DThreads

DTHREADS: Efficient Deterministic Multithreading

Tongping Liu Charlie Curtsinger Emery D. Berger

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

Contributions - DThreads

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 `pthread` 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 exceed 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.

Why do multiple processes matter?

What is copy-on-write?

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 Commits

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 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.

Example – Race Condition

```
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.

Discuss – Why is this a race condition?

Barriers / Forced Ordering for Commits

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.

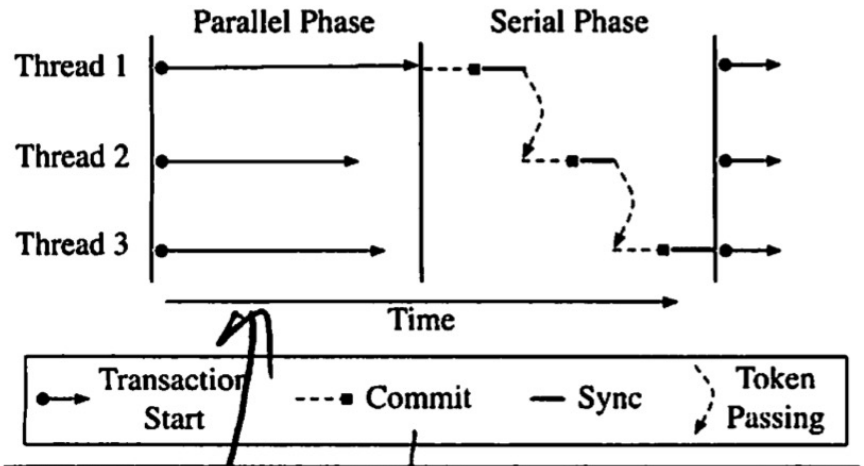

Figure 3. An overview of DTHREADS phases. Program execution with DTHREADS alternates between parallel and serial phases.

VM Trick – set the memory access to read only, catch the ”fault” (write) and then switch from parallel to serial

pthread_barrier

```
NAME
     pthread_barrier, pthread_barrier_init, pthread_barrier_destroy,
     pthread_barrier_wait – barrier interface

LIBRARY
     POSIX Threads Library (libpthread, -lpthread)

SYNOPSIS
     #include <pthread.h>

     int
     pthread_barrier_init(pthread_barrier_t * restrict barrier,
         const pthread_barrierattr_t * restrict attr, unsigned int count);

     int
     pthread_barrier_destroy(pthread_barrier_t *barrier);

     int
     pthread_barrier_wait(pthread_barrier_t *barrier);

DESCRIPTION
     The pthread_barrier_init() function creates a new barrier with attributes
     attr and count.  The count parameter indicates the number of threads
     which will participate in the barrier.  The pthread_barrierattr_init(3)
     function may be used to specify the attributes supplied in attr.  If attr
     is NULL, the default attributes are used.  Barriers are most commonly
     used in the decomposition of parallel loops.

     The pthread_barrier_destroy() function causes the resources allocated to
     barrier to be released.  No threads should be blocked on barrier.

     The pthread_barrier_wait() function causes the current thread to wait on
     the barrier specified.  Once as many threads as specified by the count
     parameter to the corresponding pthread_barrier_init() call have called
     pthread_barrier_wait(), all threads will wake up, return from their
     respective pthread_barrier_wait() calls and continue execution.
```

Initialize to the value being the number of threads that need to synchronize

No thread can proceed until the requested count of threads get to the barrier (wait)

Barriers / Forced Ordering for Commits

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.

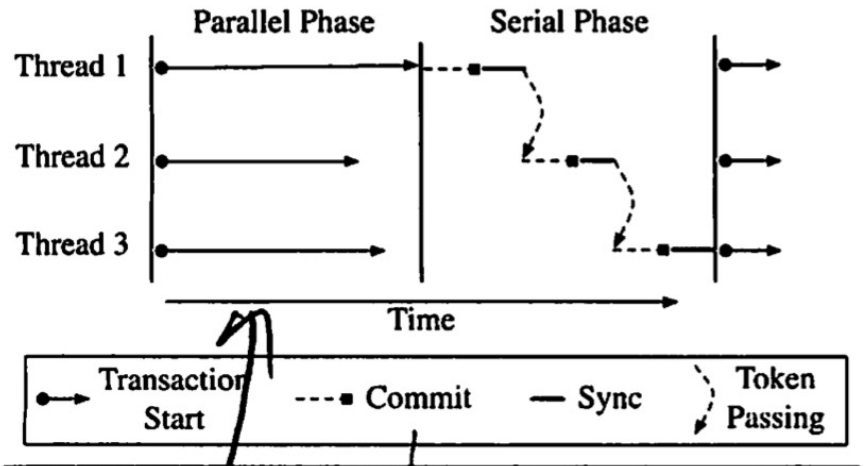

Figure 3. An overview of DTHREADS phases. Program execution with DTHREADS alternates between parallel and serial phases.

VM Trick – set the memory access to read only, catch the ”fault” (write) and then switch from parallel to serial

Global Synchronization

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`.

What Is the Python Global Interpreter Lock (GIL)?

What Problem Did the GIL Solve for Python?

Python uses reference counting for memory management. It means that objects created in Python have a reference count variable that keeps track of the number of references that point to the object. When this count reaches zero, the memory occupied by the object is released.

Discuss: Can we have a race condition if we have a GIL?

Better Performance?

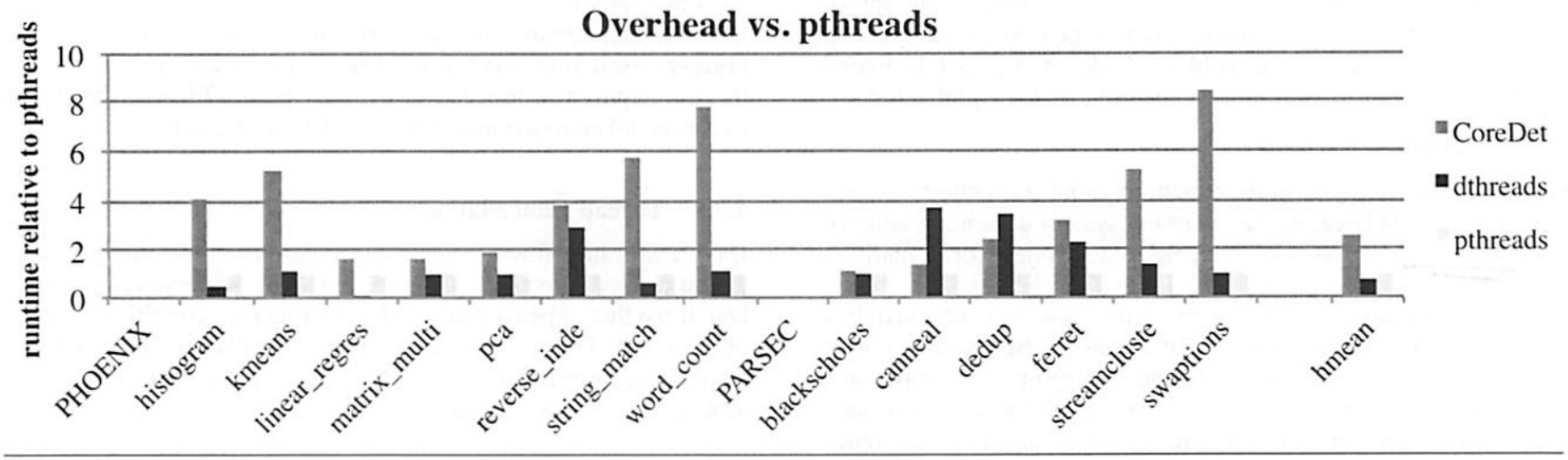

Figure 4. Normalized execution time with respect to `pthreads` (lower is better). For 9 of the 14 benchmarks, DTHREADS runs nearly as fast or faster than `pthreads`, while providing deterministic behavior.