Scaling Deterministic Multithreading

Marek Olszewski Jason Ansel Saman Amarasinghe

Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
{mareko, jansel, saman}@csail.mit.edu

Abstract

Today's parallel architectures provide an execution model that does not guarantee deterministic results for parallel programs. This nondeterminism causes significant challenges for parallel programmers by making it difficult to write parallel applications with repeatable results. As a consequence, parallel applications are significantly harder to debug, test, and maintain than their sequential predecessors.

Deterministic multithreading, which provides an always-on deterministic execution model for parallel applications, has emerged as a potential solution to this problem. In previous work, we introduced the Kendo deterministic multithreading framework. Kendo provides deterministic multithreading by enforcing a deterministic ordering of synchronization operations, which results in deterministic execution for programs that are correctly synchronized. Our prior results showed that Kendo incurs only minimal overheads on programs running with a small number of threads. However, our original approach suffered from a number of issues that limit its scalability to a large numbers of threads.

This paper present two important advances to the Kendo deterministic multithreading framework to improve its scalability. The first enables multiple threads to perform independent synchronization operations in parallel, eliminating a major serialization bottleneck in the original technique. The second advance uses combining trees to remove all-to-one communication when acquiring locks. Experimental results with a scalability microbenchmark show results that appear to be asymptotically superior to our previous approach with an orders of magnitude performance improvement at 48 threads. Additionally, results from a stochastic model evaluation predict that a key portion of our new approach will continue to scale past 48-cores.

1. Introduction

While multicore processors continue their proliferation, writing parallel applications that target them remains a daunting task. The nondeterministic execution model provided by these architectures causes significant challenges for parallel programmers by hindering their ability to create multithreaded applications with repeatable results. Under this nondeterministic model, the interleaving of memory accesses to shared data in multithreaded applications can vary from run to run, resulting in a potentially exponential space of legal program states that a programmer must reason about. Additionally, debugging nondeterministic programs poses significant challenges since bugs may not be reproducible from run to run. Such *Heisenbugs* [9] limit the use of cyclic debugging techniques, which try to use program re-execution to iteratively hone in on a bug. Nondeterminism also complicates traditional testing strategies, which can no longer offer strong correctness guarantees for the tested inputs [11, 12], and makes it difficult to use multithreaded code for replica-based fault tolerant systems [15].

Deterministic multithreading (DMT) has emerged as a potential solution to this problem [13, 8, 6, 5, 3, 4, 2, 7]. DMT provides an always-on deterministic execution model for parallel applications, eliminating the nondeterminism problem altogether. DMT approaches can enforce two types of determinism: *weak determinism*, where the ordering of synchronization operations, such as locks, is deterministic; or *strong determinism*, which deterministically orders all memory accesses [13]. Strong determinism guarantees reproducibility for all programs. While attractive, strong determinism can be expensive to enforce as it requires ordering all memory accesses to shared data. In contrast, weak determinism guarantees deterministic execution for programs that are correctly synchronized (a property that can be checked, for a given input, with a dynamic race detector [13, 14, 7]). Techniques that offer weak determinism can be more efficient because they can ignore the ordering of the majority of memory accesses. Additionally, their exclusive focus on high level synchronization operations enables them perform additional optimizations by leveraging the semantics of such synchronization operations.

In previous work, we introduced Kendo [13], a framework for enforcing weak determinism. We showed that Kendo can provide deterministic execution with minimal overheads on machines with a small number of processors. The approach uses a logical clock and turn-based scheduling algorithm that serializes the order with which threads can begin synchronization operations. However, while the raw performance on benchmarks with few processors is good, the approach does not scale well to systems with large numbers of processors because of this serialization.

In this paper, we introduce a new version of the Kendo deterministic multithreading algorithm to resolve these scalability issues. We present two advances to our original algorithm. The first change allows thread turns to be executed in parallel, eliminating the serialization bottleneck in the original algorithm. The second change improves the scalability of the turn-waiting algorithm. We provide an experimental evaluation of the scalability of both our old and new algorithms both with a microbenchmark on a 48-core system and using a stochastic model to predict how wait times will grow beyond 48-cores.

2. Kendo Background

In the original Kendo design [13], which we will subsequently refer to as Kendo-O, a deterministic schedule of synchronization operations is dynamically constructed using P monotonically increasing logical clocks, one for each thread. Each logical clock tracks the progress of a thread in a manner that is repeatable but also serves as a good estimate of the thread's execution. In Kendo-O, we use performance counters to automatically advance each thread's clock, using a hardware interrupt, every time it executes N store instructions, where the interval, N, is user specified.

The construction of the deterministic schedule revolves around the concept of a *turn*. It is only one thread's *turn* at any time, and the order of turns is deterministic. A thread's turn begins when its logical clock becomes the global minimum (with ties broken by thread id) and ends when the thread increments its logical clock, causing another thread to become the new global minimum. Thus, any work performed during a thread's turn will be reproduced deterministically on each run since the turn-ordering is always the same. By using clocks to determine turn-ordering (rather than a token), threads that do not wish to perform synchronization operations can easily and quickly forgo future turns by advancing their logical clocks without waiting to become the minimum.

Figure 1 shows the pseudocode for the Kendo-O deterministic mutex acquire and release operations. First, logical clock counting is paused to prevent the thread's clock from unexpectedly changing while it waits for, and later takes, its turn. Next, the thread enters the spin loop where it will repeatedly attempt to acquire the lock. Within this loop, the algorithm calls `wait_for_turn` to enforce the deterministic ordering with which threads may attempt to acquire a lock. Next the thread attempts to `trylock` a nondeterministic *underlying lock* associated with the mutex. If a thread fails to acquire the lock, it increments its logical clock to end its turn and retries. If it succeeds, it makes sure that its clock is greater or equal to the clock of the releasing thread (which it stored in the lock structure). If not, it releases the lock, ends its turn and starts again. This process eliminates a race condition that would occur with the releasing thread calling `det_mutex_unlock`, which unlocks the underlying lock without waiting for a turn. Once the thread successfully acquires the lock, it exits the loop, ends its turn and re-enables the logical clock counting.

```
function det_mutex_lock(mutex) {
  pause_clock();
  while (true) {
    wait_for_turn();
    if (trylock(mutex)) {
      if (mutex.released_clock $\geq$ clock_read(tid)) {
        unlock(mutex);
      } else {
        break;
      }
    }
    clock_add(tid, 1);
  }
  clock_add(tid, 1);
  resume_clock();
}
```

(a) Deterministic Lock Acquire

```
function det_mutex_unlock(mutex) {
  pause_clock();
  mutex.released_time = clock_read(tid);
  unlock(mutex);
  clock_add(tid, 1);
  resume_clock();
}
```

(b) Deterministic Lock Release

Figure 1. Pseudocode for the Kendo-O deterministic mutex acquire and release routines.

On the release side, each thread pauses the clock counting, stores its logical clock in the mutex, and then performs a standard nondeterministic unlock on the underlying lock object. It also performs an increment to the thread's deterministic logical clock.

3. Scalability of Kendo-O

While performing well on large benchmarks [16] with up to 4 threads, the Kendo-O algorithm has a number of limitations that inhibit its scalability. The first limitation is the serial section of code performed during a thread's turn. The second limitation is the `wait_for_turn` algorithm that queries each thread's logical clock to determine the global minimum (which requires at least P reads).

Additionally, one potential limitation to the scalability of Kendo-O is the time a thread spends waiting to become the thread with the minimum clock. This time is both theoretically unbounded and depends on the number of threads in the system, and is therefore a potential concern for scalability. We present a study to characterize how this will change with the number of threads in Section 5.1.

4. Scalable Kendo

In this section, we introduce two new variants of the Kendo algorithm that improve the scalability of Kendo-O. The first, which we will refer to as Kendo-P, eliminates the sequential turn taking bottleneck in the case where threads are performing independent synchronization operations. The second, which we will refer to as Kendo-T, eliminates the non-scalable all-to-one communication required to determine whether a thread has the minimum clock.

4.1 Kendo-P

The Kendo-P algorithm leverages the observation that threads that are accessing independent synchronization operations do not have to order their operations in a deterministic manner because the operations are themselves commutative. Thus, the `wait_for_turn` operation can be updated to not wait on a thread that has a lower logical clock if it is not concurrently accessing the same synchronization object *and* there is a sufficiently small enough difference between the two clocks that the second thread would not be able to access the same synchronization object without first incrementing its logical clock past the current thread's clock.

Figure 2 shows the pseudocode for the Kendo-P mutex acquisition code that uses this concept to allow threads to acquire different mutexes in parallel. In this code, each thread starts by recording the mutex that they are about to acquire and then adds an `ACQUIRE_AMOUNT` offset to its clock. Subsequently, to determine a thread's turn in the acquisition loop, each thread calls a modified turn waiting operation, which we call `wait_for_turn_offset`. Like `wait_for_turn`, `wait_for_turn_offset` reads and compares the logical clocks of all other threads to see if it is the global minimum. However, while examining another thread's logical clock, it also checks whether the other thread is waiting on the same mutex. If not, then it is guaranteed that this thread's clock will increase by at least `ACQUIRE_AMOUNT` before attempting to acquire the same mutex, so this amount can be added to the other thread's clock when doing the comparison. Thus, when two threads attempting to acquire independent mutexes have clocks whose difference is less than or equal to `ACQUIRE_AMOUNT`, they will complete their calls to `wait_for_turn_offset` and be able to acquire the mutex concurrently. By tuning this constant to be within a small multiple of the standard deviation of observed differences in logical clocks during the execution of a program, we can ensure that, in expectation, the majority of mutex acquisitions to independent locks will occur in parallel.

```
function det_mutex_lock(mutex) {
  pause_clock();
  set_waiting_for(tid, mutex);
  clock_add(tid, ACQUIRE_AMOUNT);
  while (true) {
    wait_for_turn_offset(mutex, ACQUIRE_AMOUNT);
    if (trylock(mutex)) {
      if (mutex.released_clock $\geq$ clock_read(tid)) {
        unlock(mutex);
      } else {
        break;
      }
    }
    clock_add(tid, 1);
  }
  set_waiting_for(tid, NULL)
  clock_add(tid, 1);
  resume_clock();
}
```

Figure 2. Pseudocode for Kendo-P deterministic lock acquire.

4.2 Kendo-T

While Kendo-P allows multiple threads to access different synchronization objects in parallel, it continues to require reading the logical clocks of all threads in every call to `wait_for_turn_offset`. Thus, the lower bound cost of performing a synchronization operation for Kendo-P (and Kendo-O) increases linearly as new threads are added. This may limit scalability at large thread counts. In this section, we describe a new algorithm, which we will call Kendo-T, that builds on the concepts of the previous algorithm with a more scalable technique for determining thread turns.

Central to the design, is the use of a new concurrent minimum combining tree (MCT), that enforces the invariant that each node in the tree stores a logical clock and tid that is less than or equal to the logical clocks and ids of its children. The data structure supports wait-free $O(1)$ lookups to determine the current minimum logical clock (and associated thread id), as well as an efficient obstruction-free [10] $O(\lg P)$ increment operation that updates the tree when a thread's logical clock increases. Our MCT data structure supports the following operations:

- `mct_set_clock` – Sets the clock of a given thread to a given value, adding the thread if needed, propagating the change up the tree.
- `mct_clear_clock` – Remove the thread from the given MCT.
- `mct_get_clock` – Read the clock of a given thread.
- `mct_wait_min_ge` – Block until the minimum clock in the tree is greater than or equal to the given value (breaking ties by thread id). Implemented by spinning on the value of root node of the combining tree.

Figure 3 shows the updated code for deterministic lock acquires using our MCT data structure in Kendo-T. This code is similar to Kendo-P, except that the all-to-one communication for `wait_for_turn_offset` is replaced with two MCT data structures to efficiently propagate the minimum clock between threads. Threads now never directly read the clock of another thread and instead just poll the root of one of the combining trees. The first MCT data structure, `global_mct` computes the minimum clock for all threads. The second MCT data structure, `mutex.waiting_mct` is a per-mutex structure for storing the minimum clock of only the threads waiting on a specific mutex. When waiting for its turn, a thread first waits for all threads to reach its current clock minus the `ACQUIRE_AMOUNT`, then it waits for its own clock to be the minimum of those threads in `mutex.waiting_mct`. When called one after another, the two tree-based waits have the same behavior that `wait_for_turn_offset` had in Kendo-P, where a thread waits for other threads not accessing the current lock to be within `ACQUIRE_AMOUNT` clock ticks. The remaining differences between Kendo-P and Kendo-T are to propagate clock information to the two MCT data structures in the correct order.

```
function det_mutex_lock(mutex) {
  pause_clock();
  long my_clock = mct_get_clock(global_mct, tid);
  my_clock += ACQUIRE_AMOUNT;
  mct_set_clock(mutex.waiting_mct, tid, my_clock);
  mct_set_clock(global_mct, tid, my_clock);
  while (true) {
    mct_wait_min_ge(global_mct, tid,
        my_clock − ACQUIRE_AMOUNT);
    mct_wait_min_ge(mutex.waiting_mct, tid, my_clock);
    if (trylock(mutex)) {
      if (mutex.released_clock ≥ my_clock) {
        unlock(mutex);
      } else {
        break;
      }
    }
    my_clock += 1;
    mct_set_clock(mutex.waiting_mct, tid, my_clock);
    mct_set_clock(global_mct, tid, my_clock);
  }
  mct_set_clock(global_mct, tid, my_clock + 1);
  mct_clear_clock(mutex.waiting_set, tid);
  resume_clock();
}
```

Figure 3. Pseudocode for Kendo-T deterministic lock acquire.

Figure 4 shows the pseudocode for one of the key MCT functions, `mct_set_clock` and its helper function `mct_bubble_up` which propagates changes up the tree. `mct_bubble_up` is a recursive function that updates the clock of a node, after a thread makes a change to the node's child. The function starts with the condition on line 9 to test if the value of the node's child, before the update, was previously the minimum out of the node's children. If not, the update to the child would not have changed the minimum of the node's children, and the operation can complete. In the case that two children concurrently update their clocks (where the true minimum changes twice), at least one thread is guaranteed to enter this condition since the old value is read before any modifications are made. Once in the condition, the thread will read the logical clocks of all of the node's children and attempt to update the node's value. To handle the race when multiple threads change the minimum in sequence, two actions are performed. First, the CAS on line 14 guarantees that only one update takes place at a time. Secondly, a second polling of the minimum on line 19 is required to handle the case where the minimum changes between lines 13 and 14, since the thread that caused this change might not enter the update loop.

```
1  function mct_set_clock(mct, tid, val) {
2    node = mct_get_leaf_node(tid);
3    old_clock = node.clock;
4    node.clock = val;
5    mct_bubble_up(node.parent, old_clock);
6  }
7
8  function mct_bubble_up(node, old_child_clock) {
9    if (node.clock ≥ old_child_clock) {
10     old_clock = node.clock;
11     while (true) {
12       clock = node.clock;
13       min_clock = mct_get_min_of_children(node);
14       if (!CAS(&node.clock, clock, min_clock)) {
15         continue;
16       }
17       // Re−read the children to see if minimum changed.
18       new_min_clock = mct_get_min_of_children(node);
19       if (min_clock == new_min_clock) {
20         if (node.parent) {
21           mct_bubble_up(node.parent, old_clock);
22         }
23         break;
24       }
25     }
26   }
27 }
```

Figure 4. Pseudocode for Kendo-T clock update code, including the recursive MCT combine operation.

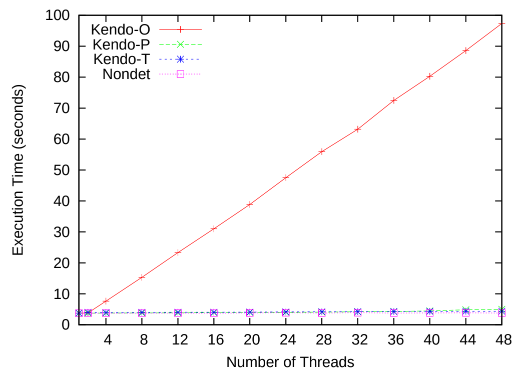
5. Evaluation

We have evaluated each of the three described variants of the Kendo system on a scalability microbenchmark. Our test machine consisted of a 48 core (4×12 cores) AMD Opteron 6168 1.9GHz machine running Debian 5.0.

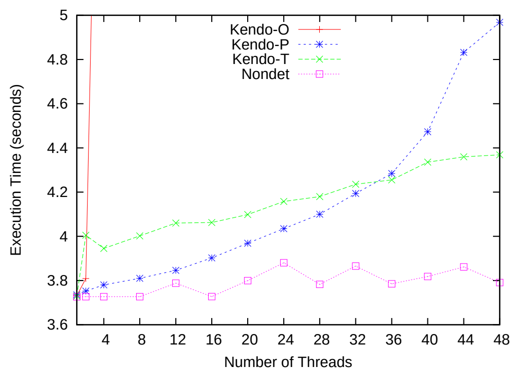
Figure 5 lists the pseudocode for our scalability microbenchmark. The microbenchmark spawns `NUM_THREADS` threads operating in a tight loop acquiring and releasing a private mutex. Note that since the total amount of work grows with the number of threads, perfect scalability would be a horizontal line. When run without a deterministic multithreading system, this benchmarks scales extremely well

```
function thread_start() {
  for (i = 0; i < 100000; i++) {
    kendo_mutex_lock(mutexes[tid]);
    for (j = 0; j < 100; j++) {
      /∗ nop ∗/
    }
    kendo_mutex_unlock(mutexes[tid]);
    for (j = 0; j < 10000; j++) {
      /∗ nop ∗/
    }
  }
}
```

Figure 5. Pseudocode for our scalability microbenchmark



(a) Zoomed out.



(b) Zoomed in.

Figure 6. Execution times of the microbenchmark shown in Figure 5.

since there is no communication between threads. It is important to note that this benchmark is always deterministically since all the synchronization operations are to private mutexes. However, since neither of our approaches leverage techniques for tracking and transferring exclusive ownership of synchronization objects (e.g.: the approaches taken by DMP-ShTab/DMP-O [8, 3]), this benchmark represents a worst case scenario for our system[1].

Figure 6 shows the performance results for the microbenchmark running nondeterministically with pthreads and with each of the three Kendo versions. We can see the poor scalability of the original Kendo system (Kendo-O) in Figure 6(a). Kendo-O's execution time grows significantly with each new thread added and at 48 threads it incurs a $25.7\times$ overhead over the nondeterministic pthreads execution. In contrast, both new versions of Kendo scale significantly better (close up show in Figure 6(b)). Kendo-P performs very well at low thread counts, but its overhead begins to grow rapidly at 36 threads and above. We attribute this to the high degree of cache coherence traffic between all 4 chips that is required for all threads to repeatedly read the logical clocks of all of the other threads each time they wait for their turn. In contrast, Kendo-T incurs only a modest initial overhead at 2 processors due to the increased cost required to update the minimum combining tree on every logical clock increment. However, due to the scalability of this data structure, Kendo-T's execution grows very slowly with threads added. At 48 cores, Kendo-T incurs only a 15% overhead and performs 22.3 times better than the original version of Kendo.

5.1 Analysis of Wait Times

One of the major potential limitations to scalability of our all of our proposed algorithms is the time spent in `wait_for_turn` or `wait_for_turn_offset` waiting for the logical clock of other threads to reach a given point. This wait time is required to maintain a deterministic ordering, and is both theoretically unbounded and depends on the number of threads in the system. As a result, we are interested in predicting how it will vary as we add more than 48 threads.

To help characterize the scalability of wait times we conducted a Monte Carlo simulation to try to predict how wait times will increase with thread counts greater than 48. To make this simulation tractable, we made the simplifying assumption that for a given program point, the probability distribution of the difference between each threads clock and the average is a normal distribution. For programs that have logical clocks that can match their execution time well, we believe that this is a reasonable first order approximation because it has been shown that in the presence of random delays, parallel process execution times asymptotically approach a normal distribution [1].

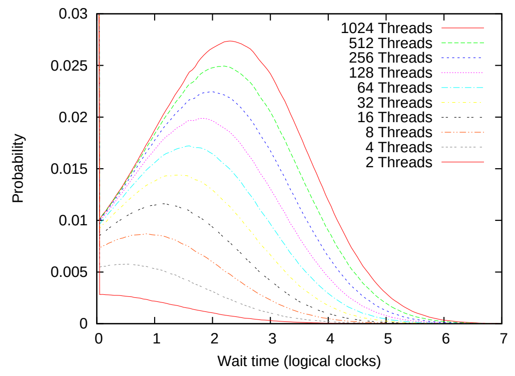
We construct the simulation by modeling a random variable, Y_t, describing the probability distribution of the amount of time (counted in logical clock ticks) thread t will have to wait when arriving at a uncontested synchronization operation. In the case where all other threads are executing in parallel (either by executing code that is not communicating or is performing independent synchronization operations), then the amount of time a thread has to wait for is simply its offset from the minimum clock at the time of arrival, which can be written as:

$$Y_t = X_t - min(X_1, ..., X_P)$$

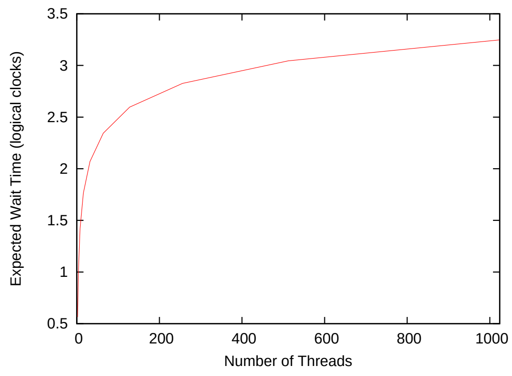
where X_i is the deviation from the average logical clock of all threads, for thread i, and $min(X_1, ..., X_P)$ represents the largest negative of these deviations. Figure 7(a) displays the results of this simulation showing the distribution of Y_t for different numbers of threads.

We can use this distribution to predict the expected value of the wait time experienced by a thread upon reaching a uncontested synchronization object. Figure 7(b) shows these expected values for growing thread counts. We see that the

[1] We have found that techniques that transfer exclusive ownership of mutexes do not perform well on larger benchmarks and have therefore chosen to avoid them (this is in line with the results seen with CoreDet [3] where the DMP-B variant scales better than DMP-O)



(a) Probability distribution of wait time.



(b) Expected wait time.

Figure 7. Results of a Monte Carlo simulation modeling wait time for different numbers of threads.

expected wait time grows logarithmically with the number of threads, which makes us optimistic about the future scalability of our approach.

6. Conclusion

In this paper we have presented two new changes to the original Kendo algorithm to improve its scalability. The first change allows threads to take turns in parallel, eliminating a major sequential bottleneck in the original algorithm. The second change improves the algorithmic complexity of the turn waiting algorithm. We demonstrated the performance of these two versions on a microbenchmark when running with up to 48 threads, and showed a performance improvement of up to 22.3 times when compared to the original version of Kendo. Additionally, we used a stochastic model to predict the scalability of the wait times under our approach for machines with more than 48 processors.

References

[1] V. S. Adve and M. K. Vernon. The influence of random delays on parallel execution times. In *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 61–73, New York, NY, USA, 1993.

[2] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient system-enforced deterministic parallelism. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–16, 2010.

[3] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. CoreDet: A compiler and runtime system for deterministic multithreaded execution. In J. C. Hoe and V. S. Adve, editors, *Proceeding of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 53–64. ACM, 2010.

[4] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble. Deterministic process groups in dOS. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–16, 2010.

[5] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: safe multithreaded programming for C/C++. In *Proceeding of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA'09, pages 81–96, 2009.

[6] 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, 2009.

[7] H. Cui, J. Wu, C.-C. Tsai, and J. Yang. Stable deterministic multithreading through schedule memoization. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–13, 2010.

[8] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: Deterministic shared memory multiprocessing. In *Proceeding of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '09, pages 85–96, 2009.

[9] J. Gray. Why do computers stop and what can be done about it? In *Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12, 1986.

[10] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, ICDCS '03, pages 522–529, 2003.

[11] E. A. Lee. The problem with threads. *Computer*, 39(5):33–42, May 2006.

[12] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 267–280, 2008.

[13] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: Efficient deterministic multithreading in software. In *The International Conference on Architectural Support for Programming Languages and Operating Systems*, Washington, DC, Mar 2009.

[14] M. Ronsse and K. De Bosschere. Recplay: A fully integrated practical record/replay system. *ACM Transactions on Computer Systems*, 17:133–152, May 1999.

[15] J. H. Slye and E. N. Elnozahy. Supporting nondeterministic execution in fault-tolerant systems. In *Proceedings of the The Twenty-Sixth Annual International Symposium on Fault-Tolerant Computing*, FTCS '96, pages 250–259, Washington, DC, USA, 1996.

[16] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, 22-24 Jun 1995.