hints to the first four. Although QiThread has better performance than Parrot without PCS hints, QiThread policies are not enough to further improve the performance. We leave it for future work to develop policies that can achieve comparable performance as Parrot with PCS hints.

5.2 Effectiveness of Different Policies

We next report how effective different policies are on improving the performance. As discussed in Section 5.1, we did not test all combinations of different policies. Rather, we applied the policies in the order of BoostBlocked, CreateAll, CSWhole, WakeAMAP, and BranchedWake, due to the large search space for 108 programs. We consider a policy as beneficial if it reduces the execution time to be less than 90% of the time under the previous policy configuration. For the first policy we apply, we compare its performance with vanilla round-robin scheduling. To understand how well the default all-policy-enabled configuration works, we will also report cases that achieve better performance with some policies turned off compared with the default configuration.

Among all 108 programs, the BoostBlocked policy benefits 43 programs. After applying CreateAll, 39 programs get performance improvements. The CSWhole policy benefits 49 programs, 18 of which have already gained some speedup with BoostBlocked or CreateAll. Although WakeAMAP only benefits five programs, two of them, i.e., pbzip2 and ferret in the PARSEC suite, gain very large performance improvements. Specifically, the two pbzip2 tasks, decompression and compression, get speedups of 300% and almost 1000%, respectively, and ferret gets a speedup of more than 150%. At last, BranchedWake is able to benefit 20 programs, all of which utilize the OpenMP library.

Since our scheduling policies are all heuristic-based, applying a policy can sometimes hurt the performance. We observed three instances of such cases that resulted in an increased execution time of more than 10%: the CreateAll policy increased the overhead of aget from −3.11% to 14.52% and the overhead of STL partial_sort from −1.9% to 16.38%, and the WakeAMAP policy increased the overhead of convert "paint_effect" case in ImageMagick from −7.24% to 3.39%.

No QiThread policy is able to improve performance, i.e., by more than 10%, for vips from PARSEC. The vips program maintains an idle queue to hold threads that have finished their workload and dispatch workload with the producer-consumer idiom. The WakeAMAP policy is supposed to help align all consumer threads, just like how it helps the example in Figure 1a. However, each consumer has a unique condition variable that the producer uses to wake up the consumer. As a result, although the loop-contained wake up operation in the producer can wake up many consumers, the wrappers in Figure 6 cannot keep track of the number of consumers to wake up. We envision that the WakeAMAP policy can be further amplified with more sophisticated static analysis and instrumentation to benefit the vips program, and we leave such an extension for future work.

We next compare the best performance we can get under different policy configurations with the default all-policy-enabled configuration. The former is able to make one more program to achieve similar performance, i.e., +10% or less, as Parrot w/o PCS. This application is x264, which achieved an overhead of 162% with BoostBlocked turned off, where the overhead numbers for Parrot and the QiThread default configuration are 242% and 317%, respectively. This further shows that the all-policy-enabled configuration can work well in practice as the default.

5.3 Scalability

We next measure how the performance changes with different thread counts, and we report how the overhead varies.

We first randomly selected five programs for scalability analysis: barnes from SPLASH-2x, bodytrack from PARSEC, histogram from phoenix, convert "shear" case from ImageMagick, and pbzip2 decompress. For each program, we used four different thread counts: 4, 8, 16, and 32. All these five programs varied within 42% from each program's mean overhead across four thread counts. We did the same measurement with Parrot, and QiThread shows smaller performance variations than Parrot, where Parrot performance varies to a maximum of 47%.

We next focused on the 30 programs on which QiThread achieved non-negligible speedup compared with Parrot with soft barrier hints, and we selected 23 of them for scalability analysis, as we can increase the thread number in these programs to 32. Compared with 16 or 24 threads that we used in performance evaluation presented in Section 5.1, 17 of the 23 programs have their overhead increased when running with QiThread, and the other 6 programs enjoy a reduction in overhead. Parrot also encounters increased overhead on 17 programs, on 16 of which QiThread encounters overhead increase as well. Among the 16 programs that both QiThread and Parrot encounter increased overhead, QiThread leads to less increase on 12 of them.

Overall, we find that QiThread's performance is robust to thread-count changes. Comparing with Parrot, most of the programs we tested are less sensitive to thread-count changes under QiThread.

6 Related Work

Nondeterminism makes it a challenging task to ensure the reliability of multithreaded programs, as concurrency bugs only nondeterministically manifest under specific interleavings. Researchers have developed many different tools on concurrency bug detection [50, 59, 60], diagnosis [14, 15, 38], and fixing [24, 37, 39–41, 57]. To deterministically reproduce concurrency bugs for easier debugging, researchers have proposed record and replay tools [13, 35, 51], which log different information under nondeterministic executions and provide different determinism guarantees while replaying. Orthogonal to record and replay tools, researchers have also developed many different types of DMT systems that eliminate nondeterminism from the sources and enforce deterministic execution. DMT systems can ease debugging without recording runtime information, and they also have usages other than debugging. Below, we focus on related work about DMT systems that are closest to QiThread.

Different DMT systems achieve determinism at different system levels, including hardware architectures [29, 30, 34, 53], runtime systems [17, 20, 26, 28, 33, 42–46, 49, 55, 61], programming languages [22, 23, 48], and operating systems [16, 19, 36]. QiThread enforces synchronization determinism with a runtime system, and systems at different levels can complement each other.

Among all runtime DMT systems, Kendo [49] and Parrot [26] only enforce synchronization determinism. Since Kendo is logical-clock-based, the schedules generated by Kendo may not be stable. Both Parrot and QiThread use round robin as their base scheduling policy to ensure schedule stability. To achieve good performance, Parrot proposes performance hints, while we propose scheduling policies that are highly related to synchronization semantics.

One group of researchers has specifically analyzed the sources of nondeterminism while enforcing synchronization determinism and the cost of the two existing scheduling policies [54]. While they share the same focus on synchronization determinism as us and their results provide many insights into synchronization determinism, they do not propose any solution for performance improvement.

Most existing runtime DMT systems enforce memory-access determinism to provide strong determinism. The enforcement of memory-access determinism and synchronization determinism are two orthogonal tasks, and our scheduling policies can be incorporated by strong DMT systems.

7 Conclusion

Deterministic multithreading systems can be useful for the development and deployment of multithreaded programs. However, the importance of synchronization determinism has been overlooked, and existing scheduling policies for synchronization determinism have limitations. We have presented QiThread, a runtime system that enforces synchronization determinism with scheduling policies that leverage the semantics of synchronization operations. We evaluated QiThread on a comprehensive set of 108 real-world multithreaded programs. Our results show QiThread is able to achieve low overhead and good scalability without the limitations of existing systems.

Acknowledgments

The authors would like to thank our shepherd Vasileios Trigonakis and the anonymous reviewers for their valuable feedback and helpful suggestions. The authors would also like to thank Heming Cui for making the Parrot code available and helping us reproduce some Parrot results.

References

[1] Aget. http://www.enderunix.org/aget/.
[2] pfscan. http://freshmeat.sourceforge.net/projects/pfscan.
[3] SPLASH-2x. http://parsec.cs.princeton.edu/parsec3-doc.htm.
[4] The Princeton application repository for shared-memory computers (PARSEC). http://parsec.cs.princeton.edu/.
[5] Parallel BZIP2 (PBZIP2). https://launchpad.net/pbzip2.
[6] MPlayer. http://www.mplayerhq.hu/design7/news.html.
[7] STL Parallel Mode. http://gcc.gnu.org/onlinedocs/libstdc++/manual/parallel_mode.html.

[8] Berkeley DB. http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/overview/index.html.
[9] ImageMagick. http://www.imagemagick.org/script/index.php.
[10] NASA Parallel Benchmarks. http://www.nas.nasa.gov/publications/npb.html.
[11] OpenLDAP. http://www.openldap.org/.
[12] Redis. http://redis.io/.
[13] Gautam Altekar and Ion Stoica. 2009. ODR: Output-deterministic Replay for Multicore Debugging. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09)*. ACM, New York, NY, USA, 193–206. https://doi.org/10.1145/1629575.1629594
[14] Joy Arulraj, Po-Chun Chang, Guoliang Jin, and Shan Lu. 2013. Production-run Software Failure Diagnosis via Hardware Performance Counters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. ACM, New York, NY, USA, 101–112. https://doi.org/10.1145/2451116.2451128
[15] Joy Arulraj, Guoliang Jin, and Shan Lu. 2014. Leveraging the Short-term Memory of Hardware to Diagnose Production-run Software Failures. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. ACM, New York, NY, USA, 207–222. https://doi.org/10.1145/2541940.2541973
[16] Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. 2010. Efficient System-enforced Deterministic Parallelism. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*. USENIX Association, Berkeley, CA, USA, 193–206. http://dl.acm.org/citation.cfm?id=1924943.1924957
[17] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. 2010. CoreDet: A Compiler and Runtime System for Deterministic Multithreaded Execution. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*. ACM, New York, NY, USA, 53–64. https://doi.org/10.1145/1736020.1736029
[18] Tom Bergan, Joseph Devietti, Nicholas Hunt, and Luis Ceze. 2011. The Deterministic Execution Hammer: How Well Does it Actually Pound Nails?. In *The 2nd Workshop on Determinism and Correctness in Parallel Programming (WODET '11)*.
[19] Tom Bergan, Nicholas Hunt, Luis Ceze, and Steven D. Gribble. 2010. Deterministic Process Groups in dOS. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*. USENIX Association, Berkeley, CA, USA, 177–191. http://dl.acm.org/citation.cfm?id=1924943.1924956
[20] Emery D. Berger, Ting Yang, Tongping Liu, and Gene Novark. 2009. Grace: Safe Multithreaded Programming for C/C++. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*. ACM, New York, NY, USA, 81–96. https://doi.org/10.1145/1640089.1640096
[21] Robert L. Bocchino, Jr., Vikram S. Adve, Sarita V. Adve, and Marc Snir. 2009. Parallel Programming Must Be Deterministic by Default. In *Proceedings of the First USENIX Conference on Hot Topics in Parallelism (HotPar'09)*. USENIX Association, Berkeley, CA, USA, 4–4. http://dl.acm.org/citation.cfm?id=1855591.1855595
[22] Robert L. Bocchino, Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. 2009. A Type and Effect System for Deterministic Parallel Java. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*. ACM, New York, NY, USA, 97–116. https://doi.org/10.1145/1640089.1640097
[23] Robert L. Bocchino, Jr., Stephen Heumann, Nima Honarmand, Sarita V. Adve, Vikram S. Adve, Adam Welc, and Tatiana Shpeisman. 2011. Safe Nondeterminism in a Deterministic-by-default Parallel Language. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. ACM, New York, NY, USA, 535–548. https://doi.org/10.1145/1926385.1926447
[24] Yan Cai and Lingwei Cao. 2016. Fixing Deadlocks via Lock Pre-acquisitions. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 1109–1120. https://doi.org/10.1145/2884781.2884819
[25] Heming Cui, Rui Gu, Cheng Liu, Tianyu Chen, and Junfeng Yang. 2015. Paxos Made Transparent. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. ACM, New York, NY, USA, 105–120. https://doi.org/10.1145/2815400.2815427
[26] Heming Cui, Jiri Simsa, Yi-Hong Lin, Hao Li, Ben Blum, Xinan Xu, Junfeng Yang, Garth A. Gibson, and Randal E. Bryant. 2013. Parrot: A Practical Runtime for Deterministic, Stable, and Reliable Threads. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 388–405. https://doi.org/10.1145/2517349.2522735
[27] Heming Cui, Jingyue Wu, John Gallagher, Huayang Guo, and Junfeng Yang. 2011. Efficient Deterministic Multithreading Through Schedule Relaxation. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*. ACM, New York, NY, USA, 337–351. https://doi.org/10.1145/2043556.2043588
[28] Heming Cui, Jingyue Wu, Chia-Che Tsai, and Junfeng Yang. 2010. Stable Deterministic Multithreading Through Schedule Memoization. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*. USENIX Association, Berkeley, CA, USA, 207–221. http://dl.acm.org/citation.cfm?id=1924943.1924958
[29] Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. 2009. DMP: Deterministic Shared Memory Multiprocessing. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*. ACM, New York, NY, USA, 85–96. https://doi.org/10.1145/1508244.1508255
[30] Joseph Devietti, Jacob Nelson, Tom Bergan, Luis Ceze, and Dan Grossman. 2011. RCDC: A Relaxed Consistency Deterministic Computer. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, New York, NY, USA, 67–78. https://doi.org/10.1145/1950365.1950376
[31] Monika Dhok, Rashmi Mudduluru, and Murali Krishna Ramanathan. 2015. Pegasus: Automatic Barrier Inference for Stable Multithreaded Systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*. ACM, New York, NY, USA, 153–164. https://doi.org/10.1145/2771783.2771813
[32] Perry A. Emrath and David A. Padua. 1988. Automatic Detection of Nondeterminacy in Parallel Programs. In *Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging (PADD '88)*. ACM, New York, NY, USA, 89–99. https://doi.org/10.1145/68210.69224
[33] Gagan Gupta, Srinath Sridharan, and Gurindar S. Sohi. 2014. Globally Precise-restartable Execution of Parallel Programs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 181–192. https://doi.org/10.1145/2594291.2594306
[34] Derek R. Hower, Polina Dudnik, Mark D. Hill, and David A. Wood. 2011. Calvin: Deterministic or Not? Free Will to Choose. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA '11)*. IEEE Computer Society, Washington, DC, USA, 333–334. http://dl.acm.org/citation.cfm?id=2014698.2014870
[35] Jeff Huang, Charles Zhang, and Julian Dolby. 2013. CLAP: Recording Local Executions to Reproduce Concurrency Failures. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, 141–152. https://doi.org/10.1145/2491956.2462167
[36] Nicholas Hunt, Tom Bergan, Luis Ceze, and Steven D. Gribble. 2013. DDOS: Taming Nondeterminism in Distributed Systems. In *Proceedings*

of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13). ACM, New York, NY, USA, 499–508. https://doi.org/10.1145/2451116.2451170

[37] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. 2011. Automated Atomicity-violation Fixing. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 389–400. https://doi.org/10.1145/1993498.1993544

[38] Guoliang Jin, Aditya Thakur, Ben Liblit, and Shan Lu. 2010. Instrumentation and Sampling Strategies for Cooperative Concurrency Bug Isolation. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '10)*. ACM, New York, NY, USA, 241–255. https://doi.org/10.1145/1869459.1869481

[39] Guoliang Jin, Wei Zhang, Dongdong Deng, Ben Liblit, and Shan Lu. 2012. Automated Concurrency-bug Fixing. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. USENIX Association, Berkeley, CA, USA, 221–236. http://dl.acm.org/citation.cfm?id=2387880.2387902

[40] Yiyan Lin and Sandeep S. Kulkarni. 2014. Automatic Repair for Multithreaded Programs with Deadlock/Livelock Using Maximum Satisfiability. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA 2014)*. ACM, New York, NY, USA, 237–247. https://doi.org/10.1145/2610384.2610398

[41] Haopeng Liu, Yuxi Chen, and Shan Lu. 2016. Understanding and Generating High Quality Patches for Concurrency Bugs. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, New York, NY, USA, 715–726. https://doi.org/10.1145/2950290.2950309

[42] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. 2011. Dthreads: Efficient Deterministic Multithreading. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*. ACM, New York, NY, USA, 327–336. https://doi.org/10.1145/2043556.2043587

[43] Kai Lu, Xu Zhou, Tom Bergan, and Xiaoping Wang. 2014. Efficient Deterministic Multithreading Without Global Barriers. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '14)*. ACM, New York, NY, USA, 287–300. https://doi.org/10.1145/2555243.2555252

[44] Kai Lu, Xu Zhou, Xiao-Ping Wang, Tom Bergan, and Chen Chen. 2015. An Efficient and Flexible Deterministic Framework for Multithreaded Programs. *Journal of Computer Science and Technology* 30, 1 (01 Jan 2015), 42–56. https://doi.org/10.1007/s11390-015-1503-8

[45] Timothy Merrifield, Joseph Devietti, and Jakob Eriksson. 2015. High-performance Determinism with Total Store Order Consistency. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys '15)*. ACM, New York, NY, USA, Article 31, 13 pages. https://doi.org/10.1145/2741948.2741960

[46] Timothy Merrifield and Jakob Eriksson. 2013. Conversion: Multi-version Concurrency Control for Main Memory Segments. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys '13)*. ACM, New York, NY, USA, 127–139. https://doi.org/10.1145/2465351.2465365

[47] Robert H. B. Netzer and Barton P. Miller. 1992. What Are Race Conditions?: Some Issues and Formalizations. *ACM Lett. Program. Lang. Syst.* 1, 1 (March 1992), 74–88. https://doi.org/10.1145/130616.130623

[48] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2014. Deterministic Galois: On-demand, Portable and Parameterless. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. ACM, New York, NY, USA, 499–512. https://doi.org/10.1145/2541940.2541964

[49] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. 2009. Kendo: Efficient Deterministic Multithreading in Software. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*. ACM, New York, NY, USA, 97–108. https://doi.org/10.1145/1508244.1508256

[50] Soyeon Park, Shan Lu, and Yuanyuan Zhou. 2009. CTrigger: Exposing Atomicity Violation Bugs from Their Hiding Places. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*. ACM, New York, NY, USA, 25–36. https://doi.org/10.1145/1508244.1508249

[51] Soyeon Park, Yuanyuan Zhou, Weiwei Xiong, Zuoning Yin, Rini Kaushik, Kyu H. Lee, and Shan Lu. 2009. PRES: Probabilistic Replay with Execution Sketching on Multiprocessors. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09)*. ACM, New York, NY, USA, 177–192. https://doi.org/10.1145/1629575.1629593

[52] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. 2007. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture (HPCA '07)*. IEEE Computer Society, Washington, DC, USA, 13–24. https://doi.org/10.1109/HPCA.2007.346181

[53] Cedomir Segulja and Tarek S. Abdelrahman. 2012. Architectural support for synchronization-free deterministic parallel programming. In *IEEE International Symposium on High-Performance Comp Architecture*. 1–12. https://doi.org/10.1109/HPCA.2012.6169038

[54] Cedomir Segulja and Tarek S. Abdelrahman. 2014. What is the Cost of Weak Determinism?. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT '14)*. ACM, New York, NY, USA, 99–112. https://doi.org/10.1145/2628071.2628099

[55] Tiago M. Vale, João A. Silva, Ricardo J. Dias, and João M. Lourenço. 2016. Pot: Deterministic Transactional Execution. *ACM Trans. Archit. Code Optim.* 13, 4, Article 52 (Dec. 2016), 24 pages. https://doi.org/10.1145/3017993

[56] Jiří Šimša, Randy Bryant, and Garth Gibson. 2011. dBug: Systematic Testing of Unmodified Distributed and Multi-threaded Systems. In *Proceedings of the 18th International SPIN Conference on Model Checking Software*. Springer-Verlag, Berlin, Heidelberg, 188–193. http://dl.acm.org/citation.cfm?id=2032692.2032712

[57] Yin Wang, Terence Kelly, Manjunath Kudlur, Stéphane Lafortune, and Scott Mahlke. 2008. Gadara: Dynamic Deadlock Avoidance for Multithreaded Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, Berkeley, CA, USA, 281–294. http://dl.acm.org/citation.cfm?id=1855741.1855761

[58] Weiwei Xiong, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, and Zhiqiang Ma. 2010. Ad Hoc Synchronization Considered Harmful. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*. USENIX Association, Berkeley, CA, USA, 163–176. http://dl.acm.org/citation.cfm?id=1924943.1924955

[59] Wei Zhang, Junghee Lim, Ramya Olichandran, Joel Scherpelz, Guoliang Jin, Shan Lu, and Thomas Reps. 2011. ConSeq: Detecting Concurrency Bugs Through Sequential Errors. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, New York, NY, USA, 251–264. https://doi.org/10.1145/1950365.1950395

[60] Wei Zhang, Chong Sun, and Shan Lu. 2010. ConMem: Detecting Severe Concurrency Bugs Through an Effect-oriented Approach. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*. ACM, New York, NY, USA, 179–192. https://doi.org/10.1145/1736020.1736041

[61] Yu Zhang, Zhao-Peng Li, and Hui-Fang Cao. 2015. System-Enforced Deterministic Streaming for Efficient Pipeline Parallelism. *Journal of Computer Science and Technology* 30, 1 (01 Jan 2015), 57–73. https://doi.org/10.1007/s11390-015-1504-7

A Artifact Appendix

A.1 Abstract

Our artifact can be downloaded through Github or ACM DL, and it provides the source code of our QiThread library, as well as scripts to download all the benchmarks used in our paper, automatically build them, and generate time data from these benchmarks using our library. Our installation instructions are tested on a Ubuntu Linux 16.04 machine with an x86_64 architecture, and our experiments are conducted on a machine having two Xeon E5-2670 processors with a total of 32 logical processors and 128GB memory. While our artifact should be runnable on any multi-core machine, some benchmarks are configured to create 24 threads in our experiments. All benchmarks and dataset needed for evaluation can be downloaded using our scripts or are already included in our artifact.

A.2 Artifact Check-List (Meta-Information)

- **Algorithm:** Semantic-aware scheduling algorithms for a synchronization-determinism runtime.
- **Program:** SPLASH-2x, NPB 3.3.1, PARSEC 2.0, Phoenix 2, pbzip2, aget, pfscan, bdb_bench3n, openldap, mplayer mencoder, redis, ImageMagick, and STL libstdc++-v3. Scripts are included to automatically download and build them.
- **Compilation:** GCC 5.4 and GCC 4.7 installed from Ubuntu repository. Instructions are provided to install them.
- **Data set:** Included or downloaded with scripts provided.
- **Run-time environment:** Tested on Ubuntu Linux 16.04 x86_64. Instructions are provided to install all the necessary libraries and tools from a fresh Ubuntu installation.
- **Hardware:** Some benchmarks are configured to create 24 threads in our experiments, so we recommend a configuration with at least 24 logical processors.
- **Metrics:** Overheads against nondeterministic executions are reported for all the benchmarks with all scheduling policies turned on.
- **Output:** The metrics are written to csv files in a folder created for each evaluation script execution.
- **Experiments:** A script to evaluate everything is provided. Configuration files and evaluation scripts for evaluating each individual benchmark suite are also included.
- **How much disk space required (approximately)?:** About 40 GB.
- **How much time is needed to complete experiments (approximately)?:** For our machine, it takes about 70 hours to run all the benchmarks with the default configuration.
- **Publicly available?:** Yes, it's available on both GitHub and ACM DL.

A.3 Description

A.3.1 How Delivered

Our artifact is available on Github:
https://github.com/chyiz/QiThread
and ACM DL:
https://doi.org/10.1145/3300171

A.3.2 Hardware Dependencies

We conducted our experiments on a machine with two Xeon E5-2670 processors and 128GB memory. While our code should be runnable on any multi-core machines, we recommend a configuration with more than 24 logical processors, since benchmarks are sometimes configured to create 24 threads in our experiments.

A.3.3 Software Dependencies

Our build scripts assume a Ubuntu Linux 16.04 installation on an x86_64 platform. For other Linux distributions, you may use the provided Dockerfile to create a Docker image. You need to have Docker installed to use the Dockerfile.

Software packages required to build the library and benchmarks will be installed automatically by the build scripts. No other software is needed to build or run the experiments.

A.4 Installation

Clone our repository from Github:

```
git clone https://github.com/chyiz/QiThread.git
```

The easiest way to use our artifact is to create a Docker image from the Dockerfile provided in the repository. In the project root folder, run:

```
sudo docker build -t qithread .
```

After a while, a Docker image will be available with tag `qithread`. You may run it with:

```
sudo docker run -it qithread
```

You will then be able to run all the experiments inside the Docker image.

Alternatively, you can compile our library and all the benchmarks directly. However, our build scripts assume an x86_64 Ubuntu 16.04 environment. For other Linux distributions, you may need to make some necessary changes or just use the Dockerfile.

Please see more detailed installation instructions in the README file provided in the artifact.

A.5 Experiment Workflow

To run the experiments, go to the `eval` folder and execute:

```
./eval_policy.py all-default-config.cfg
```

The evaluation results will be saved to a folder named `all-default-config<Execution_Date_and_Time>`.

A symbolic link `current` is also modified to always point to the latest evaluation results.

To extract all results into one file, run:

```
./get_all_results.sh current/ > results.csv
```

The results are then stored into a comma separated values (csv) file that can be imported to a spreadsheet.

At last, to generate the graphs used in our paper, open `generate-figure.ipynb` in the `eval` folder with Jupyter Notebook and run all cells. It will read the `results.csv` file in the same folder by default.

A.6 Evaluation and Expected Result

The results contain all 108 benchmarks evaluated, and each of them reports the average overhead and multiple execution times when running with different configurations. Each benchmark will have four or five rows in the final results: one `non-det` row representing nondeterministic execution as the baseline, one `no-hint` row for round-robin scheduling with no performance hints, one `hinted` row for round-robin scheduling with all performance hints, one `all-policies` row representing the QiThread results with all policies on, and some benchmarks have one `no-pcs-hint` row for round-robin scheduling with only soft barrier hints but no performance critical section hints.

Given the hardware differences, it may be difficult to reproduce the exact same numbers as in our paper, but we expect the trend to be the same.

A.7 Experiment Customization

The experiment parameters can be adjusted by modifying the evaluation configuration files (`.cfg` in the `eval` folder). Several sample configuration files are provided in the artifact. For example, `all-default-config.cfg` runs all the benchmarks under (1) round-robin scheduling with or without performance hints and (2) QiThread with all policies enabled, and `all-compare-policies.cfg` additionally tries other different combinations of policies. There are also configuration files, like `phoenix-compare-policies.cfg`, that only execute one benchmark suite.

You may start by copying a segment from one of the sample configuration files and modify some of the parameters. Some interesting ones are listed below:

- `REPEATS`: number of tests to run for a benchmark. Execution times will be averaged to get an overhead.
- `INPUTS`: command line parameters for a benchmark.
- `EXPORT`: environment variables for a benchmark.
- `INIT_ENV_CMD`: commands to initialize the environment before running a benchmark.
- `TARBALL` or `GZIP`: data files that need to be extracted before running a benchmark.
- `REQUIRED_FILES`: files that need to be copied to the evaluation folder before running a benchmark. This can be configuration files and data files that do not require decompression.
- `RUN_CONFIGS`: a list of scheduling policies to try.

A.8 Methodology

Submission, reviewing, and badging methodology:

- http://cTuning.org/ae/submission-20180713.html
- http://cTuning.org/ae/reviewing-20180713.html
- https://www.acm.org/publications/policies/artifact-review-badging