DTHREADS: Efficient and Deterministic Multithreading

Tongping Liu Charlie Curtsinger Emery D. Berger

Dept. of Computer Science
University of Massachusetts, Amherst
Amherst, MA 01003
{tonyliu,charlie,emery}@cs.umass.edu

Abstract

Multithreaded programming is notoriously difficult to get right. A key problem is non-determinism, which complicates debugging, testing, and reproducing errors in multithreaded applications. One way to simplify multithreaded programming is to enforce deterministic execution. However, past deterministic systems are incomplete or impractical. Language-based approaches require programmers to write their code in specialized languages. Other systems require program modification, do not ensure determinism in the presence of data races, do not work with general-purpose multithreaded programs, or suffer substantial performance penalties (up to 8× slower than `pthreads`) that limit their usefulness.

This paper presents DTHREADS, an efficient deterministic multithreading system for unmodified C/C++ applications. DTHREADS not only prevents semantic errors like race conditions and deadlocks, but also can enhance performance by eliminating false sharing of cache lines. DTHREADS leverages virtual memory and process isolation, combined with a deterministic commit protocol, to ensure robust deterministic execution with low runtime overhead. Experimental results show that DTHREADS substantially outperforms a state-of-the-art deterministic runtime system, and often matches—and occasionally exceeds—the performance of `pthreads`.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming–Parallel Programming; D.2.5 [*Software Engineering*]: Testing and Debugging–Debugging Aids

General Terms Design, Reliability, Performance

Keywords Deterministic Multithreading, Determinism, Parallel Programming, Concurrency, Debugging, Multicore

1. Introduction

The advent of multicore architectures has increased the demand for multithreaded programs, but writing them remains painful. It is notoriously far more challenging to write concurrent programs than sequential ones because of the wide range of concurrency errors, including deadlocks and race conditions [18, 23, 24]. Because thread interleavings are non-deterministic, different runs of the same multithreaded program can unexpectedly produce different results. These "Heisenbugs" greatly complicate debugging, and eliminating them requires extensive testing to account for possible thread interleavings [2, 13].

Instead of testing, one promising alternative approach is to attack the problem of concurrency bugs by eliminating its source: non-determinism. A fully *deterministic multithreaded system* would prevent Heisenbugs by ensuring that executions of the same program with the same inputs always yield the same results, even in the face of race conditions in the code. Such a system would not only dramatically simplify debugging of concurrent programs [15] and reduce their attendant testing overhead, but would also enable a number of other applications. For example, a deterministic multithreaded system would greatly simplify record and replay for multithreaded programs [16, 22] and the execution of multiple replicas of multithreaded applications for fault tolerance [4, 7, 11, 26].

Several recent software-only proposals aim at providing deterministic multithreading, but these all suffer from a variety of disadvantages. Language-based approaches are effective at removing determinism but require programmers to write code in specialized languages, which can be impractical [10, 12, 29]. Recent deterministic systems that target legacy programming languages (especially C/C++) are either incomplete or impractical. Kendo ensures determinism of synchronization operations with low overhead, but does not guarantee determinism in the presence of data races [25]. Grace prevents all concurrency errors but is limited to fork-join programs, and although it is efficient, it can require code modifications to avoid large runtime overheads [6]. CoreDet, a compiler and runtime system, enforces deterministic execution for arbitrary multithreaded C/C++ programs [3]. However, it exhibits prohibitively high overhead (running up to 8× slower than `pthreads`; see Section 4) and generates thread interleavings at arbitrary points in the code, complicating program debugging and testing.

Contributions: This paper presents **DTHREADS**, an efficient deterministic runtime system for multithreaded C/C++ applications. DTHREADS guarantees deterministic execution of multithreaded programs even in the presence of data races (notwithstanding external sources of non-determinism like I/O): given the same sequence of inputs, a program using DTHREADS always produces the same output. DTHREADS' deterministic commit protocol not only eliminates data races but also prevents lock-based deadlocks.

DTHREADS is easy to deploy: it works as a direct replacement for the `pthreads` library, requiring no code modifications or recompilation. DTHREADS is also efficient. DTHREADS leverages process isolation and virtual memory protection to track and isolate concurrent memory updates, which it deterministically commits. Not only does this approach greatly reduce overhead versus approaches that use software read and write barriers, it also eliminates cache-line based false sharing, a notorious performance problem for multithreaded programs. These two features combine to en-

able DTHREADS to nearly match or even exceed the performance of `pthreads` for the majority of the benchmarks examined here. DTHREADS thus marks a significant improvement over the state of the art in deployability and performance, and provides promising evidence that fully deterministic multithreaded programming may be practical.

The remainder of this paper is organized as follows. Section 2 describes the DTHREADS architecture and algorithms in depth, and Section 3 discusses key limitations. Section 4 evaluates DTHREADS experimentally, comparing its performance and scalability to `pthreads` and CoreDet. Section 5 provides an overview of related work, Section 6 describes future directions, and Section 7 concludes.

2. DTHREADS Architecture

```
1  int a = 0;            void thread1() {    void thread2() {
2  int b = 0;              if(b == 0) {        if(a == 0) {
3  int main() {              a = 1;              b = 1;
4    spawn(thread1);       }                   }
5    spawn(thread2);     }                   }
6    print(a, b);
7  }
```

Figure 1. A simple multithreaded program with a data race.

2.1 Overview

Figure 1 shows an example multithreaded program that, because of data races, non-deterministically produces the outputs "1,0," "0,1" and "1,1." The order in which these modifications occur can change from run to run, resulting in non-deterministic output. Using DTHREADS, this program will *always* produce the output "1,1". If this is not the desired behavior, the fact that the result is reproducible would make it simple for the developer to reproduce and locate the data race. DTHREADS ensures this determinism using the following key notions:

Isolated memory access: In DTHREADS, threads are implemented using separate processes, an idea borrowed from Grace [6]. Because processes have separate address spaces, they are a convenient mechanism to isolate memory accesses between threads. DTHREADS uses this isolation to ensure that updates to shared memory are not visible to other threads until a synchronization point is reached. The time between synchronization points can be thought of as a single, atomic *transaction*. Memory isolation, in combination with a deterministic commit protocol, is sufficient to guarantee deterministic execution even in the presence of data races. Section 2.2 discusses the implementation of this mechanism in depth.

Deterministic memory commit: Because multithreaded programs frequently use updates to shared memory to communicate, DTHREADS must implement a mechanism to expose one thread's updates to all other threads. To ensure deterministic execution, these updates must be exposed at deterministic times, and in deterministic order. Rather than using retired instruction counts to demarcate commit boundaries (as done by CoreDet and Kendo), DTHREADS updates shared state only at synchronization points: thread create and exit; mutex lock and unlock; condition variable wait and signal; and barrier wait. These natural synchronization points make DTHREADS code more *robust*: when the boundary is the number of instructions retired, it is difficult for programmers to know when a transaction ends. Such boundaries could vary depending on the underlying architecture, and would also be input-dependent, meaning that slightly different inputs could lead to dramatically different thread interleavings. By contrast, DTHREADS

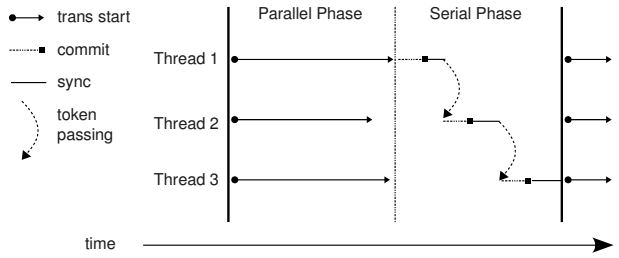

Figure 2. An overview of DTHREADS phases. Program execution with DTHREADS alternates between parallel and serial phases.

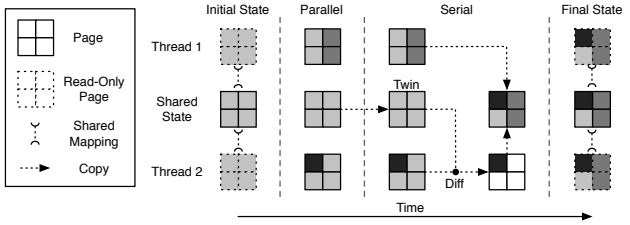

Figure 3. An overview of DTHREADS execution.

guarantees that only changes to the sequence of synchronization operations will affect the order in which updates are exposed.

When it comes time to for a thread to commit updates to shared state, DTHREADS compares local updates to a "twin" (copy) of the original shared page, and then writes only modified bytes to the shared state. This idea is borrowed from the distributed shared memory systems TreadMarks and Munin [14, 20]. The order in which threads write their updates to shared state is enforced by a single global token passed from thread to thread, in deterministic order. Section 2.3 describes this mechanism in more detail.

Deterministic Synchronization: DTHREADS supports the full array of `pthreads` synchronization primitives. Current operating systems make no guarantees about the order in which threads will acquire locks, wake from condition variables, or pass through barriers. DTHREADS re-implements these primitives to ensure deterministic ordering via its global token. Details on the DTHREADS implementations of these primitives are given in Section 2.4.

Fixing the Data Race Example: Returning to the example program in Figure 1, we can now illustrate how memory isolation and a deterministic commit order ensure deterministic output. At the start of execution, thread 1 and thread 2 see the same view of shared state, with $a = 0$ and $b = 0$. Because changes by one thread to the value of a or b will not be exposed to the other until thread exit, both threads' checks on line 2 will be true. Thread 1 will set the value of a to 1, and thread 2 will set the value of b to 1. These threads then commit their updates to shared state and exit, with thread 1 always committing before thread 2. The main thread then has an updated view of shared memory, and will thus print "1, 1" on every execution.

2.2 Isolated Memory Access

As described above, in order to achieve deterministic memory access, DTHREADS isolates memory accesses among different threads between commit points, and commits the updates of each thread deterministically.

To isolate the memory access among different threads, DTHREADS replaces threads by processes. In a multithreaded program running with `pthreads`, threads share all memory except for the stack. Changes to memory immediately become visible to all

other threads. Threads share the same file descriptors, sockets, device handles, and windows. By contrast, because DTHREADS runs threads in separate processes, it must manage these shared resources explicitly.

2.2.1 Thread Creation

DTHREADS replaces the `pthread_create()` function. Using the `clone` system call, DTHREADS controls which resources are shared between processes. The `CLONE_FILES` flag (shown on line 3 of Figure 13), is used to create processes that have disjoint address spaces but share the same file descriptor table.

2.2.2 Deterministic Thread Index

POSIX does not guarantee deterministic process identifiers. To avoid exposing this non-determinism to threads running as processes, DTHREADS uses an internal thread index and shims the `getpid()` function. This internal thread index is managed using a single global variable that is incremented on thread creation. This unique thread index is also used to manage per-thread heaps and as an offset into an array of thread entries.

2.2.3 Shared Memory

In order to create the illusion of different threads sharing the same address space, DTHREADS uses memory mapped files to share memory across processes (globals and the heap, but not the stack; see Section 3).

DTHREADS creates two different mappings for both the heap and the globals. One is a *shared* mapping, which is used to hold shared state. The other is a *private*, copy-on-write (COW) per-process mapping that each process works on directly. Private mappings are linked to the shared mapping through a single fixed-size memory-mapped file. Reads initially go directly to the shared mapping, but after the first write operation, both reads and writes are entirely private.

Memory allocations from the shared heap memory use a scalable per-thread heap organization loosely based on Hoard [5] and built using HeapLayers [8]. DTHREADS divides the heap into a fixed number of sub-heaps (currently 16). Each thread uses a hash of its thread index to find the appropriate sub-heap.

2.3 Deterministic Memory Commit

Figure 2 illustrates the progression of parallel and serial phases. To guarantee determinism, DTHREADS isolates memory accesses in the parallel phase. Those memory accesses in the parallel phase work on their own private copies of memory; that is, updates are not shared while in the parallel phase. When a synchronization point is reached, updates are applied (and made visible) in a deterministic order. This section describes the mechanism used to alternate between parallel and serial execution phases and guarantee deterministic commit order, and the details of commits to shared memory.

2.3.1 The Fence and the Token

The boundary between the parallel and serial phase is the internal fence. It is not possible to implement the internal fence using the standard `pthreads` barrier because the number of threads required to proceed can change during execution, a feature that `pthreads`' barrier does not support (see Section 2.4).

Figure 4 presents pseudocode for the internal fence. Threads must wait at the fence until all threads from the previous fence have departed. Those waiting threads must block until the departure phase (lines 8–11). If the thread is the last to enter the fence, it sets the departure phase and wakes the waiting threads (lines 12-15). As threads leave the fence, they decrement the waiting thread count. The last thread to leave sets the fence to the arrival phase and wakes any waiting threads (lines 17-21).

```
1  void waitFence(void) {
2      lock();
3      while(!isArrivalPhase()) {
4          CondWait();
5      }
6
7      waiting_threads++;
8      if(waiting_threads < live_threads) {
9          while(!isDeparturePhase()) {
10             CondWait();
11         }
12     } else {
13         setDeparturePhase();
14         CondBroadcast();
15     }
16
17     waiting_threads--;
18     if (waiting_threads == 0) {
19         setArrivalPhase();
20         CondBroadcast();
21     }
22     unlock();
23 }
```

Figure 4. Pseudocode for DTHREADS' internal fence. (§ 2.3.1)

```
1  void waitToken() {
2      waitFence();
3      while(token_holder != thread_id) {
4          yield();
5      }
6  }
7  void putToken() {
8      token_holder = token_queue.nextThread();
9  }
```

Figure 5. Pseudocode for token management (§ 2.3.1).

A key mechanism used by DTHREADS is its global token, which it uses to enforces determinism. In order to guarantee determinism, each thread must wait for the token before it can communicate with other threads. The token is a shared pointer that points to the next runnable thread entry. Since the token is unique in the entire system, waiting for the token guarantees a global order for all operations in the serial phase.

DTHREADS uses two internal subroutines to manage tokens. `waitToken` first waits at the internal fence and then waits to acquire the global token in order to enter serial mode. `putToken` passes the token to the next waiting thread.

To achieve determinism (see Figure 2), those threads leaving the parallel phase must wait at the internal fence before they can enter into the serial phase (by calling `waitToken`). Note that it is crucial that threads wait at the fence even for a thread which is guaranteed to obtain the token next, since one thread's commits can affect another threads' behavior if there is no fence.

2.3.2 Commit Protocol

Figure 3 shows the steps taken by DTHREADS to capture modifications to shared state and expose them in a deterministic order. At the beginning of the parallel phase, threads have a read-only mapping for all shared pages. If a thread writes to a shared page during the parallel phase, this write is trapped and re-issued on a private copy of the shared page. Reads go directly to shared memory and are not trapped. In the serial phase, threads commit their updates one at a time. The first thread to commit to a page can directly copy its private copy to the shared state, but subsequent commits must copy only the modified bytes. DTHREADS computes diffs from a twin page, an unmodified copy of the shared page created at the

beginning of the serial phase. At the end of the serial phase, private copies are released and these addresses are restored to read-only mappings of the shared memory.

```
1  void atomicBegin() {
2      foreach(page in modifiedPages) {
3          page.writeProtect();
4          page.privateCopy.free();
5      }
6      modifiedPages.emptyList()
7  }
```

Figure 6. Pseudocode for `atomicBegin` (§ 2.3.2).

Figure 6 presents pseudocode for `atomicBegin`. First, all previously-written pages are write-protected (line 3). The old working copies of these pages are then discarded, and mappings are updated to reference the shared state (line 4).

```
1  void atomicEnd() {
2      foreach(page in modifiedPages) {
3          if(page.writers > 1 && !page.hasTwin()) {
4              page.createTwin();
5          }
6
7          if(page.version == page.localCopy.version) {
8              page.copyCommit();
9          } else {
10             page.diffCommit();
11         }
12
13         page.writers--;
14         if(page.writers == 0 && page.hasTwin()) {
15             page.twin.free();
16         }
17         page.version++;
18     }
19 }
```

Figure 7. Pseudocode for `atomicEnd` (§ 2.3.2).

Figure 7 presents pseudocode for `atomicEnd`. `atomicEnd` commits all changes from the current transaction to the shared page. For each modified page with more than one writer, DTHREADS ensures that a twin page is created (lines 3-5). If the version number of the private copy matches the shared page, then the current thread must be the first thread to commit. In this case, the entire private copy can be copied to the shared state (lines 7 and 8). If the version numbers do not match, then another thread has already committed changes to the page and a diff-based commit must be used (lines 9-10). After changes have been committed, the number of writers to the page is decremented (line 13), and if there are no writers left to commit, the twin page is freed (lines 14-16). Finally, the shared page's version number is incremented (line 17).

2.4 Deterministic Synchronization

DTHREADS enforces determinism for the full range of synchronizations in the `pthreads` API, including locks, conditional variables, barriers and various flavors of thread exit.

2.4.1 Locks

DTHREADS uses a single global token to guarantee atomicity in the serial phase. This means that all of a program's locks are turned into a single global lock. While this approach has the potential to compromise performance, it is necessary to guarantee a deterministic order of commits to shared memory.

Figure 8 presents the pseudocode for lock acquisition. First, DTHREADS checks to see if the current thread is already holding any locks. If not, the thread first waits for the token, commits

```
1  void mutex_lock() {
2      if(lock_count == 0) {
3          waitToken();
4          atomicEnd();
5          atomicBegin();
6      }
7      lock_count++;
8  }
```

Figure 8. Pseudocode for `mutex_lock` (§ 2.4.1).

changes to shared state by calling `atomicEnd`, and begins a new atomic section (lines 2-6). Finally, the thread increments the number of locks it is currently holding. This count must be kept to ensure that a thread will not pass the token until it has release all of the locks it acquired in the serial phase.

```
1  void mutex_unlock(){
2      lock_count--;
3      if(lock_count == 0) {
4          atomicEnd();
5          putToken();
6          atomicBegin();
7          waitFence();
8      }
9  }
```

Figure 9. Pseudocode for `mutex_unlock` (§ 2.4.1).

Figure 9 presents the implementation of `mutex_unlock`. First, the thread decrements its lock count (line 2). If no more locks are held, any local modifications are committed to shared state, the token is passed, and a new atomic section is started (lines 3-6). Finally, the thread waits on the internal fence until the start of the next round's parallel phase (line 7).

2.4.2 Condition Variables

Guaranteeing determinism for condition variables is more complex than for mutexes because the operating system does not guarantee processes will wake up in the order they wait for a condition variable.

```
1  void cond_wait() {
2      waitToken();
3      atomicEnd();
4
5      token_queue.removeThread(thread_id);
6      live_threads--;
7      cond_queue.addThread(thread_id);
8      putToken();
9
10     while(!threadReady()) {
11         real_cond_wait();
12     }
13
14     while(token_holder != thread_id) {
15         yield();
16     }
17     atomicBegin();
18 }
```

Figure 10. Pseudocode for `cond_wait` (§ 2.4.2).

Figure 10 presents pseudocode for the DTHREADS implementation of `cond_wait`. When a thread calls `cond_wait`, it first acquires the token and commits local modifications (lines 2 and 3). It removes itself from the token queue (line 4) because threads waiting on a condition variable do not participate in the serial phase until they are woken up. The thread decrements the live thread count

(used for the fence between parallel and serial phases), adds itself to the condition variable's queue, and passes the token (lines 6-8). While threads are waiting on DTHREADS condition variables, they are suspended on a `pthreads` condition variable (lines 10-12). Once a thread is woken up, it busy-waits on the token and finally begins the next transaction (lines 14-17). Threads must acquire the token before proceeding because `cond_wait` is called within a mutex's critical section.

```
1  void cond_signal() {
2      if(token_holder != thread_id) {
3          waitToken();
4      }
5      atomicEnd();
6
7      if(cond_queue.length == 0) {
8          return;
9      }
10
11     lock();
12     thread = cond_queue.removeNext();
13     token_queue.insert(thread);
14     live_threads++;
15     thread.setReady(true);
16     real_cond_signal();
17     unlock();
18     atomicBegin();
19 }
```

Figure 11. Pseudocode for `cond_signal` (§ 2.4.2).

The DTHREADS implementation of `cond_signal` is presented in Figure 11. The calling thread first waits for the token, and then commits any local modifications (lines 2-5). If no threads are waiting on the condition variable, this function returns immediately (lines 7-9). Otherwise, the first thread in the condition variable queue is moved to the head of the token queue and the live thread count is incremented (lines 12-14). This thread is then marked as ready and woken up from the real condition variable, and the calling thread begins another transaction (lines 15-18).

In `pthreads`, the only difference between `cond_signal` and `cond_broadcast` is that `cond_signal` just wakes the first waiting thread and `cond_broadcast` wakes all waiting threads. In DTHREADS, `cond_broadcast` wakes all threads but only *one* is marked as ready because these threads are holding locks, and therefore running in the serial phase. The threads not marked as ready will wait on the condition variable again.

2.4.3 Barriers

As with condition variables, DTHREADS must ensure that threads waiting on a barrier do not disrupt the token passing of running threads. DTHREADS removes threads entering into the barrier from the token queue and places them on the corresponding barrier queue.

To avoid blocking on the barrier, the last thread entering into the barrier moves all threads to the runnable queue and increases the fence's thread count.

Figure 12 presents pseudocode for `barrier_wait`. The calling thread first waits for the token to commit any local modifications in order to ensure deterministic commit (lines 2 and 3). If the current thread is the last to enter the barrier, then DTHREADS moves the entire list of threads on the barrier queue to the token queue (line 7), increases the fence's thread count (line 8), and passes the token to the first thread in the barrier queue (line 9). Otherwise, DTHREADS removes the current thread from the token queue (line 12), places it on the barrier queue (line 13), and releases token (line 14). Finally, the thread waits on the actual barrier (line 19).

```
1  void barrier_wait() {
2      waitToken();
3      atomicEnd();
4      lock();
5      if(barrier_queue.length == barrier_count-1) {
6          token_holder = barrier_queue.first();
7          live_threads += barrier_queue.length;
8          barrier_queue.moveAllTo(token_queue);
9      } else {
10         token_queue.remove(thread_id);
11         barrier_queue.insert(thread_id);
12         putToken();
13     }
14     unlock();
15     atomicBegin();
16     real_barrier_wait();
17 }
```

Figure 12. Pseudocode for barrier_wait (§ 2.4.3).

2.4.4 Thread Creation and Exit

To guarantee determinism, thread creation and exit are performed in the serial phase. Newly-created threads are immediately added to the token queue. Creating a thread does not immediately release the token; this approach allows a single thread to quickly create multiple child threads without having to wait for a new serial phase for each creation.

```
1  void thread_create() {
2      waitToken();
3      clone(CLONE_FS | CLONE_FILES | CLONE_CHILD);
4      if(child_process) {
5          thread_id = next_thread_index;
6          next_thread_index++;
7          notifyChildRegistered();
8          waitParentProadcast();
9      } else {
10         waitChildRegistered();
11     }
12 }
```

Figure 13. Pseudocode for `thread_create` (§ 2.4.4).

Figure 13 presents pseudocode for thread creation. The caller first waits for the token before proceeding (line 2). It then creates a new process with shared file descriptors but a distinct address space using the `clone` system call (line 3). The newly created child obtains the global thread index (line 5), places itself in the token queue (line 6), and notifies the parent that child has registered itself in the active list (line 7). The child thread then waits for the parent to reach a synchronization point.

```
1  void thread_exit() {
2      waitToken();
3      atomicEnd();
4      token_queue.remove(thread_id);
5      live_threads--;
6      putToken();
7      real_thread_exit();
8  }
```

Figure 14. Pseudocode for `thread_exit` (§ 2.4.4).

Figure 14 presents pseudocode for `thread_exit`. When this function is called, the caller first waits for the token and then commits any local modifications (line 3). It then removes itself from the token queue (line 4) and decreases the number of threads required to proceed to the next phase (line 5). Finally, the thread passes its token to the next thread in the token queue (line 6) and exits (line 7).

2.4.5 Thread Cancellation

DTHREADS implements thread cancellation in the serial phase. `thread_cancel` can only be called while holding the token. If the thread being cancelled is waiting on a condition variable or barrier, it is removed from the queue. The real `pthread_cancel` function is then called, and the calling thread can proceed.

2.5 Optimizations

DTHREADS performs a number of optimizations to improve performance.

Single-threaded execution: When only one thread is running, DTHREADS does not employ memory protection and treats all synchronization operations as no-ops. In addition, when only one thread is active because other threads are waiting on conditional variables, DTHREADS does not try to commit local changes to the shared mapping (or discard private dirty pages). Updates are only committed when the thread issues a `cond_signal` or `cond_broadcast` call, which will wake up a thread and thus require publication of any updates.

Lazy twin creation: Twin pages are only created when a page has multiple writers during the same transaction. During the commit phase, the single writer can directly copy its working copy to the shared state without performing a diff. This reduces the overhead in the common case, where a single thread is the exclusive writer of a page.

Exclusive/first writers: If one thread is the only writer on one page, or if it is the first thread to commit to a page, it can directly copy its working copy to shared state. DTHREADS currently relies on the comparison between local version number and the global page version number. In the page handler, each thread obtains the version number for every dirty page. In the commit phase, DTHREADS compares this local version number with the global version number to check whether it is the first to commit (see Figure 7).

Parallelization: DTHREADS attempts to expose as much parallelism as possible in the runtime system itself. One optimization is for `atomicBegin`, which performs cleanup tasks, including releasing private page frames and resetting pages to read-only mode by calling the `madvise` and `mprotect` system calls. If all this cleanup work is done simultaneously for all threads in the beginning of parallel phase (Figure 2), this can hurt performance for some benchmarks.

Since `atomicBegin` does not affect other the behavior of other threads, most of its work can be parallelized with other threads' commit operations without holding the global token. With this optimization, the token is passed to the next thread as soon as possible, saving time in the serial phase. Before passing the token, any local copies of pages that have been modified by other threads must be discarded, and the shared read-only mapping is restored. This ensures all threads have a complete image of this page which later transactions may refer to. In the actual implementation, `atomicEnd` performs this cleanup.

3. Discussion

This section analyzes some key limitations of DTHREADS that restrict its ability to run certain programs, limit the extent of determinism it can guarantee, or potentially affect performance.

Unsupported programs: DTHREADS currently does not support programs with ad hoc synchronization that avoids the `pthreads` library, such as those that use atomic operations implemented in assembly. However, the upcoming C++0X standard includes a library interface for atomic operations [19, pp. 1107–1128], and a future version of DTHREADS could correctly implement these by intercepting these library calls and treating them as synchronization points. While ad hoc synchronization is a common practice, it is also a notorious source of bugs; Xiong et al. show that 22–67% of the uses of ad hoc synchronization lead to bugs or severe performance issues [30].

DTHREADS also currently does not write-share the stack across threads, so that updates made by a thread to a stack variable would not be reflected back to the parent, which could cause a program to fail. Passing stack variables to a thread for modification is extremely error-prone and generally deprecated, making this a rare coding practice.

External determinism: While DTHREADS provides internal determinism, it does not guarantee determinism when a program's behavior depends on external sources of non-determinism, such as system time or I/O events. Incorporation of DTHREADS in the dOS framework, an OS proposal that enforces system-level determinism, would provide full deterministic execution, although this remains future work [4].

Runtime performance: Section 4 shows that DTHREADS can provide high performance for a number of applications; in fact, for the majority of the benchmarks examined, DTHREADS matches or even exceeds the performance of `pthreads`. However, DTHREADS could occasionally degrade performance, sometimes substantially. One way it could do so would be to exhibit an intensive use of locks (that is, acquiring and releasing locks at high frequency), which are much more expensive in DTHREADS than in `pthreads`. However, because of its determinism guarantees, DTHREADS could allow programmers to greatly reduce their use of locks, and thus improve performance. Other application characteristics, also explored in Section 4.3, can also impair performance with DTHREADS.

Memory consumption: Finally, because DTHREADS creates private, per-process copies of modified pages between commits, it can increase a program's memory footprint by the number of modified pages between synchronization operations. This increased footprint does not seem to be a problem in practice, both because the number of modified pages is generally far smaller than the number of pages read, and because it is transitory: all private pages are relinquished to the operating system (via `madvise`) at the end of every commit operation.

4. Evaluation

We perform our evaluation on an Intel Core 2 dual-processor CPU system equipped with 16GB of RAM. Each processor is a 4-core 64-bit Xeon running on at 2.33GHZ with a 4MB L2 cache. The operating system is an unmodified CentOS 5.5, running with Linux kernel version 2.6.18-194.17.1.el5.

4.1 Methodology

We evaluate the performance and scalability of DTHREADS versus CoreDet and `pthreads` across the PARSEC [9] and Phoenix [28] benchmark suites. We do not include results for `bodytrack`, `fluidanimate`, `x.264`, `facesim`, `vips`, and `raytrace` benchmarks from PARSEC, since they do not currently work with DTHREADS (note that many of these also do not work for CoreDet).

In order to compare performance directly against CoreDet, which relies on the LLVM infrastructure [21], all benchmarks are compiled with the LLVM compiler at the "-O5" optimization level [21]. Since DTHREADS does not currently support 64-bit binaries, all benchmarks are compiled for 32 bit environments (using the "-m32" compiler flag). Each benchmark is executed ten times on a quiescent machine. To reduce the effect of outliers, the lowest and highest execution times for each benchmark are discarded, so each result represents the average of the remaining eight runs.

Tuning CoreDet: The performance of CoreDet [3] is extremely sensitive to three parameters: the granularity for the ownership ta-

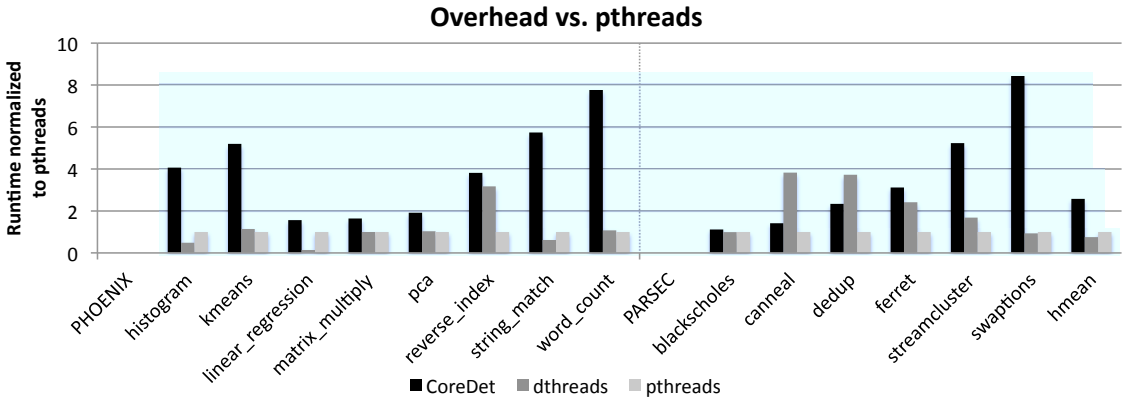

Figure 15. Normalized execution time with respect to `pthreads` (lower is better). For 9 of the 14 benchmarks, DTHREADS runs nearly as fast or faster than `pthreads`, while providing deterministic behavior.

ble (in bytes), the quantum size (in number of instructions retired), and the choice between full serial mode and reduced serial mode. We compare the performance and scalability of DTHREADS with the best possible results that we could obtain for CoreDet on our system—that is, with the lowest average normalized runtimes—after an extensive search of the parameter space (six possible granularities and 8 possible quanta, for each benchmark). The results presented here are for a 64-byte granularity and a quantum size of 100,000 instructions, in full serial mode.

For all scalability experiments, we logically disable CPUs using Linux's CPU hotplug mechanism, which allows us to disable or enable individual CPUs by writing "0" (or "1") to a special pseudo-file (`/sys/devices/system/cpu/cpuN/online`).

4.2 Determinism

We first experimentally verify DTHREADS' ability to ensure determinism by executing the *racey* determinism tester [25]. This stress test contains, as its name suggests, numerous data races and is thus extremely sensitive to memory-level non-determinism. DTHREADS reports the same results for 2,000 runs.

4.3 Performance

We next compare the performance of DTHREADS to CoreDet and `pthreads`. Figure 15 presents these results graphically (normalized to `pthreads`); Table 1 provides detailed numbers.

DTHREADS outperforms CoreDet on 12 out of 14 benchmarks (running between 20% and 11.2× faster), while for 9 benchmarks, DTHREADS provides nearly the same as or higher performance than `pthreads`. Because DTHREADS isolates updates in separate processes, it can improve performance by eliminating false sharing—since concurrent "threads" actually execute in different address spaces, there is no coherence traffic between synchronization points. DTHREADS eliminates catastrophic false sharing in the `linear_regression` benchmark, allowing it to execute over 7× faster than `pthreads` and 11× faster than CoreDet. The `string_match` benchmark exhibits a similar, though less dramatic, false sharing problem, allowing DTHREADS to run almost 60% faster than `pthreads` and 9× faster than CoreDet. Two benchmarks, `histogram` and `swaptions`, also run faster with DTHREADS than with `pthreads` (2× and 6%, respectively; 2.7× and 9× faster than with CoreDet). We believe but have not yet verified that the reason is false sharing.

For some benchmarks, DTHREADS incurs modest overhead. For example, unlike most benchmarks examined here, which create long-lived threads, the `kmeans` benchmark creates and destroys over 1,000 threads in the course of its execution. While Linux processes are relatively lightweight, creating and tearing down a process is still more expensive than the same operations for threads, accounting for a 14% performance degradation of DTHREADS over `pthreads` (though it runs 4.6× faster than CoreDet).

DTHREADS runs substantially slower than `pthreads` for 4 of the 14 benchmarks examined here. The `ferret` benchmark relies on an external library to analyze image files during the first stage in its pipelined execution model; this library makes intensive (and in the case of DTHREADS, unnecessary) use of locks. Lock acquisition and release in DTHREADS imposes higher overhead than ordinary `pthreads` mutex operations. More importantly in this case, the intensive use of locks in one stage forces DTHREADS to effectively serialize the other stages in the pipeline, which must repeatedly wait on these locks to enforce a deterministic lock acquisition order. The other three benchmarks (`canneal`, `dedup`, and `reverse_index`) modify a large number of pages. With DTHREADS, each page modification triggers a segmentation violation, a system call to change memory protection, the creation of a private copy of the page, and a subsequent copy into the shared space on commit (see Section 6 for planned optimizations that may reduce this cost). We note that CoreDet also substantially degrades performance for these benchmarks, so much of this slowdown may be inherent to any deterministic runtime system.

4.4 Scalability

To measure the scalability cost of running DTHREADS, we ran our benchmark suite (excluding `canneal`) on the same machine with eight cores and again with two cores enabled. Whenever possible without source code modifications, the number of threads was matched to the number of CPUs enabled. We have found that DTHREADS scales at least as well as `pthreads` for 9 of 13 benchmarks, and scales as well or better than CoreDet for all but one benchmark where DTHREADS outperforms CoreDet by 2×. Detailed results of this experiment are presented in Figure 16 and discussed below.

`canneal` was excluded from the scalability experiment because this benchmark does more work when more threads are present, making the comparison between eight and two threads invalid. DTHREADS hurts scalability relative to `pthreads` for four of the benchmarks: `kmeans`, `word_count`, `dedup`, and `streamcluster` although only marginally in most cases. In all of these cases, DTHREADS scales better than CoreDet.

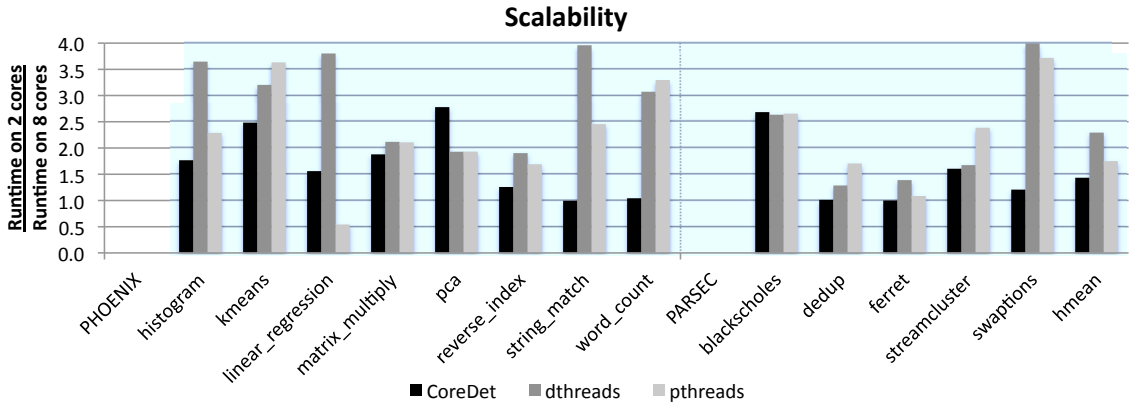

Figure 16. Speedup of eight cores versus two cores (higher is better). DTHREADS generally scales nearly as well or even better than `pthreads`, and almost always (with one exception) scales as well as or better than CoreDet.

DTHREADS is able to match the scalability of `pthreads` for three benchmarks: `matrix_multiply`, `pca`, and `blackscholes`. With DTHREADS, scalability actually *improves* over `pthreads` for 6 out of 13 benchmarks.

4.5 Performance Analysis

The data presented in Table 2 are obtained from the executions running on all 8 cores. Column 2 shows the percentage of time spent in the serial phase. In DTHREADS, all memory commits and actual synchronization operations are performed in the serial phase. The percentage of time spent in the serial phase thus can affect performance and scalability. Applications with higher overhead in DTHREADS often spend a higher percentage of time in the serial phase, primarily because they modify a large number of pages that are committed during that phase.

Column 3 shows the number of transactions in each application and Column 4 provides the average length of each transaction (ms). Every synchronization operation, including locks, conditional variable, barriers, and thread exits, demarcate transaction boundaries in DTHREADS. For example, `reverse_index`, `dedup`, `ferret` and `streamcluster` perform numerous transactions whose execution time is less than 1ms, imposing a performance penalty for these applications. Benchmarks with longer (or fewer) transactions run almost the same speed as or faster than `pthreads`, including `histogram` or `pca`. In DTHREADS, longer transactions amortize the overhead of memory protection and copying.

Column 5 provides more detail on the costs associated with memory updates (the number and total volume of dirtied pages). From the table, it becomes clear why `canneal` (the most notable outlier) runs much slower with DTHREADS than with `pthreads`. This benchmark updates over 3 million pages, leading to the creation of private copies, protection faults, and commits to the shared memory space. Copying alone is quite expensive: we found that copying one gigabyte of memory takes approximately 0.8 seconds when using `memcpy`, so for `canneal`, copying overhead alone accounts for at least 20 seconds of time spent in DTHREADS (out of 39s total).

Conclusion: Most benchmarks examined here contain either a small number or long running transactions, and modify a modest number of pages during execution. For these applications, DTHREADS is able to amortize its various overheads; by eliminating false sharing, it can even run faster than `pthreads`. However, for the few benchmarks that perform numerous short-lived transactions, or modify a large amount of pages, DTHREADS can exhibit substantial overhead.

Benchmark	Serial Phase (% of time)	Transactions Count	Transactions Time (ms)	Dirtied Pages
histogram	0	23	15.47	29
kmeans	0	3929	3.82	9466
linear_reg.	0	24	23.92	17
matrix_mult.	0	24	841.2	3945
pca	0	48	443	11471
reverseindex	17%	61009	1.04	451876
string_match	0	24	82	41
word_count	1%	90	26.5	5261
blackscholes	0	24	386.9	991
canneal	26.4%	1062	43	3606413
dedup	31%	45689	0.1	356589
ferret	12.3%	11282	1.49	147027
streamcluster	18.4%	130001	0.04	131992
swaptions	0	24	163	867

Table 2. Benchmark characteristics.

5. Related Work

The area of deterministic multithreading has seen considerable recent activity. Due to space limitations, we focus here on software-only, non language-based approaches.

Determinator is a microkernel-based operating system that enforces system-wide determinism [1]. Processes on Determinator run in isolation, and are able to communicate only at explicit synchronization points. Currently, Determinator is only a proof-of-concept system, and cannot be used for general multithreaded applications without modifications (e.g., it does not currently support condition variables). DTHREADS also isolates threads by running them in separate processes, but supports communication via its diffing and twinning mechanism. Also unlike Determinator, DTHREADS is a drop-in replacement for `pthreads` that does not require any special operating system support.

Isolator guarantees memory isolation during critical sections using code instrumentation, data replication, and virtual memory protection [27]. Data accesses that do not follow the application's locking discipline are not visible to threads running in critical sections. Unlike DTHREADS, Isolator does not enforce determinism, and cannot prevent all data races.

Benchmark	CoreDet	DTHREADS	pthreads	$\frac{\text{CoreDet}}{\text{pthreads}}$	$\frac{\text{DTHREADS}}{\text{pthreads}}$	Input
histogram	0.97	0.35	0.73	1.32×	0.48×	*large.bmp*
kmeans	68.41	15.02	13.16	5.20×	1.14×	*-d 3 -c 1000 -p 100000 -s 1000*
linear_regression	6.42	0.57	4.11	1.56×	0.14×	*key_file_500MB.txt*
matrix_multiply	31.68	19.28	19.32	1.63×	0.99×	*2000 2000*
pca	39.24	21.14	20.49	1.92×	1.03×	*-r 4000 -c 4000 -s 100*
reverse_index	7.85	6.53	2.06	3.81×	3.17×	*datafiles*
string_match	18.31	1.97	3.19	5.74×	0.62×	*key_file_500MB.txt*
word_count	17.17	2.37	2.17	7.91×	1.09×	*word_100MB.txt*
blackscholes	10.49	9.30	9.47	1.11×	0.98×	*8 in_1M.txt prices.txt*
canneal	14.74	39.82	10.41	1.42×	3.83×	*7 15000 2000 400000.nets 128*
dedup	3.38	5.39	1.45	2.33×	3.72×	*-c -p -f -t 2 -i media.dat output.txt*
ferret	21.89	16.95	7.02	3.11×	2.41×	*corel lsh queries 10 20 1 output.txt*
streamcluster	14.33	4.61	2.74	5.23×	1.68×	*10 20 128 16384 16384 1000 none output.txt 8*
swaptions	35.21	3.88	4.18	8.42×	0.93×	*-ns 128 -sm 50000 -nt 8*

Table 1. Benchmarks: execution time (in seconds) and input parameters.

Kendo guarantees a deterministic order of lock acquisitions on commodity hardware [25]. TERN [17] uses code instrumentation to memoize safe thread schedules for applications, and uses these memoized schedules for future runs on the same input. Both are only able to guarantee determinism for race-free programs, whereas DTHREADS guarantees determinism even in the presence of races.

CoreDet uses alternating parallel and serial phases, and a token-based global ordering that we adapt for DTHREADS [3]. Like DTHREADS, CoreDet guarantees deterministic execution in the presence of races, but at a much higher cost. All reads and writes to memory that cannot be proven via static analysis to remain thread-local must be instrumented. Additionally, CoreDet serializes *all* external library calls, except for specific variants provided by the CoreDet runtime. DTHREADS does not serialize library calls unless they perform synchronization operations, and only traps on the first write to a page during a transaction. Because of these differences, CoreDet runs as much as 8× slower than DTHREADS.

dOS [4] is an extension to CoreDet, and uses the same deterministic scheduling framework. dOS provides deterministic process groups (DPGs), which eliminate all internal non-determinism and control external non-determinism by recording and replaying interactions across DPG boundaries. Like Kendo, CoreDet and dOS use retired instruction counts as transaction boundaries. This approach can make it difficult for programmers to reason about or debug multithreaded programs even when they are deterministic: small changes in the code or inputs could unexpectedly trigger different thread interleavings. Because DTHREADS uses synchronization operations as boundaries for transactions, changing the code or input will not affect the schedule so long as the sequence of synchronization operations remains unchanged.

Grace prevents a wide range of concurrency errors, including deadlocks, race conditions, and atomicity violations, by imposing sequential semantics on multithreaded programs [6]. DTHREADS borrows Grace's threads-as-processes paradigm to provide memory isolation. Unlike DTHREADS, Grace provides stronger semantic guarantees, but is limited to fork-join parallelism and does not support inter-thread communication. When Grace detects that multiple threads have written to the same page, all but one of the threads will be rolled back, which can substantially degrade performance. DTHREADS does not rely on rollbacks but rather uses its deterministic commits to provide deterministic multithreaded execution with higher performance.

6. Future Work

We are investigating ways to enhance the performance and determinism guarantees of DTHREADS.

While DTHREADS ensures full *internal* determinism, it does not currently guarantee determinism if the application is sensitive to *external non-determinism*, such as the latency of network connections or time of day. We are examining the use of a shim layer to enforce deterministic delivery of such events in order to achieve the same kind of external determinism guarantees provided by dOS [4], although without changes to the underlying operating systems. We also plan to examine whether DTHREADS can be used directly in conjunction with dOS.

The key performance problem for DTHREADS is when an application modifies an extremely large number of pages. For those benchmarks, DTHREADS can incur high overheads. We plan to incorporate an ownership protocol to limit page copying and thus improve performance. In many cases, these pages are "owned" (read and modified) by a single thread. Tracking these pages would allow DTHREADS to avoid copying modified private pages to the shared memory space.

7. Conclusion

DTHREADS is a deterministic replacement for the `pthreads` library that supports general-purpose multithreaded applications. DTHREADS is straightforward to deploy, requiring no source code, and operates on commodity hardware. By converting threads into processes, DTHREADS leverages process isolation and virtual memory protection to track and isolate concurrent memory updates with low overhead. By committing these changes deterministically at natural synchronization points in the code, rather than at boundaries based on hardware performance counters, DTHREADS not only ensures full internal determinism—eliminating data races as well as deadlocks—but does so in a way that is portable and easy to understand. Its software architecture prevents false sharing, a notorious performance problem for multithreaded applications running on multiple, cache-coherent processors. The combination of these approaches enables DTHREADS to match or even exceed the performance of `pthreads` for the majority of the benchmarks examined here, making DTHREADS a safe and efficient alternative to `pthreads` for some applications.

8. Acknowledgements

The authors acknowledge the support of the Gigascale Systems Research Center, one of six research centers funded under the Focus Center Research Program (FCRP), a Semiconductor Research Corporation entity. This material is based upon work supported by Intel, Microsoft Research, and the National Science Foundation under CAREER Award CNS-0347339 and CNS-0615211. Any opin-

ions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient system-enforced deterministic parallelism. In *OSDI'10: Proceedings of the 9th Conference on Symposium on Opearting Systems Design & Implementation*, pages 193–206, Berkeley, CA, USA, 2010. USENIX Association.

[2] T. Ball, S. Burckhardt, J. de Halleux, M. Musuvathi, and S. Qadeer. Deconstructing concurrency heisenbugs. In *ICSE Companion*, pages 403–404. IEEE, 2009.

[3] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. 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 '10, pages 53–64, New York, NY, USA, 2010. ACM.

[4] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble. Deterministic process groups in dOS. In *OSDI'10: Proceedings of the 9th Conference on Symposium on Opearting Systems Design & Implementation*, pages 177–192, Berkeley, CA, USA, 2010. USENIX Association.

[5] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 117–128, Cambridge, MA, Nov. 2000.

[6] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: safe multithreaded programming for C/C++. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 81–96, New York, NY, USA, 2009. ACM.

[7] E. D. Berger and B. G. Zorn. DieHard: Probabilistic memory safety for unsafe languages. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 158–168, New York, NY, USA, 2006. ACM Press.

[8] E. D. Berger, B. G. Zorn, and K. S. McKinley. Composing high-performance memory allocators. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Snowbird, Utah, June 2001.

[9] C. Bienia and K. Li. Parsec 2.0: A new benchmark suite for chip-multiprocessors. In *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*, June 2009.

[10] 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 *Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 97–116, New York, NY, USA, 2009. ACM.

[11] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 1–11, New York, NY, USA, 1995. ACM Press.

[12] S. Burckhardt, A. Baldassin, and D. Leijen. Concurrent programming with revisions and isolation types. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, pages 691–707, New York, NY, USA, 2010. ACM.

[13] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In J. C. Hoe and V. S. Adve, editors, *ASPLOS*, ASPLOS '10, pages 167–178, New York, NY, USA, 2010. ACM.

[14] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and performance of munin. In *SOSP '91: Proceedings of the thirteenth ACM symposium on Operating systems principles*, pages 152–164, New York, NY, USA, 1991. ACM.

[15] R. H. Carver and K.-C. Tai. Replay and testing for concurrent programs. *IEEE Softw.*, 8:66–74, March 1991.

[16] J.-D. Choi and H. Srinivasan. Deterministic replay of java multithreaded applications. In *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, SPDT '98, pages 48–59, New York, NY, USA, 1998. ACM.

[17] H. Cui, J. Wu, C. Tsa, and J. Yang. Stable deterministic multithreaded through schedule memoization. In *OSDI'10: Proceedings of the 9th Conference on Symposium on Opearting Systems Design & Implementation*, pages 207–222, Berkeley, CA, USA, 2010. USENIX Association.

[18] J. W. Havender. Avoiding deadlock in multitasking systems. *IBM Systems Journal*, 7(2):74–84, 1968.

[19] ISO. Programming languages – c++. Available at http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3092.pdf.

[20] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. Treadmarks: distributed shared memory on standard workstations and operating systems. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, pages 10–10, Berkeley, CA, USA, 1994. USENIX Association.

[21] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[22] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Trans. Comput.*, 36:471–482, April 1987.

[23] C. E. McDowell and D. P. Helmbold. Debugging concurrent programs. *ACM Comput. Surv.*, 21(4):593–622, 1989.

[24] R. H. B. Netzer and B. P. Miller. What are race conditions?: Some issues and formalizations. *ACM Lett. Program. Lang. Syst.*, 1(1):74–88, 1992.

[25] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: efficient deterministic multithreading in software. In *ASPLOS '09: Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 97–108, New York, NY, USA, 2009. ACM.

[26] J. Pool, I. Sin, and D. Lie. Relaxed determinism: Making redundant execution on multiprocessors practical. In *Proceedings of the 11th Workshop on Hot Topics in Operating Systems (HotOS 2007)*, May 2007.

[27] S. Rajamani, G. Ramalingam, V. P. Ranganath, and K. Vaswani. ISOLATOR: dynamically ensuring isolation in comcurrent programs. In *ASPLOS '09: Proceeding of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 181–192, New York, NY, USA, 2009. ACM.

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

[29] D. J. Simpson and F. W. Burton. Space efficient execution of deterministic parallel programs. *IEEE Trans. Softw. Eng.*, 25:870–882, November 1999.

[30] W. Xiong, S. Park, J. Zhang, Y. Zhou, and Z. Ma. Ad hoc synchronization considered harmful. In *OSDI'10: Proceedings of the 9th Conference on Symposium on Opearting Systems Design & Implementation*, pages 163–176, Berkeley, CA, USA, 2010. USENIX Association.