DTHREADS: Efficient Deterministic Multithreading

Tongping Liu Charlie Curtsinger Emery D. Berger

Dept. of Computer Science
University of Massachusetts, Amherst
Amherst, MA 01003

Abstract

Multithreaded programming is notoriously difficult to get right. A key problem is non-determinism, which complicates debugging, testing, and reproducing errors. One way to simplify multithreaded programming is to enforce deterministic execution, but current deterministic systems for C/C++ are incomplete or impractical. These systems require program modification, do not ensure determinism in the presence of data races, do not work with general-purpose multithreaded programs, or run up to 8.4× slower than `pthreads`.

This paper presents DTHREADS, an efficient deterministic multithreading system for unmodified C/C++ applications that replaces the `pthreads` library. DTHREADS enforces determinism in the face of data races and deadlocks. DTHREADS works by exploding multithreaded applications into multiple processes, with private, copy-on-write mappings to shared memory. It uses standard virtual memory protection to track writes, and deterministically orders updates by each thread. By separating updates from different threads, DTHREADS has the additional benefit of eliminating false sharing. Experimental results show that DTHREADS substantially outperforms a state-of-the-art deterministic runtime system, and for a majority of the benchmarks evaluated here, matches and occasionally exceeds the performance of `pthreads`.

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 [16, 20, 21]. 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, 11].

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 [13] and reduce 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 by eliminating the need to track memory operations [14, 19], and it would enable the execution of multiple replicas of multithreaded applications for fault tolerance [4, 7, 10, 23].

Several recent software-only proposals aim at providing deterministic multithreading for C/C++ programs, but these suffer from a variety of disadvantages. Kendo ensures determinism of synchronization operations with low overhead, but does not guarantee determinism in the presence of data races [22]. Grace prevents all concurrency errors but is limited to fork-join programs. Although it can be efficient, it often requires 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 8.4× slower than `pthreads` (see Section 6) and generates thread interleavings at arbitrary points in the code, complicating program debugging and testing.

Contributions

This paper presents **DTHREADS**, a deterministic multithreading (DMT) runtime system with the following features:

- DTHREADS guarantees deterministic execution of multithreaded programs even in the presence of data races. Given the same sequence of inputs or OS events, a program using DTHREADS always produces the same output.
- DTHREADS is straightforward to deploy: it replaces the `pthreads` library, requiring no recompilation or code changes.
- DTHREADS is *robust* to changes in inputs, architectures, and code, enabling `printf` debugging of concurrent programs.
- DTHREADS eliminates cache-line *false sharing*, a notorious performance problem for multithreaded applications.
- DTHREADS is efficient. It nearly matches or even exceeds the performance of `pthreads` for the majority of the benchmarks examined here.

DTHREADS works by exploding multithreaded applications into multiple processes, with private, copy-on-write mappings to shared memory. It uses standard virtual memory protection to track writes, and deterministically orders updates by each thread. By separating updates from different threads, DTHREADS has the additional benefit of eliminating false sharing.

Our key insight is counterintuitive: the runtime costs and benefits of DTHREADS' mechanisms (processes, protection faults, copying and diffing, and false sharing elimination) balance out, for the majority of applications we evaluate here, the costs and benefits of `pthreads` (threads, no protection faults, and false sharing).

By committing changes only when needed, DTHREADS amortizes most of its costs. For example, because it only uses virtual memory protection to track the first write to a page, DTHREADS amortizes the cost of a fault over the length of a transaction.

DTHREADS provides deterministic execution while performing as well as or even better than `pthreads` for the majority of applications examined here, including much of the PARSEC benchmark suite (designed to be representative of next-generation shared-memory programs for chip-multiprocessors). DTHREADS isn't suitable for all applications: DTHREADS intercepts communication using the `pthreads` API, so programs using ad-hoc synchronization will not work with DTHREADS. Other application characteristics make it impossible for DTHREADS to amortize the costs of isolation and synchronization, resulting in poor performance. Despite these and other limitations, which we discuss in-depth in Section 7.2, DTHREADS still outperforms the previous state-of-the-art deterministic system by between 14% and 11.2× when evaluated using 14 parallel benchmarks.

DTHREADS marks a significant advance over the state of the art in deployability and performance, and provides promising evidence that fully deterministic multithreaded programming may be practical.

2. 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.

Grace prevents a wide range of concurrency errors, including deadlocks, race conditions, ordering and atomicity violations by imposing sequential semantics on threads with speculative execution [6]. DTHREADS borrows Grace's threads-as-processes paradigm to provide memory isolation, but differs from Grace in terms of semantics, generality, and performance.

Because it provides the effect of a serial execution of all threads, one by one, Grace rules out all interthread communication, including updates to shared memory, condition variables, and barriers. Grace supports only a restricted class of multithreaded programs: fork-join programs (limited to thread create and join). Unlike Grace, DTHREADS can run most general-purpose multithreaded programs while guaranteeing deterministic execution.

DTHREADS enables far higher performance than Grace for several reasons: It deterministically resolves conflicts, while Grace must rollback and re-execute threads that update any shared pages (requiring code modifications to avoid serialization); DTHREADS prevents false sharing while Grace exacerbates it; and DTHREADS imposes no overhead on reads.

CoreDet is a compiler and runtime system that represents the current state-of-the-art in deterministic, general-purpose software multithreading [3]. It uses alternating parallel and serial phases, and a token-based global ordering that we adapt for DTHREADS. Like DTHREADS, CoreDet guarantees deterministic execution in the presence of races, but with different mechanisms that impose a far higher cost: on average 3.5× slower and as much as 11.2× slower than DTHREADS (see Section 6). The CoreDet compiler instruments all reads and writes to memory that it cannot prove by static analysis to be thread-local. CoreDet also serializes *all* external library calls, except for specific variants provided by the CoreDet runtime.

CoreDet and DTHREADS also differ semantically. DTHREADS only allows interleavings at synchronization points, but CoreDet relies on the count of instructions retired to form quanta. This approach makes it impossible to understand a program's behavior by examining the source code—the only way to know what a program does in CoreDet (or dOS and Kendo, which rely on the same mechanism) is to execute it on the target machine. This instruction-based commit schedule is also brittle: even small changes to the input or program can cause a program to behave differently, effectively ruling out `printf` debugging. DTHREADS uses synchronization operations as boundaries for transactions, so changing the code or input does not affect the schedule as long as the sequence of synchronization operations remains unchanged. We call this more stable form of determinism *robust determinism*.

dOS [4] is an extension to CoreDet that 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. dOS is orthogonal and complementary to DTHREADS, and in principle, the two could be combined.

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. For programs that use condition variables, Determinator emulates a legacy thread API with quantum-based determinism similar to CoreDet. This legacy support suffers from the same performance and robustness problems as CoreDet.

Like Determinator, DTHREADS isolates threads by running them in separate processes, but natively supports all `pthreads` communication primitives. DTHREADS is a drop-in replacement for `pthreads` that needs no special operating system support.

Finally, some recent proposals provide limited determinism. Kendo guarantees a deterministic order of lock acquisitions on commodity hardware ("weak determinism"); Kendo only enforces full ("strong") determinism for race-free programs [22]. TERN [15] uses code instrumentation to memoize safe thread schedules for applications, and uses these memoized schedules for future runs on the same input. Unlike these systems, DTHREADS guarantees full determinism even in the presence of races.

3. DTHREADS Overview

We begin our discussion of how DTHREADS works with an example execution of a simple, racy multithreaded program, and explain at a high level how DTHREADS enforces deterministic execution.

Figure 1 shows a simple multithreaded program that, because of data races, non-deterministically produces the outputs "1,0," "0,1" and "1,1." With `pthreads`, the order in which these modifications occur can change from run to run, resulting in non-deterministic output.

With DTHREADS, however, this program *always* produces the same output, ("1,1"), which corresponds to exactly one possible thread interleaving. DTHREADS ensures determinism using the following key approaches, illustrated in Figure 2:

Isolated memory access: In DTHREADS, threads are implemented using separate processes with private and shared views of memory, an idea introduced by Grace [6]. Because processes have separate address spaces, they are a convenient mechanism to isolate memory accesses between threads. DTHREADS uses this isolation to control the visibility of updates to shared memory, so each "thread" operates independently until it reaches a synchronization point (see below). Section 4.1 discusses the implementation of this mechanism in depth.

Deterministic memory commit: Multithreaded programs often use shared memory for communication, so DTHREADS must propagate one thread's writes to all other threads. To ensure deterministic execution, these updates must be applied at deterministic times, and in a deterministic order.

DTHREADS updates shared state in sequence at synchronization points. These points include thread creation and exit; mutex lock and unlock; condition variable wait and signal; posix sigwait and signal; and barrier waits. Between synchronization points, all

```
int a = b = 0;
main() {
  pthread_create(&p1, NULL, t1, NULL);
  pthread_create(&p2, NULL, t2, NULL);
  pthread_join(&p1, NULL);
  pthread_join(&p2, NULL);
  printf ("%d,%d\n", a, b);
}
```

```
void * t1 (void *) {
  if (b == 0) {
    a = 1;
  }
  return NULL;
}
```

```
void * t2 (void *) {
  if (a == 0) {
    b = 1;
  }
  return NULL;
}
```

Figure 1. A simple multithreaded program with data races on `a` and `b`. With `pthreads`, the output is non-deterministic, but DTHREADS guarantees the same output on every execution.

code effectively executes within an atomic *transaction*. This combination of memory isolation between synchronization points with a deterministic commit protocol guarantees deterministic execution even in the presence of data races.

Deterministic synchronization: DTHREADS supports the full array of `pthreads` synchronization primitives. Because 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 guarantee a deterministic ordering. Details on the DTHREADS implementations of these primitives are given in Section 4.3.

Twinning and diffing: Before committing updates, DTHREADS first compares each modified page to a "twin" (copy) of the original shared page, and then writes only the modified bytes (diffs) into shared state (see Section 5 for optimizations that avoid copying and diffing). This algorithm is adapted from the distributed shared memory systems TreadMarks and Munin [12, 17]. The order in which threads write their updates to shared state is enforced by a single global token passed from thread to thread; see Section 4.2 for full details.

Fixing the data race example

Returning to the example program in Figure 1, we can now see how DTHREADS' memory isolation and a deterministic commit order ensure deterministic output. DTHREADS effectively isolates each thread from each other until it completes, and then orders updates by thread creation time using a deterministic last-writer-wins protocol.

At the start of execution, thread 1 and thread 2 have 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 made visible to the other until thread exit, both threads' checks on line 2 will be true. Thread 1 sets the value of a to 1, and thread 2 sets 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 prints "1, 1" on every execution.

This determinism not only enables record-and-replay and replicated execution, but also effectively converts Heisenbugs into "Bohr" bugs, making them reproducible. In addition, DTHREADS optionally reports any conflicting updates due to racy writes, further simplifying debugging.

4. DTHREADS Architecture

This section describes DTHREADS' key algorithms—memory isolation, deterministic (diff-based) memory commit, deterministic synchronization, and deterministic memory allocation—as well as other implementation details.

4.1 Isolated Memory Access

To achieve deterministic memory access, DTHREADS isolates memory accesses among different threads between commit points, and commits the updates of each thread deterministically.

DTHREADS achieves cross-thread memory isolation by replacing threads with 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.

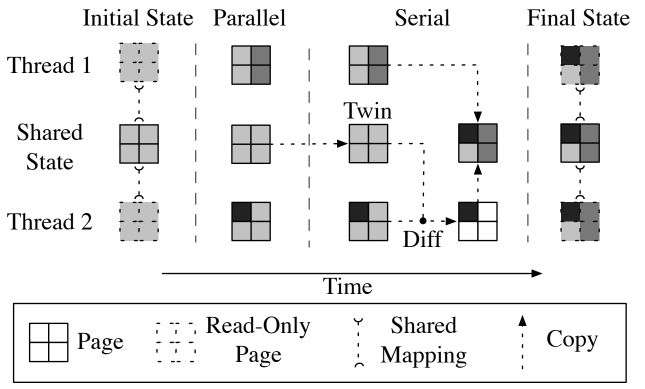

Figure 2. An overview of DTHREADS execution.

4.1.1 Thread Creation

DTHREADS replaces the `pthread_create()` function with the `clone` system call provided by Linux. To create processes that have disjoint address spaces but share the same file descriptor table, DTHREADS uses the `CLONE_FILES` flag. DTHREADS shims the `getpid()` function to return a single, globally-shared identifier.

4.1.2 Deterministic Thread Index

POSIX does not guarantee deterministic process or thread identifiers; that is, the value of a process id or thread id is not deterministic. To avoid exposing this non-determinism to threads running as processes, DTHREADS shims `pthread_self()` to return an internal thread index. The 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.

4.1.3 Shared Memory

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 7).

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 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 deterministic thread index to find the appropriate sub-heap.

4.2 Deterministic Memory Commit

Figure 3 illustrates the progression of parallel and serial phases. To guarantee determinism, DTHREADS isolates memory accesses in the parallel phase. These accesses work on private copies of memory; that is, updates are not shared between threads during 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.

4.2.1 Fence and Token

The boundary between the parallel and serial phases is the internal fence. We implement this fence with a custom barrier, because the standard `pthreads` barrier does not support a dynamic thread count (see Section 4.3).

Threads wait at the internal fence until all threads from the previous fence have departed. Waiting threads must block until the departure phase. If the thread is the last to enter the fence, it initiates the departure phase and wakes all waiting threads. 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.

To reduce overhead, whenever the number of running threads is less than or equal to the number of cores, waiting threads block by spinning rather than by invoking relatively expensive cross-process `pthreads` mutexes. When the number of threads exceeds the number of cores, DTHREADS falls back to using `pthreads` mutexes.

A key mechanism used by DTHREADS is its global token. 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. The `waitToken` function first waits at the internal fence and then waits to acquire the global token before entering serial mode. The `putToken` function passes the token to the next waiting thread.

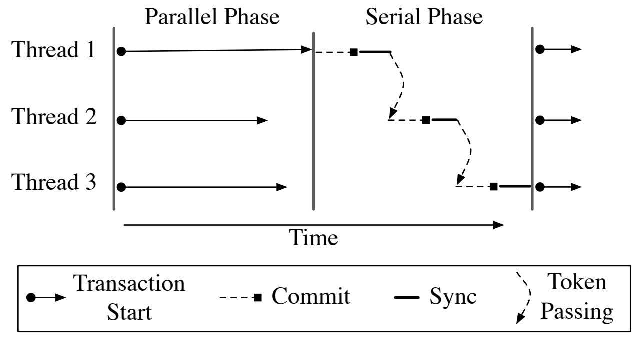

Figure 3. An overview of DTHREADS phases. Program execution with DTHREADS alternates between parallel and serial phases.

To guarantee determinism (see Figure 3), 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.

4.2.2 Commit Protocol

Figure 2 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.

At the start of every transaction (that is, right after a synchronization point), DTHREADS starts by write-protecting all previously-written pages. The old working copies of these pages are then discarded, and mappings are then updated to reference the shared state.

Just before every synchronization point, DTHREADS first waits for the global token (see below), and then commits all changes from the current transaction to the shared pages in order. DTHREADS maintains one "twin" page (a snapshot of the original) for every modified page with more than one writer. 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 working copy can be copied directly to the shared state. 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.

Once changes have been committed, the number of writers to the page is decremented and the shared page's version number is incremented. If there are no writers left to commit, the twin page is freed.

4.3 Deterministic Synchronization

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

4.3.1 Locks

DTHREADS uses a single global token to guarantee ordering and atomicity during the serial phase. When acquiring a lock, threads must first wait for the global token. Once a thread has the token it can attempt to acquire the lock. If the lock is currently held, the thread must pass the token and wait until the next serial phase to acquire the lock. It is possible for a program run with DTHREADS to deadlock, but only for programs that can also deadlock with `pthreads`.

Lock acquisition proceeds as follows. First, DTHREADS checks to see if the current thread is already holding any locks. If not, the thread first waits for the token, commits changes to shared state by calling `atomicEnd`, and begins a new atomic section. Finally, the thread increments the number of locks it is currently holding. The lock count ensures that a thread does not pass the token on until it has released all of the locks it acquired in the serial phase.

`pthread_mutex_unlock`'s implementation is similar. First, the thread decrements its lock count. 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. Finally, the thread waits on the internal fence until the start of the next round's parallel phase. If other locks are still held, the lock count is just decreased and the running thread continues execution with the global token.

4.3.2 Condition Variables

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

When a thread calls `pthread_cond_wait`, it first acquires the token and commits local modifications. It then removes itself from the token queue, because threads waiting on a condition variable do not participate in the serial phase until they are awakened. 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. While threads are waiting on DTHREADS condition variables, they are suspended on a `pthreads` condition variable. When a thread is awakened (signalled), it busy-waits on the token before beginning the next transaction. Threads must acquire the token before proceeding because the condition variable wait function must be called within a mutex's critical section.

In the DTHREADS implementation of `pthread_cond_signal`, the calling thread first waits for the token, and then commits any local modifications. If no threads are waiting on the condition variable, this function returns immediately. 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. This thread is then marked as ready and woken up from the real condition variable, and the calling thread begins another transaction.

To impose an order on signal wakeup, DTHREADS signals actually call `pthread_cond_broadcast` to wake all waiting threads, but then marks only the logically next one as ready. The threads not marked as ready will wait on the condition variable again.

4.3.3 Barriers

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

In `pthread_barrier_wait`, the calling thread first waits for the token to commit any local modifications. 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, increases the live thread count, and passes the token to the first thread in the barrier queue. Otherwise, DTHREADS removes the current thread from the token queue, places it on the barrier queue, and releases token. Finally, the thread waits on the actual `pthreads` barrier.

4.3.4 Thread Creation and Exit

To guarantee determinism, thread creation and exit are performed in the serial phase. Newly-created threads are added to the token queue immediately after the parent thread. Creating a thread does not 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 child thread.

When creating a thread, the parent first waits for the token. It then creates a new process with shared file descriptors but a distinct address space using the `clone` system call. The newly created child obtains the global thread index, places itself in the token queue, and notifies the parent that the child has registered itself in the active list. The child thread then waits for the next parallel phase before proceeding.

Similarly, DTHREADS' `pthread_exit` first waits for the token and then commits any local modifications to memory. It then removes itself from the token queue and decreases the number of threads required to proceed to the next phase. Finally, the thread passes its token to the next thread in the token queue and exits.

4.3.5 Thread Cancellation

DTHREADS implements thread cancellation in the serial phase. A thread can only invoke `pthread_cancel` while holding the token. If the thread being cancelled is waiting on a condition variable or barrier, it is removed from the queue. Finally, to cancel the corresponding thread, DTHREADS kills the target process with a call to `kill(tid, SIGKILL)`.

4.4 Deterministic Memory Allocation

Programs sometimes rely on the addresses of objects returned by the memory allocator intentionally (for example, by hashing objects based on their addresses), or accidentally. A program with a memory error like a buffer overflow will yield different results for different memory layouts.

This reliance on memory addresses can undermine other efforts to provide determinism. For example, CoreDet is unable to fully enforce determinism because it relies on the Hoard scalable memory allocator [5]. Hoard was not designed to provide determinism and several of its mechanisms, thread id based hashing and non-deterministic assignment of memory to threads, lead to non-deterministic execution in CoreDet for the `canneal` benchmark.

To preserve determinism in the face of intentional or inadvertent reliance on memory addresses, we designed the DTHREADS memory allocator to be fully deterministic. DTHREADS assigns subheaps to each thread based on its thread index (deterministically assigned; see Section 4.1.2). In addition to guaranteeing the same mapping of threads to subheaps on repeated executions, DTHREADS allocates superblocks (large chunks of memory) deterministically by acquiring a lock (and the global token) on each superblock allocation. Thus, threads always use the same subheaps, and these subheaps always contain the same superblocks on each execution. The remainder of the memory allocator is entirely deterministic. The superblocks themselves are allocated via `mmap`: while DTHREADS could use a fixed address mapping for the heap, we currently simply disable ASLR to provide deterministic `mmap` calls. If a program does not use the absolute address of any heap object, DTHREADS can guarantee determinism even with ASLR enabled. Hash functions and lock-free algorithms frequently use absolute addresses, and any deterministic multithreading system must disable ASLR to provide deterministic results for these cases.

4.5 OS Support

DTHREADS provides shims for a number of system calls both for correctness and determinism (although it does not enforce deterministic arrival of I/O events; see Section 7).

System calls that write to or read from buffers on the heap (such as `read` and `write`) will fail if the buffers contain protected pages. DTHREADS intercepts these calls and touches each page passed in as an argument to trigger the copy-on-write operation before issuing the real system call. DTHREADS conservatively marks all of these pages as modified so that any updates made by the system will be committed properly.

DTHREADS also intercepts other system calls that affect program execution. For example, when a thread calls `sigwait`, DTHREADS behaves much like it does for condition variables. It removes the calling thread from the token queue before issuing the system call, and after being awakened the thread must re-insert itself into the token queue and wait for the token before proceeding.

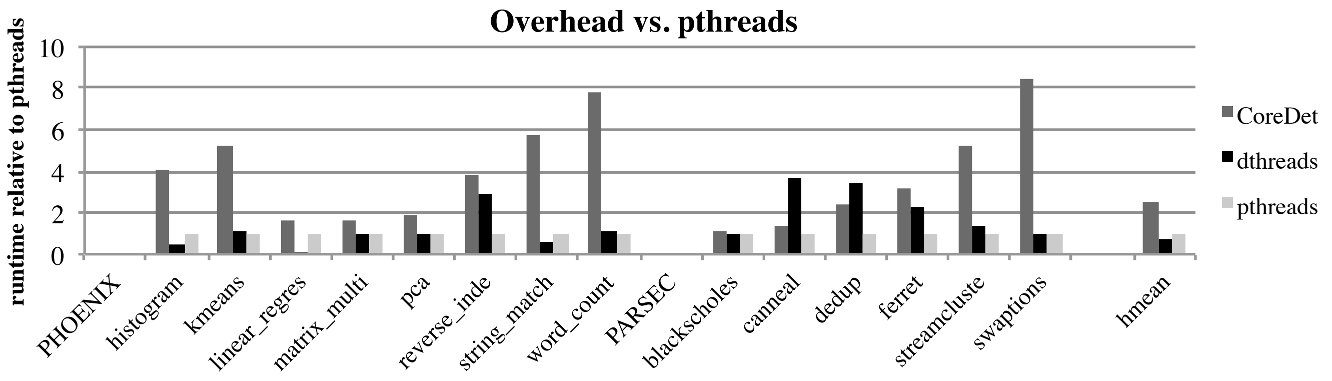

Figure 4. Normalized execution time with respect to `pthreads` (lower is better). For 9 of the 14 benchmarks, DTHREADS runs nearly as fast or faster than `pthreads`, while providing deterministic behavior.

5. Optimizations

DTHREADS employs a number of optimizations that improve its performance.

Lazy commit: DTHREADS reduces copying overhead and the time spent in the serial phase by *lazily* committing pages. When only one thread has ever modified a page, DTHREADS considers that thread to be the page's owner. An owned page is committed to shared state only when another thread attempts to read or write this page, or when the owning thread attempts to modify it in a later phase. DTHREADS tracks reads with page protection and signals the owning thread to commit pages on demand. To reduce the number of read faults, pages holding global variables (which we expect to be shared) and any pages in the heap that have ever had multiple writers are all considered unowned and are not read-protected.

Lazy twin creation and diff elimination: To further reduce copying and memory overhead, a twin page is only created when a page has multiple writers during the same transaction. In the commit phase, a single writer can directly copy its working copy to shared state without performing a diff. DTHREADS does this by comparing the local version number to the global page version number for each dirtied page. At commit time, DTHREADS directly copies its working copy for each page whenever its local version number equals its global version number. This optimization saves the cost of a twin page allocation, a page copy, and a diff in the common case where just one thread is the sole writer of a page.

Single-threaded execution: Whenever only one thread is running, DTHREADS stops using memory protection and treats certain synchronization operations (locks and barriers) as no-ops. In addition, when all other threads are waiting on condition variables, DTHREADS does not commit local changes to the shared mapping or discard private dirty pages. Updates are only committed when the thread issues a signal or broadcast call, which wakes up at least one thread and thus requires that all updates be committed.

Lock ownership: DTHREADS uses lock ownership to avoid unnecessary waiting when threads are using distinct locks. Initially, all locks are unowned. Any thread that attempts to acquire a lock that it does not own must wait until the serial phase to do so. If multiple threads attempt to acquire the same lock, this lock is marked as shared. If only one thread attempts to acquire the lock, this thread takes ownership of the lock and can acquire and release it during the parallel phase.

Lock ownership can result in starvation if one thread continues to re-acquire an owned lock without entering the serial phase. To avoid this, each lock has a maximum number of times it can be acquired during a parallel phase before a serial phase is required.

Parallelization: DTHREADS attempts to expose as much parallelism as possible in the runtime system itself. One optimization takes place at the start of trasactions, where DTHREADS performs a variety of cleanup tasks. These include 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 3), this can hurt performance for some benchmarks.

Since these operations do not affect other the behavior of other threads, most of this 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, this cleanup occurs at the end of each transaction.

6. 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 at 2.33GHZ with a 4MB L2 cache. The operating system is CentOS 5.5 (unmodified), running with Linux kernel version 2.6.18-194.17.1.el5. The glibc version is 2.5. Benchmarks were built as 32-bit executables with version 2.6 of the LLVM compiler.

6.1 Methodology

We evaluate the performance and scalability of DTHREADS versus CoreDet and `pthreads` across the PARSEC [9] and Phoenix [24] 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 [18], all benchmarks are compiled with the LLVM compiler at the "-O3" optimization level [18]. 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 is 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 table (in bytes), the quantum size (in number of instructions retired), and the choice between full and reduced serial mode. We performed an extensive search of the parameter space to find the one that

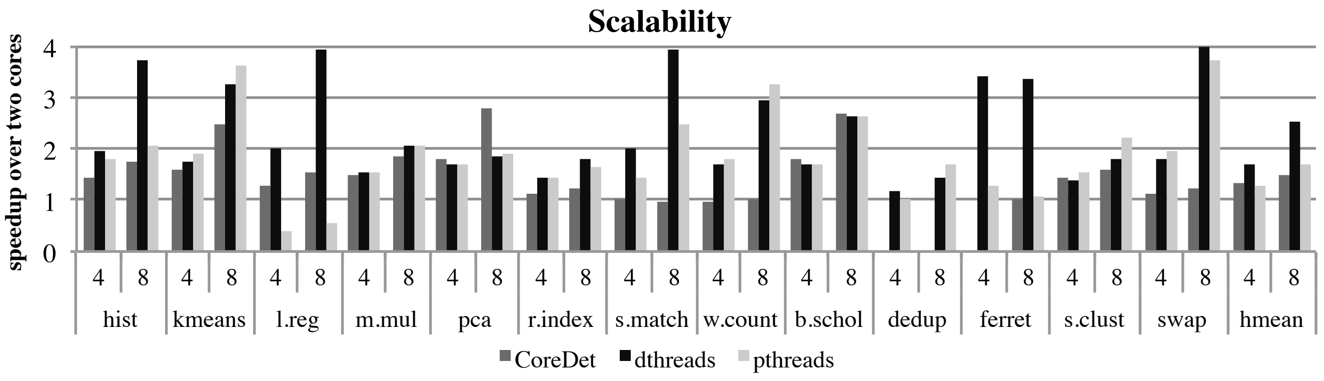

Figure 5. Speedup with four and eight cores relative to two cores (higher is better). DTHREADS generally scales nearly as well or better than `pthreads` and almost always as well or better than CoreDet. CoreDet was unable to run `dedup` with two cores and `ferret` with four cores, so some scalability numbers are missing.

yielded the lowest average normalized runtimes (using six possible granularities and eight possible quanta for each benchmark), and found that the best settings on our system were 64-byte granularity and a quantum size of 100,000 instructions, in full serial mode.

Unsupported Benchmarks: We were unable to evaluate DTHREADS on seven of the PARSEC benchmarks: `vips` and `raytrace` would not build as 32-bit executables; `bodytrack`, `facesim`, and `x264` depend on sharing of stack variables; `fluidanimate` uses ad-hoc synchronization, so it will not run without modifications; and `freqmine` does not use `pthreads`.

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`).

6.2 Determinism

We first experimentally verify DTHREADS' ability to ensure determinism by executing the *racey* determinism tester [22]. This stress test is extremely sensitive to memory-level non-determinism. DTHREADS reports the same results for 2,000 runs. We also compared the schedules and outputs of all benchmarks used to ensure that every execution is identical.

6.3 Performance

We next compare the performance of DTHREADS to CoreDet and `pthreads`. Figure 4 presents these results graphically (normalized to `pthreads`).

DTHREADS outperforms CoreDet on 12 out of 14 benchmarks (between 14% and 11.2× faster); for 8 benchmarks, DTHREADS matches or outperforms `pthreads`. DTHREADS results in good performance for several reasons:

- Process invocation is only slightly more expensive than thread creation. This is because both rely on the `clone` system call. Copy-on-write semantics allow process creation without expensive copying.
- Context switches between processes are more expensive than for threads because of the required TLB shootdown. The number of context switches was minimized by running on a quiescent system with the number of threads matched to the number of cores whenever possible.
- DTHREADS incurs no read overhead and very low write overhead (one page fault per written page), but commits are expensive. Most of our benchmarks (and many real applications) result in small, infrequent commits.
- DTHREADS isolates updates in separate processes, which can improve performance by eliminating false sharing. Because threads actually execute in different address spaces, there is no coherence traffice between synchronization points.

By eliminating catastrophic false sharing, DTHREADS dramatically improves the performance of the `linear_regression` benchmark, running 7× faster than `pthreads` and 11.2× faster than CoreDet. The `string_match` benchmark exhibits a similar, if less dramatic, false sharing problem: with DTHREADS, it runs almost 40% faster than `pthreads` and 9.2× faster than CoreDet. Two benchmarks also run faster with DTHREADS than with `pthreads` (`histogram`, 2× and `swaptions`, 5%; respectively 8.5× and 8.9× 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 over the course of one run. While Linux processes are relatively lightweight, creating and tearing down a process is still more expensive than the same operation for threads, accounting for a 5% performance degradation of DTHREADS over `pthreads` (though it runs 4.9× 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. We note that CoreDet also substantially degrades performance for these benchmarks, so much of this slowdown may be inherent to any deterministic runtime system.

6.4 Scalability

To measure the scalability cost of running DTHREADS, we ran our benchmark suite (excluding `canneal`) on the same machine with eight cores, four corse, and just 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 3.5×. Detailed results of this experiment are presented in Figure 5 and discussed below.

The `canneal` benchmark was excluded from the scalability experiment because it matches the workload to the number of threads, making the comparison between different numbers of threads invalid. DTHREADS hurts scalability relative to `pthreads` for the `kmeans`, `word_count`, `dedup`, and `streamcluster` benchmarks, although only marginally in most cases. In all of these cases, DTHREADS scales 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. This is because DTHREADS prevents false sharing, avoiding unnecessary cache invalidations that normally hurt scalability.

6.5 Performance Analysis

6.5.1 Benchmark Characteristics

The data presented in Table 1 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, condition variables, 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 a total execution time of 39 seconds).

Conclusion: For the few benchmarks that perform large numbers of short-lived transactions, modify a large number of pages per-transaction, or both, DTHREADS can result in substantial overhead. Most benchmarks examined here run fewer, longer-running transactions with a modest number of modified pages. For these applications, overhead is amortized. With the side-effect of eliminating false sharing, DTHREADS can sometimes even outperform `pthreads`.

6.5.2 Performance Impact Analysis

To understand the performance impact of DTHREADS, we re-ran the benchmark suite on two individual components of DTHREADS: deterministic synchronization and memory protection.

	Serial	Transactions		Dirtied
Benchmark	**(% time)**	**Count**	**Time (ms)**	**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 1. Benchmark characteristics.

Sync-only: This configuration enforces only a deterministic synchronization order. Threads have direct access to shared memory with no isolation. Overhead from this component is largely due to load imbalance from the deterministic scheduler.

Prot-only: This configuration runs threads in isolation, with commits at synchronization points. The synchronization and commit order is not controlled by DTHREADS. This configuration eliminates false sharing, but also introduces isolation and commit overhead. The lazy twin creation and single-threaded execution optimizations are disabled here because they are unsafe without deterministic synchronization.

The results of this experiment are presented in Figure 6 and discussed below.

- The `reverse_index`, `dedup`, and `ferret` benchmarks show significant load imbalance with the *sync-only* configuration. Additionally, these benchmarks have high overhead from the *prot-only* configuration because of a large number of transactions.
- Both `string_match` and `histogram` run faster with the *sync-only* configuration. The reason for this is not obvious, but may be due to the per-thread allocator.
- Memory isolation in the *prot-only* configuration eliminates false sharing, which resulted in speedups for `histogram`, `linear_regression`, and `swaptions`.
- Normally, the performance of DTHREADS is not better than the *prot-only* configuration. However, both `ferret` and `canneal` run faster with deterministic synchronization enabled. Both benchmarks benefit from optimizations described in Section 5 that are only safe with deterministic synchronization enabled. `ferret` benefits from the single threaded execution optimization, and `canneal` sees performance gains due to the shared twin page optimization.

7. Discussion

All DMT systems must impose an order on updates to shared memory and synchronization operations. The mechanism used to isolate updates affects the limitations and performance of the system. DTHREADS represents a new point in the design space for DMT systems with some inherent advantages and limitations which we discuss below.

7.1 Design Tradeoffs

CoreDet and DTHREADS both use a combination of parallel and serial phases to execute programs deterministically. These two sys-

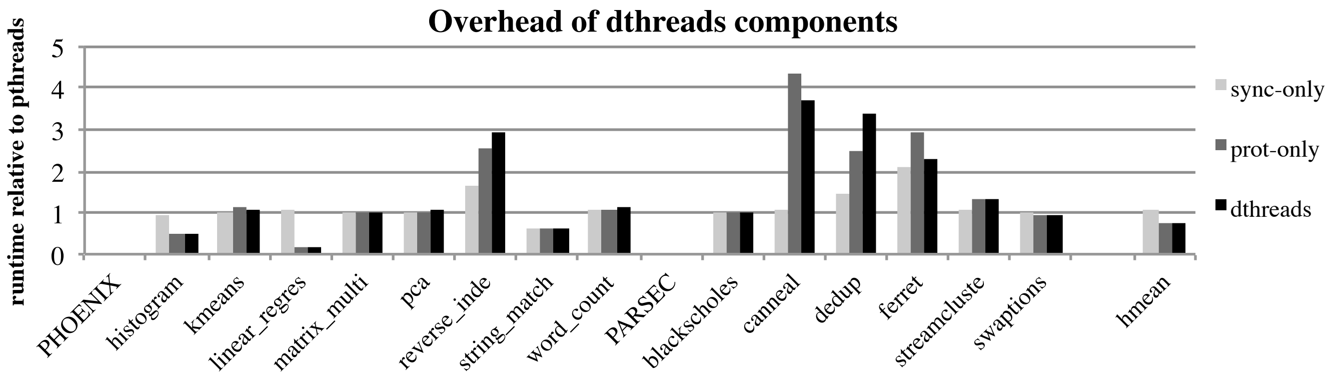

Figure 6. Normalized execution time with respect to `pthreads` (lower is better) for three configurations. The *sync-only* and *prot-only* configurations are described in Section 6.5.2.

tems take different approaches to isolation during parallel execution, as well as the transitions between phases:

Memory isolation: CoreDet orders updates to shared memory by instrumenting all memory accesses that could reference shared data. Synchronization operations and updates to shared memory must be performed in a serial phase. This approach results in high instrumentation overhead during parallel execution, but incurs no additional overhead when exposing updates to shared state.

DTHREADS takes an alternate approach: updates to shared state proceed at full speed, but are isolated using hardware-supported virtual memory. When a serial phase is reached, these updates must be exposed in a deterministic order with the twinning and diffing method described in Section 4.2.2.

A pleasant side-effect of this approach is the elimination of false sharing. Because threads work in separate address spaces, there is no need to keep caches coherent between threads during the parallel phase. For some programs this results in a performance improvement as large as 7× when compared to `pthreads`.

Phases: CoreDet uses a quantum-based scheduler to execute the serial phase. After the specified number of instructions is executed, the scheduler transitions to the serial phase. This approach bounds the waiting time for any threads that are blocked until a serial phase. One drawback of this approach is that transitions to the serial phase do not correspond to static program points. Any code changes (and most inputs) will result in a new, previously-untested schedule.

Transitions between phases are static in DTHREADS. Any synchronization operation will result in a transition to a serial phase, and parallel execution will resume once all threads have executed their critical sections. This makes DTHREADS susceptible to delays due to load imbalance between threads but results in more robust determinism. With DTHREADS, only the order of synchronization operations affects the schedule. For most programs this means that different inputs, and even many code changes, will not change the schedule produced by DTHREADS.

7.2 Limitations

External non-determinism: DTHREADS provides only internal determinism. It does not guarantee determinism when a program's behavior depends on external events, such as system time or the arrival order of network packets. The dOS framework is a proposed OS mechanism that provides system-level determinism [4]. dOS provides Deterministic Process Groups and a deterministic replay shim for external events, but uses CoreDet to make each individual process deterministic. DTHREADS could be used instead CoreDet within the dOS system, which would add support for controlling external non-determinism.

Unsupported programs: DTHREADS supports programs that use the `pthreads` library, but does not support programs that bypass it by rolling their own *ad hoc* synchronization operations. While *ad hoc* synchronization is common, 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 [25].

DTHREADS does not write-share the stack across threads, so any updates to stack variables are only locally visible. While sharing of stack variables is supported by `pthreads`, this practice is error-prone and relatively uncommon. Support for shared stack variables could be added to DTHREADS by handling stack memory like the heap and globals, but this would require additional optimizations to avoid poor performance in the common case where stack memory is unshared.

Memory consumption: DTHREADS creates private, per-process copies of modified pages between commits. Because of this, it can increase a program's memory footprint by the number of modified pages between synchronization operations. This increased footprint does not pose 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.

Memory consistency: DTHREADS provides a form of release consistency for parallel programs, where updates are exposed at static program points. CoreDet's DMP-B mode also uses release consistency, but the update points depend on when the quantum counter reaches zero. To the best of our knowledge, DTHREADS cannot produce an output that is not possible with `pthreads`, although for some cases it will result in unexpected output. When run with DTHREADS, the example in Figure 1 will always produce the output "1,1." This ouptut is also possible with `pthreads`, but is much less likely (occurring in just 0.01% of one million runs) than "1,0" (99.43%) or "0,1" (0.56%). Of course, the same unexpected output will be produced on every run with DTHREADS, making it easier for developers to track down the source of the problem than with `pthreads`.

8. Conclusion

DTHREADS is a deterministic replacement for the `pthreads` library that supports general-purpose multithreaded applications. It is straightforward to deploy: DTHREADS resuires 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. Changes are committed 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 many applications.

9. Acknowledgements

The authors thank Robert Grimm, Sam Guyer, Shan Lu, Tom Bergan, Daan Leijen, Dan Grossman, Yannis Smaragdakis, the anonymous reviewers, and our shepherd Steven Hand for their invaluable feedback and suggestions which helped improve this paper. We 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 CCF-1012195 and CCF-0910883. Any opinions, 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] 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.

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

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

[17] 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.

[18] 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.

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

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

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.