Kendo: Efficient Deterministic Multithreading in Software

Marek Olszewski Jason Ansel Saman Amarasinghe
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
{mareko, jansel, saman}@csail.mit.edu

Abstract

Although chip-multiprocessors have become the industry standard, developing parallel applications that target them remains a daunting task. Non-determinism, inherent in threaded applications, causes significant challenges for parallel programmers by hindering their ability to create parallel applications with repeatable results. As a consequence, parallel applications are significantly harder to debug, test, and maintain than sequential programs.

This paper introduces Kendo: a new software-only system that provides deterministic multithreading of parallel applications. Kendo enforces a deterministic interleaving of lock acquisitions and specially declared non-protected reads through a novel dynamically load-balanced deterministic scheduling algorithm. The algorithm tracks the progress of each thread using performance counters to construct a deterministic logical time that is used to compute an interleaving of shared data accesses that is both deterministic and provides good load balancing. Kendo can run on today's commodity hardware while incurring only a modest performance cost. Experimental results on the SPLASH-2 applications yield a geometric mean overhead of only 16% when running on 4 processors. This low overhead makes it possible to benefit from Kendo even after an application is deployed. Programmers can start using Kendo today to program parallel applications that are easier to develop, debug, and test.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming – Parallel Programming; D.2.5 [*Software Engineering*]: Testing and Debugging – Debugging Aids; D.4.1 [*Operating Systems*]: Process Management – Synchronization

General Terms Design, Reliability, Performance

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

1. Introduction

Application developers rely heavily on the fact that given the same input, a program will produce the same output. Sequential programs, by construction, typically provide this desirable property of deterministic execution. However, in shared memory multithreaded programs, deterministic behavior is not inherent. When executed, such applications can experience one of many possible interleavings of memory accesses to shared data. As a result, multithreaded programs will often execute non-deterministically following different internal states that can sometimes lead to different outputs. For programs that are not inherently concurrent, such non-determinism is almost never required in the program's specification and comes directly as a consequence of parallelizing the program for improved performance on today's machines. This added non-determinism makes parallel applications significantly harder to debug, test, and maintain than sequential programs (Lee 2006).

In this paper, we argue that non-determinism is not a requisite aspect of threads. Instead, thread communication through shared memory can be interleaved in a deterministic manner in order to restore the determinism guarantees provided by sequential programs. We define this property as *deterministic multithreading*, and classify it into the following two categories:

- *Strong determinism* ensures a deterministic order of all memory accesses to shared data for a given program input.
- *Weak determinism* ensures a deterministic order of all lock acquisitions for a given program input.

Strong determinism is guaranteed to produce the same output for every run with a given program input. While this is an attractive property, we conjecture that it cannot be provided efficiently without hardware support. Weak determinism offers the same guarantee for exactly those inputs that lead to race-free executions under the deterministic scheduler – that is, executions in which all accesses to shared data are protected by locks. For a given input, this property can be checked with a dynamic data race detector. For programs without data races, strong determinism and weak determinism offer equivalent guarantees. We describe additional benefits of weak determinism in Section 2.

A number of existing parallel programming models also offer an improved level of determinism for specific styles of parallelism. In the fork/join model used by the Cilk language (Frigo et al. 1998), Cilk can detect data races and (if locks are not used) offer deterministic outcomes in their absence. Programs can also be collapsed to a sequential version for testing. However, it is less clear how to extend this functionality to arbitrary threaded code. While code parallelized with OpenMP can also be reduced to a sequential and deterministic version for testing, the parallel version may admit thread interleavings with different behaviors.

Additionally, record/replay systems can be used to help programmers reproduce bugs in programs that behave non-deterministically. These systems can provide strong determinism between a single record process and a set of replay processes. However, record/replay can provide neither strong nor weak determinism between different execution recordings. Two executions with identical inputs running in isolation are not guaranteed to behave the same way. Thus, multithreaded record/replay systems only selectively enforce determinism, limiting their application.

Programmers may also try to manually ensure that all interleavings yield the same program output, for example, by writing a program that uses only commutative updates to shared data and does not otherwise test or branch on intermediate values. However, such programs require careful construction and may be bug-prone or overly restrictive.

Figure 1 illustrates an example of a program that cannot easily be made deterministic using common parallel programming idioms. The program performs repeated non-commutative updates on a globally visible shared data structure, and uses a task queue to dynamically load-balance the work in an efficient manner. This pattern can result in non-deterministic executions and is exhibited by a number of well known parallel applications such as: Radiosity (Singh et al. 1994), LocusRoute (Rose 1988), and Delaunay Triangulation (Kulkarni et al. 2008).

There are two sources of non-determinism in this example, both are caused by races on synchronization objects. First, the task queue distributes work on a first-come first-served basis, making the work assigned to each thread non-deterministic. Second, the order in which each thread modifies a portion of the global shared data structure depends on the order in which each thread can acquire the lock or locks that protect it. Since the operations performed on the data structure are non-commutative, the resulting changes made to the shared data structure are non-deterministic.

It is not immediately clear how this example can be made deterministic efficiently. A naïve approach would force threads to acquire locks in a round robin manner such that each thread has to wait until all other threads have acquired a lock between its own acquisition attempts. However, this approach sacrifices load balancing if the frequency of lock acquisitions varies across tasks. Achieving determinism while still maintaining good load balancing is significantly harder and requires a notion of thread progress when determining the lock acquisition schedule.

```
// Globally visible shared state
global_state = init_state();

// Enqueue first task in task queue
task = create_initial_work();
task_queue.push(task);

fork_threads(NUM_THREADS);

// Loop until there is no more work.
while (!task_queue.completed())
{
  task = task_queue.pop();

  // Non-commutative operation on global state.
  // May enqueue more tasks.
  do_work(global_state, task);
}

join_threads(NUM_THREADS);
```

Figure 1. Task queue with non-commutative updates to global state pattern common in non-deterministic parallel programs.

1.1 Determinism via Kendo

In this work, we present Kendo: a software framework that can efficiently enforce weak deterministic execution of general purpose lock-based `C` and `C++` code targeting today's commodity shared memory chip-multiprocessors.

To achieve determinism, we introduce the concept of *deterministic logical time*, which is used to track the progress of each thread in a deterministic manner. Kendo uses deterministic logical time to compute a deterministic yet load-balanced interleaving of synchronized accesses to shared data. Because deterministic logical time can be accurately reproduced, Kendo is able to enforce a repeatable interleaving of lock acquisitions across program executions.

Kendo implements a subset of the POSIX Threads API and provides novel mechanisms to let users safely and deterministically perform unprotected, or racy, accesses to shared data. The resulting set of synchronization operations is sufficient to allow programmers to easily develop parallel applications that exhibit deterministic behavior.

Kendo runs on today's commodity hardware and incurs only a modest performance cost. Experimental results show that the applications from the SPLASH-2 benchmark suite yield a geometric mean overhead of only 16% when running on a 4-core processor. This low overhead makes Kendo practical to run even after applications are deployed. As a result, Kendo lets programmers focus their time on finding and exploiting parallelism within their algorithms without worrying about maintaining determinism, which can be difficult and expensive.

1.2 Contributions

This paper makes the following contributions: (i) we introduce the concept of weak and strong determinism; (ii) we introduce the notion of deterministic logical time and show how to efficiently obtain it on today's commodity hardware; (iii) we present a new algorithm that uses deterministic logical time to efficiently provide weak determinism on today's commodity multiprocessors. This technique is the first to provide deterministic execution of parallel applications on commodity machines without requiring a record stage; (iv) we demonstrate the practicality of our approach by evaluating it on the SPLASH-2 benchmark suite.

2. Benefits of Deterministic Multithreading

In this section we discuss a number of benefits provided by a deterministic multithreading execution model such as the Kendo framework.

Repeatability: Users have come to expect a repeatability guarantee from software. Given the same inputs, the program should produce the same outputs. For example, customers of FPGA CAD software require that their HDL code is compiled in a deterministic manner so that they can reliably test their own work. Unfortunately, record/replay systems are not a practical means of ensuring such deterministic application behavior. At most, record/replay systems can perform separate recording for each possible program input, which is not feasible for most programs. Since Kendo does not need to store record logs, it can provide a practical means of guaranteeing repeatability. Additionally, because Kendo is portable across micro-architectures and can execute with low overhead, it can be feasibly left on once an application is deployed.

Debugging: Sequential application developers depend heavily on determinism to reproduce and debug erroneous runtime behavior. Programmers often utilize a systematic cyclic debugging methodology to iteratively obtain information about a bug by repeatedly running the program to hone in on the problem. This technique does not lend itself well to non-deterministic applications since bugs may not be reproducible on every run.

Record/replay systems can be used to help replicate faulting program executions to help with cyclic debugging; however, they require that the initial execution that triggered the bug was performed during a recording session. In the absence of low overhead hardware, software recording is unlikely to be enabled during application deployment because of overhead (Dunlap et al. 2008). Additionally, since even the best hardware record/replay systems to date require gigabytes of logs per day (Montesinos et al. 2008), they are also unlikely to be enabled in many situations. Thus, in practice, record/replay systems offer few benefits to restore the debugging methodologies currently applied to sequential applications.

In contrast Kendo can deterministically reproduce bugs even if they were discovered on commodity hardware. Kendo precisely reproduces all non-concurrency bugs as well as deadlocks, atomicity violations, and order violations in correctly synchronized code. Such bugs have been shown to make up a large fraction of concurrency bugs found in real parallel applications (Lu et al. 2008).

Additionally, Kendo can be combined with a dynamic race detector to help identify races that are a result of incorrect synchronization. Under Kendo, a dynamic race detector is guaranteed to detect the first race to occur on a given input since the program will run deterministically up until that point. Thus, when a bug is encountered for a particular input, a programmer can systematically eliminate all races using a race detector, and will subsequently be able to reproduce all remaining bugs. Therefore, when combined with a race detector, Kendo offers a systematic way to reproduce an observed bug and/or a related race. As a result, Kendo can be used to eliminate all bugs for the tested set of inputs.

Testing: Comparing a program output to previously created "correct" output is a standard technique of verifying correctness in regression testing. This approach does not fare well with parallel applications that exhibit non-deterministic output, or have non-deterministic internal state that needs to be verified. By using Kendo programmers can eliminate non-determinism to enable correctness testing via program equivalence (Lee 2006). In this way, Kendo can make parallel applications more like sequential applications when it comes to maintaining current testing infrastructures. In comparison, record/replay systems offer no effective method of proving equivalence because a recorded run represents only one of many possible non-deterministic executions.

Multithreaded Replicas: Many replica-based fault tolerance systems depend on programs being deterministic. In such systems, each replica is provided with the same program input and is expected to behave uniformly in the absence of program error. When all non-faulty replicas produce the same output, a correct consensus can often be reached on the basis of a quorum. Non-determinism makes it nearly impossible to differentiate between correct and incorrect outputs and therefore makes it harder for replicas to come to a consensus. While a number of algorithms have been suggested that can ensure that all replicas execute deterministically with respect to each other, each requires significant communication among replicas. Kendo can be used to create deterministic replicas that do not require communication, thus increasing fault tolerance and reliability.

```
function det_mutex_lock(l)
{
  pause_logical_clock();
  wait_for_turn();
  lock(l);
  inc_logical_clock();
  resume_logical_clock();
}
```

(a) Deterministic Lock Acquire

```
function det_mutex_unlock(l)
{
  unlock(l);
}
```

(b) Deterministic Lock Release

Figure 2. Pseudo code for deterministic mutex lock acquire and release routines that do not support nested locking.

3. Design

In this section we describe our deterministic locking algorithms that are central to our design. The algorithms construct a deterministic interleaving of synchronization operations in deterministic logical time, which we first define.

3.1 Deterministic Logical Time

We use the notion of *deterministic logical time* as an abstract counterpart to physical time, which we use to deterministically order events in a shared memory parallel application. Deterministic logical time is constructed from P monotonically increasing *deterministic logical clocks*, where P is the number of threads in the application. Unlike Lamport Clocks (Lamport 1978), deterministic logical clocks are computed independently and never updated based on the progress of other threads. Such updates would make the clocks non-deterministic.

An event occurring on thread 1 is said to occur at an earlier deterministic logical time than an event on thread 2 if thread 1 has a lower deterministic logical clock than thread 2 at the time of the events. Deterministic logical clocks can be constructed by counting arbitrary events being performed by each thread, so long as those events are repeatable from run to run. It is desirable to choose events that track the progress of a thread in physical time as closely as possible because it makes any lock acquisition schedule computed from the clocks more load balanced. We discuss good sources for deterministic logical clocks in Section 4.1.

3.2 Locking Algorithm

The goal of our locking algorithm is to enforce a deterministic interleaving of lock acquisitions. This is done by simulating the interleaving that would occur if threads were to execute in deterministic logical time rather than physical time. For performance, threads are allowed to run with their deterministic logical clocks out of sync when executing code outside of critical sections, but they must wait for slower threads at lock acquisition points in order to guarantee determinism.

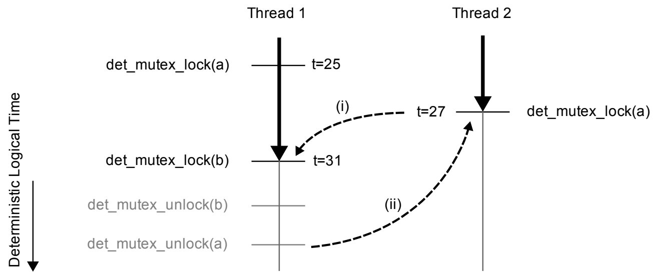

Figure 3. Example scenario where the simple algorithm can cause a deadlock. Note the cyclic dependence caused by the dependences (i) and (ii). Dependency (i) is due to thread 1 waiting for thread 2 to increase its deterministic logical clock to 31.

To help introduce the reader to our deterministic locking algorithm, we present two versions. The first, presented in Section 3.2.1, is a simplified algorithm that does not support nested locks. The second, presented in Section 3.2.2, fully supports nested locks.

3.2.1 Simplified Locking Algorithm

The simplified algorithm makes threads acquire a lock in an order defined by their deterministic logical clocks. Since each thread's deterministic logical clock is repeatable from run to run, the order of acquisitions must also be deterministic. The algorithm centers on the concept of a *turn*. It is only one thread's turn at a time, and the order of turns is deterministic. It is a thread's turn when both of the following are true:

1. All threads with a smaller id[1] have greater deterministic logical clocks.
2. All threads with a larger ID have greater *or equal* deterministic logical clocks.

Turn waiting enforces a first-come first-served ordering of threads in deterministic logical time. All threads keep their deterministic logical clocks in shared memory, and thus each thread can examine all other deterministic logical clocks to independently determine the turn ordering. A thread completes its turn by incrementing its own deterministic logical clock.

Figure 2 displays the pseudo code for the simple deterministic lock and unlock algorithms. First, the thread's deterministic logical clock must be paused to prevent the clock from changing while it waits for, and later takes, its turn. Next, the locking algorithm calls `wait_for_turn` to enforce the deterministic first-come first-served ordering with which threads may attempt to acquire a lock. Here, `lock` calls a

[1] We assign a unique thread ID to each thread when it is created.

```
function det_mutex_lock(l)
{
  pause_logical_clock();
  while (true)                          // Loop until we have successfully acquired the lock.
  {
    wait_for_turn();                    // Wait for our deterministic logical clock to be unique
                                        // global minimum.

    if(trylock(l))                      // Check the state of the lock, acquiring it if it is free.
    {
      if(l.released_logical_time        // Lock is free in physical time, but still acquired in
            >= get_logical_clock())     // deterministic logical time so we can not acquire it yet.
      {
        unlock(l);                      // Release the lock.
      }
      else
      {                                 // Lock is free in both physical and in deterministic logical
        break;                          // time, so it is safe to exit the spin loop.
      }
    }
    inc_logical_clock();                // Increment our deterministic logical clock and start over.
  }
  inc_logical_clock();                  // Increment our deterministic logical clock before exiting.
  resume_logical_clock();
}
```

(a) Deterministic Lock Acquire

```
function det_mutex_unlock(l)
{
  pause_logical_clock();
  l.released_logical_time = get_logical_clock();
  unlock(l);
  inc_logical_clock();
  resume_logical_clock();
}
```

(b) Deterministic Lock Release

Figure 4. Pseudo code for deterministic lock acquire and release. Some fairness and performance optimizations, described in Section 3.2.3, are omitted for clarity.

standard non-deterministic lock function to acquire an *underlying lock* object. Once a thread acquires an underlying lock, it increments its deterministic logical clock to allow other threads to proceed with their turns. Finally, the thread re-enables its deterministic logical clock and starts the critical section. On the release side, each thread simply performs a standard non-deterministic unlock.

3.2.2 Improved Locking Algorithm

The simplified algorithm described in Section 3.2.1 has a number of problems. First, a thread waiting on an acquired lock will prevent other threads from executing independent critical sections since it does not give up its turn until it holds the underlying lock. Second, the code does not properly handle nested locks. Lock nesting introduces possible dependences between threads that can cause deadlocks which were not present in the non-deterministic code. Figure 3 illustrates a scenario where such a deadlock can occur. When attempting to acquire lock `b`, thread 1 must wait for thread 2 to reach the same deterministic logical time (dependence (i)); however, thread 2 is stalled waiting for thread 1 to release lock `a` (dependence (ii)). The two dependencies cause a dependence cycle preventing both threads from making progress.

To address the two problems we change the locking algorithm so that a thread increments its deterministic logical clock as it spins on a contested lock (pseudo code presented in Figure 4(a)). This allows dependence (i) to be satisfied after some period of spinning (shown in Figure 5 a). Some subtlety is required in order to increment a thread's deterministic logical clock in a deterministic way. Each `lock` operation may be racing with a corresponding `unlock` operation in another thread, thus, a thread may or may not succeed in acquiring the lock during a given turn.

To eliminate this non-determinism, we impose the invariant that only one thread may hold a given lock at a given deterministic logical time (i.e., a thread cannot acquire a lock previously held by another thread until its deterministic logi-

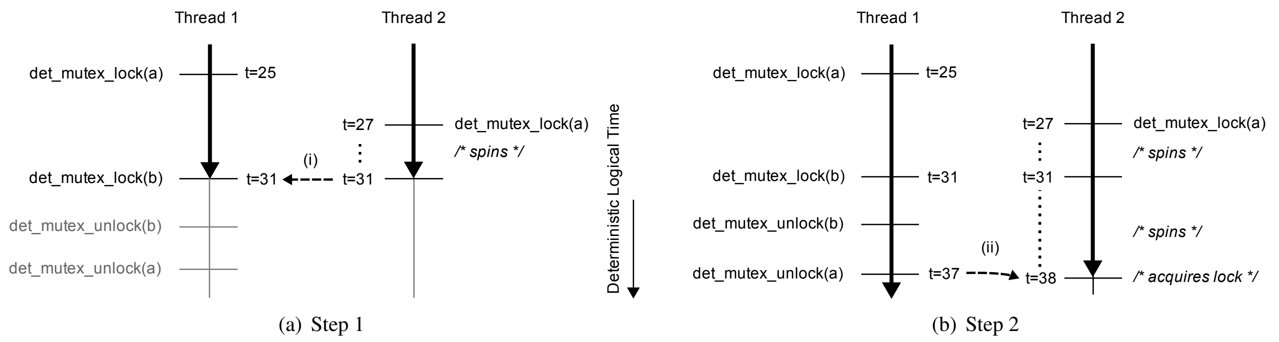

Figure 5. An illustration of how the improved algorithm solves the deadlock shown in Figure 3. When thread 2 fails to acquire lock a, it deterministically increments its deterministic logical clock until it reaches 31. At this point, the dependence (i) is satisfied, and thread 1 is able to make forward progress. In step 2, thread 2 continues to increment its deterministic logical clock until in reaches a deterministic logical time greater than when thread 1 released the lock. At this point, the second dependence (ii) is satisfied and both threads can proceed.

cal clock is greater than the deterministic logical clock of the other thread when it released the lock). This is enforced by having the last thread to hold the lock store its deterministic logical clock at the time of release, and preventing threads from acquiring the lock if they have yet to pass that deterministic logical time. Thus, threads can fail to acquire a lock in one of two ways: if the lock is held by another thread (in which case the `trylock` fails), or if it is released but still "acquired" in deterministic logical time (in which case the `trylock` succeeds but the deterministic logical time check fails). In the latter case, the deterministic logical time check is performed after the lock is acquired to eliminate a possible race with the thread releasing the lock. If the check fails, the lock must be released. If the lock is free in both real and deterministic logical time, the thread holds on to the acquired lock, exits the spin loop and increments its deterministic logical clock. Every time a thread fails to acquire the lock, it increments its deterministic logical clock and waits for a new turn.

Figure 4(b) shows the improved deterministic lock release code. In addition to the change that makes each thread record its deterministic logical clock before it releases the lock, the modified code also includes an increment to the thread's deterministic logical clock. This enables any spinning threads to quickly reach a deterministic logical time that will allow them to acquire the lock.

3.2.3 Locking Algorithm Optimizations

Queuing for fairness. One remaining problem with our algorithm as presented above is that it does not preserve fairness. Rather than preferring the thread with the lowest deterministic logical clock at the time it calls the lock function, the thread with smallest ID will always "win" a heavily contested lock because of the turn ordering. We address this by introducing a queue structure in each lock. When a lock is already held, threads add themselves to this queue when they first attempt but fail to acquire the lock. Subsequently, a thread will only attempt to acquire the lock if it is at the front of the queue; all other threads simply call `inc_logical_clock`. This strategy guarantees that threads always acquire contested locks according to a first-come first-served ordering, in deterministic logical time. The state of the queue is deterministic because it is only modified during a thread's turn.

Deterministic logical clock fast-forwarding. When a thread is waiting for its deterministic logical clock to surpass `l.released_logical_time`, it can potentially increase its deterministic logical clock by more than one to catch up to the released logical time faster. This avoids the need for many threads to take turns incrementing their deterministic logical clocks. Without queuing, waiting threads can fast forward their logical clock to `l.released_logical_time`. With queuing, the thread at the head of the queue, if possible, can take the lock and set its deterministic logical clock to one greater than `l.released_logical_time`.

Lock priority boosting. If the next thread to acquire a specific lock can be accurately predicted then performance can be improved by prioritizing the thread for that lock. Each lock may be assigned a high priority thread that is allowed to attempt to acquire the lock without waiting for other threads to catch up to the same point in deterministic logical time. This is achieved by allowing the prioritized thread to privately subtract a constant from its deterministic logical clock when waiting for its turn while attempting to acquire the lock. To maintain correctness, the same constant must be added by all other threads to their own deterministic logical clocks when attempting to acquire the same lock. This approach can significantly improve the performance of correctly predicted lock acquisitions, though it comes at the cost of slower incorrectly predicted acquisitions. Thus, priority boosting is only desirable when lock acquisition patterns can be accurately predicted and is therefore disabled by default.

4. Kendo

In this section we provide a description of Kendo, our prototype implementation. Kendo implements a deterministic subset of the POSIX Threads (pthreads) API, and offers an additional *deterministic lazy read* API to accommodate programming styles that make use of non-protected accesses to shared data. Kendo includes small modifications to the Linux operating system to enable the use of performance counter events to construct deterministic logical clocks.

4.1 Deterministic Logical Clocks

Kendo uses performance counters to build deterministic logical clocks that can efficiently track the progress of each thread. We use a slightly modified version of the perfmon2 kernel patch to enable access to the counters.

We experimented with a number of possible events to construct a deterministic logical clock that is cheap to maintain but that can still track the progress of each thread closely. We limited our search to options that were portable across micro-architectures and exhibited low overhead. This led us to examine a number of performance counter events commonly available on modern x86 chip-multiprocessors. Unfortunately, many of the performance counter events we tested did not offer deterministic results. For example, both the `retired_instructions` and `retired_loads` events are non-deterministic because, for unknown reasons, they appear to include interrupt occurrences in their counts. Fortunately, the `retired_stores` event does not exhibit this peculiarity and is therefore suitable for generating a deterministic logical clock.

While performance counter events are effective for tracking a thread's position in deterministic logical time, they are not accessible to other threads, which is necessary for our locking algorithm. Therefore, each thread maintains its deterministic logical clock in shared memory, computing it indirectly from the performance counters. This is accomplished by registering an interrupt handler that increments each thread's deterministic logical clock whenever the performance counter overflows.

Since performance counter overflow interrupts are non-precise on today's x86 micro-architectures, performing an accurate deterministic logical clock reading at an arbitrary point in the dynamic execution can present a challenge. Before a thread can read its deterministic logical clock value it must ensure that no interrupts are pending. To check for a pending interrupt from within the Kendo library, we enable the Read Performance-Monitoring Counters (`rdpmc`) instruction for user space access. A positive value in the performance counter indicates that the counter has overflowed and the interrupt handler has not yet executed. Therefore, each thread has to wait for the contents of the performance counter to become negative before reading its deterministic logical clock. We use the same technique whenever we need to pause a thread's deterministic logical clock to ensure that no overflows are missed before the counter is disabled.

There are two deterministic logical clock related parameters that can be tuned to improve the performance of the deterministic locking algorithm: *chunk size* and *increment amount*. Chunk size represents the number of stores needed to trigger a performance counter interrupt that will increment a thread's deterministic logical clock. A smaller chunk size will improve the quality of a thread's deterministic logical clock, thus decreasing wait time, but incur more overhead from the interrupt handlers. We discuss this trade off some more in Section 5.3. Increment amount is the amount by which a thread's deterministic logical clock is increased in each interrupt handler. We use this to modify the ratio between deterministic logical clock increments done as a result of application stores and those done by our locking algorithm as a result of lock acquisitions and releases. Putting a greater emphasis on the interrupt increments improves performance when there is low contention, while putting emphasis on the lock-based increments improves performance when there is high lock contention. The optimal value for both of these settings is application dependent.

4.2 Thread Creation

Kendo provides a `det_create` routine that extends the POSIX `pthread_create` routine to ensure that our lock algorithm remains deterministic in the face of thread creation. To ensure determinism, the order of thread creation requests must be deterministic because thread IDs affect how ties are broken when acquiring locks. Additionally, the initial deterministic logical clock of created threads must be deterministic. Finally, threads must be created in such a way that existing threads begin waiting on them deterministically.

To deterministically spawn a new thread, `det_create` first calls `wait_for_turn` to wait for the thread's deterministic logical clock to become the global minimum. This ensures that all other threads will either be executing private work or waiting on the spawning thread. Then `det_create` sets up the global structures for the new thread and sets the new thread's deterministic logical clock to be one larger than the thread performing the spawn. Finally, `det_create` spawns the new thread and ends the spawning thread's turn.

4.3 Lazy Reads

Many programmers use unprotected or racy reads to spin on flags, or to track the progress of monotonically increasing/decreasing values. The typical example of the latter is an application that stores a global "best" value that many threads repeatedly check. If the thread finds a new best value it acquires the lock and updates the global best. Acquiring the lock to check against the global best is needlessly expensive and therefore undesirable if the application can tolerate reading a value that is out of date. This type of access causes a data race and introduces non-determinism.

To accommodate this programming style we have created an API for deterministically reading unprotected data in a lazy manner. Semantically, a lazy read instruction can be executed without acquiring a lock, but a lazy write instruction must be executed from within a lock. The value returned from a lazy read is deterministic. To maintain performance, each lazy read has a user-defined tolerance window. A larger tolerance will make the lazy read faster, at the expense of returning an older value.

We implement deterministic lazy read support using a combination of two techniques: global write history caching, and local read caching. For write caching we maintain a history table of past written values along with the deterministic logical times at which the writes occurred. When a thread performs a read, it subtracts the user-specified tolerance from its deterministic logical clock to obtain a read deterministic logical time and waits until all other threads have progressed beyond this time. At this point, the thread is guaranteed that no new values can be written with deterministic logical times less than or equal to the read deterministic logical time. As a result, the thread can safely lookup the table to find the most recent write that occurred before the read deterministic logical time. To further improve read performance each thread caches its previous reads for a certain amount of deterministic logical time, which reduces communication and contention on the history table. Local caching makes the semantics of our lazy read API subtly different from a normal racy memory access. In practice, we found that this difference was easy to reason about and of no consequence, for the racy reads we converted in our benchmark applications.

4.4 Application Programming Interface

To make transitioning to Kendo as simple as possible, we have developed Kendo to support a deterministic subset of the POSIX Threads API. We additionally provide the functions `det_enable` and `det_disable` to allow the user to pause a thread's deterministic logical clock during code that they wish to run without Kendo's deterministic guarantee.

Functions are given names beginning with "det_" rather than "pthread_" to allow both Kendo and pthreads to coexist within the same program. We provide a header file that makes the necessary `#define` statements for existing pthreads code to use Kendo without modification.

The lazy read API consists of the following three functions:

- `det_lazy_init`: initializes a given lazy variable using a given initial value, acceptable delay (in deterministic logical time), and a protecting mutex. The acceptable delay indicates the user's tolerance for `det_lazy_read` returning stale values. A higher acceptable delay will cause reads to run faster (because of less synchronization), but they may return older values from the history. The protecting lock is the lock that the user must acquire before calling `det_lazy_write`.
- `det_lazy_read`: reads from a given lazy variable. Uses the lazy variable's history to return a deterministic value with minimal synchronization. Can be called *without* holding the protecting lock.
- `det_lazy_write`: writes to a given lazy variable. Must be called while holding the protecting lock. Properly updates the history to allow deterministic lazy reads.

Calls to thread safe library functions, such as `malloc`, must be handled specially to avoid introducing non-determinism. When called concurrently, such functions may execute with a non-deterministically amount of stores affecting the determinism of the logical clocks. Kendo provides a custom wrapper around `malloc` that disables the deterministic logical clocks during the function call. We provide similar wrappers around a small number of other `libc` functions with non-deterministic store counts. Additionally, we provide a custom pseudo random number generator that uses thread local data to make the values returned in each thread deterministic for a given seed.

5. Evaluation

In this section we evaluate Kendo on a number of parallel applications to show the practicality of our approach. Additionally, we show the effect of varying the performance counter sampling frequency and compare the performance of our lazy read API to using deterministic locks.

5.1 Experimental Framework

Tests were conducted on a 2.66GHz Intel Core 2 Quad-core CPU running Debian "sid" GNU/Linux with kernel version 2.6.23. The kernel was modified to enable the use of hardware performance counters to construct the deterministic logical clocks. In all tests four threads were executed utilizing all available cores.

5.2 Methodology

We converted all programs to use Kendo's API using a conversion process that was simple in our experience. Locks were converted automatically by renaming the calls to the lock library. Racy reads were identified by looking for simple patterns such as volatile declarations, and modified to use our API. Racy reads that executed frequently were converted to use the Kendo lazy read API, while infrequent racy reads were protected with locks. All racy writes were protected with locks. This process took approximately one day for the whole SPLASH-2 benchmark suite.

All applications have been experimentally verified to run deterministically under Kendo, both in output (which was otherwise non-deterministic in some benchmarks) and number of stores (which was otherwise non-deterministic in all benchmarks). The results were particularly remarkable for

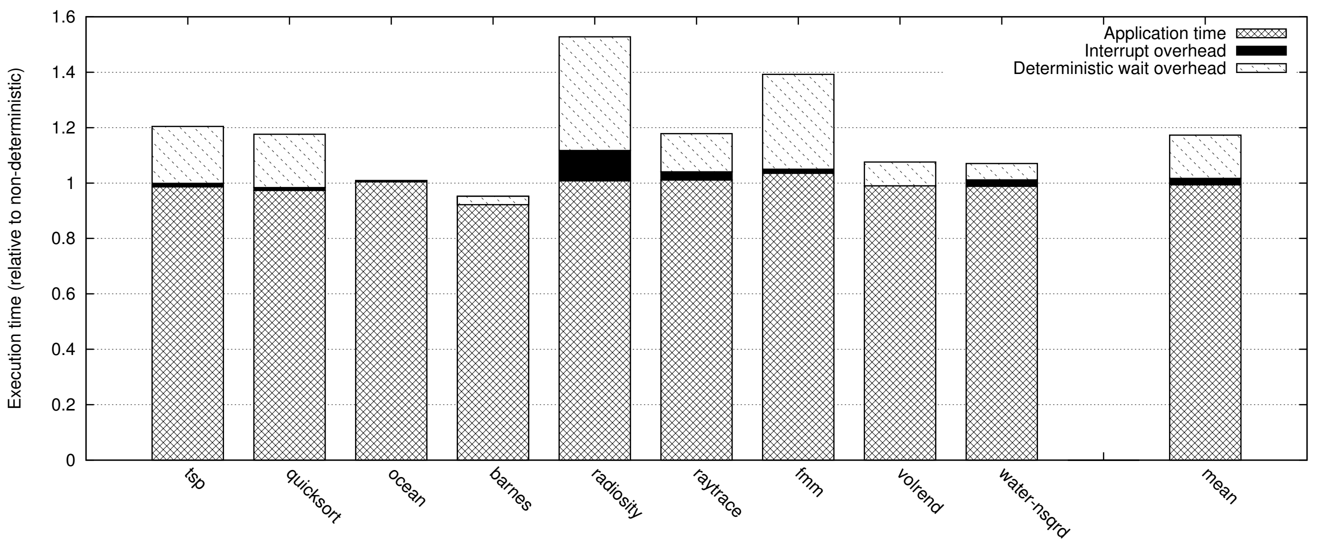

Figure 6. Performance of applications running deterministically under Kendo relative to their non-deterministic performance.

Benchmark name	Chunk size	Locks/s	Barriers/s	Lazy reads/s	Stores/s
tsp	6,000	10.0	0	4,982,115.1	931.4 M
quicksort	6,200	320,915.2	0	0	6,680.7 M
ocean	4,000	279.3	1,220.7	0	391.6 M
barnes	20,000	96,745.0	11.8	0	5,565.4 M
radiosity	2,500	939,771.1	47.4	0	9,268.8 M
raytrace	800	216,979.5	9.1	0	772.6 M
fmm	1,000	208,880.8	450.3	3,700,407.3	1,093.0 M
volrend	2,000	79,612.8	204.3	0	560.2 M
water-nsqrd	7,000	143,202.6	1,843.1	0	7,002.7 M

Table 1. Chosen chunk size for each application along with synchronization events and stores per second.

Radiosity, which produced wildly non-deterministic output. We checked the correctness of our approach by manually verifying the outputs of each of the benchmarks.

All timing tests were run 10 times and the mean value is shown. Times are presented as a percentage of non-deterministic (pthreads) execution time. We break each timing bar into three pieces:

- *Application time* is the baseline time to execute the user application. It consists of all deterministic execution time not spent in interrupt or deterministic wait overhead.
- *Interrupt overhead* is cost incurred by performance counter interrupts used to construct the deterministic logical clocks. This varies both by the frequency of stores in the user application and by the Kendo chunk size for that application.
- *Deterministic wait overhead* is the additional overhead, compared to non-deterministic locks, incurred in locking code, caused by enforcing a deterministic order on the user application. It is dominated by time spent in `wait_for_turn`, but also includes other overhead such as the time spent in system calls pausing and resuming the deterministic logical clocks.

5.3 Experimental Results

Figure 6 presents the performance of Kendo running various applications deterministically. Ocean, barnes, radiosity, raytrace, fmm, volrend, and water-nsqrd are taken from the SPLASH-2 (Woo et al. 1995) benchmark suite. Additionally, we implemented a parallel traveling salesman (TSP) micro-benchmark, based on a sequential version by Lionnel Maugis, and a parallel quicksort that was based on sequential code from the SGI Standard Template Library. On these benchmarks, Kendo incurs a geometric mean of only 16% overhead when running the applications deterministically.

Kendo's performance on each application can be most easily explained by the frequency of synchronization shown in Table 1. Applications requiring more synchronization incur higher overheads than those requiring less synchronization. Radiosity is a highly lock-intensive application, performing close to one million lock acquisitions per second.

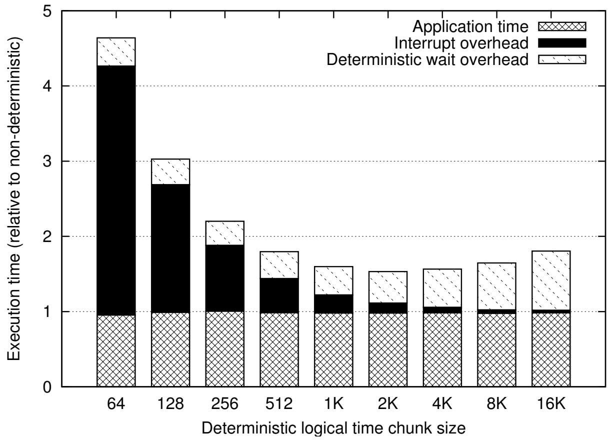

Figure 7. Performance of Radiosity with varying deterministic logical clock chunk size. The chunk size determines the number of stores between each increment of a thread's deterministic logical clock.

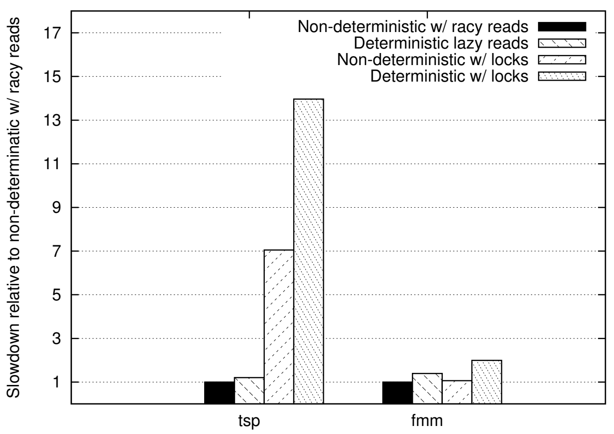

Figure 8. Performance of Kendo's lazy read API compared to locks for TSP and FMM.

As a consequence, it incurs the largest overhead for any benchmark, approaching 53%. On the other hand, Barnes, our best performing application, exhibits a 5% increase in performance. Barnes operates primarily in two phases, one that uses locks and a second where threads operate independently. Profiling reveals that the first phase takes longer when run under Kendo, while the second phase operates faster. We suspect that this is due to improved locality that results from a different interleaving of lock acquires.

Figure 7 illustrates the trade-off between interrupt overhead and deterministic wait overhead as chunk size is varied. It is shown using Radiosity, our slowest benchmark. All applications have a similar trade-off, though there is some variation in the optimal chunk size between applications. Table 1 shows the chunk size used for each application.

Figure 8 shows the performance of Kendo's lazy reads compared to locks for the two benchmarks that utilize lazy reads. The lazy reads perform significantly better than deterministic locks, especially for TSP. This performance comes at the cost of returning less up to date values. Because of this, using lazy reads is most effective when applications poll a value at a high frequency to check if an event has occurred. This is the case with TSP, where each thread frequently polls a global "best" value that changes infrequently.

6. Scalability

Although the focus of this work is not a study of the scalability of our algorithm, the reader may have some concerns about the communication requirements of our turn ordering approach as well as the serialization it causes. Here we present a number of solutions that we are currently exploring.

First, we expect to hide a significant amount of wait overhead by executing applications with more threads than processors and time multiplexing between threads. Under such an approach, threads can yield their time slice at synchronization points if they have progressed faster in deterministic logical time than other threads. This enables a processor to perform work when it would otherwise be waiting for other threads to catch up in deterministic logical time. Additionally, to reduce the communication cost needed to determine a thread's turn, turn ordering can be performed using software combining trees that are similar in vein to the trees used by scalable software barriers.

In addition, future thread-level speculation or best effort transactional memory hardware support could be employed to improve parallelism by optimistically executing the serial portion of the locking algorithm and the critical section that follows. Further hardware support could also eliminate the performance counter sampling overhead currently required by our framework. For example, memory mapped and remotely accessible performance counters could be used directly as each thread's deterministic logical clock.

7. Related Work

Concurrent to our work, Devietti *et al.* have also made a case for deterministic execution of shared memory parallel processor (Devietti et al. 2008). The work defines a deterministic execution model that matches our definition of strong determinism. The authors present a number of hardware designs that can enforce this level of determinism. A first design serializes all memory operations by passing a token in a round robin manner between processors. A processor is required to hold this token to perform a memory access. This algorithm is extended by increasing the number of operations performed by each processor while holding the token, collectively calling each group of operations a *quantum*, and by allowing quanta to execute in parallel whenever they access private memory. A dynamic and deterministically updated sharing table is used to determine what data is private and shared. Finally, a third design leverages thread-level speculative to further improve performance. While this work offers a number of solutions for hardware implementations of strong determinism, such systems are not available today. To the best of our knowledge, our system is the only one that provides weak determinism in current commodity systems.

Also related, preliminary work by Bocchino *et al.* argues that object oriented languages such as Java and C# should be augmented to provide a deterministic execution model by default (Bocchino et al. 2008). The work proposes adding effect system annotations to Java to enable static and dynamic analysis that can detect conflicting accesses before they occur, so that they can be serialized in a deterministic manner. When such annotations or analysis become impracticable, the authors suggests leveraging thread-level speculation hardware.

Record/replay systems for parallel applications can be used to help programmers reproduce non-deterministic application behavior (Wittie 1989; Dhamija and Perrig 2000; Xu et al. 2003; Dunlap et al. 2008; LeBlanc and Mellor-Crummey 1987; Russinovich and Cogswell 1996; Montesinos et al. 2008; Hower and Hill 2008). A notable related example is the pico-log version of the DeLorean record/replay system (Montesinos et al. 2008). DeLorean uses thread-level speculation hardware to efficiently enforce a round-robin interleaving of fixed-size chunks of instructions. Unfortunately, thread-level speculation cannot guarantee that the speculative working sets of each chuck can fit into the L1 data cache. As a result, DeLorean uses a record run that logs the locations and size of prematurely truncated chunks.

Multithreaded replica systems add fault tolerance by executing many replicas of a program so that nodes may fail without interruption in service (Basile et al. 2002; Domaschka et al. 2006, 2007; Saha and Dutta 1993; Karl et al. 1998; Reiser et al. 2006). This type of technique relies on determinism so that replicas remain synchronized. These systems offer a variety of techniques to enforce this synchronization.

Some programming language designs have explicitly provided a deterministic programming model. For example, languages such as StreamIt (Thies et al. 2002) eliminate non-determinism by using a streaming model for thread communication. Unfortunately, StreamIt's design choice limits it to a specific class of applications that can use streaming semantics. Programming languages such as Cilk (Frigo et al. 1998) offer deterministic guarantees for a subset of legal programs. For lock free programs or for programs that only perform commutative operations in lock protected critical sections, Cilk's Nondeterminator race detector tool offers a determinism guarantee for any inputs tested (Cheng et al. 1998).

Finally, a large body of research has focused on making parallel applications easier to debug (Lu et al. 2007b; Tucek et al. 2007; Lu et al. 2007a, 2006). These techniques focus directly on finding non-deterministic bugs without removing the non-determinism itself.

8. Conclusions

In this paper we have presented Kendo, the first efficient and practical system to provide weak determinism for parallel applications. When combined with a race detector, Kendo can provide a systematic way of reproducing many non-deterministic bugs in shared memory multithreaded applications. Like software transactional memory, Kendo will allow researchers and developers to gain early hands-on experience with deterministic multithreading programming models, and to develop a body of code that will support future research. We have evaluated Kendo on the SPLASH-2 benchmark suite and shown performance results that incur a geometric mean slowdown of only 16%. Such low overheads make the testing and debugging benefits of weak determinism accessible to developers today.

9. Acknowledgements

We acknowledge William Thies for inspiring the original idea, numerous helpful discussions, and for providing feedback on manuscripts. We would additionally like to thank Micheal I. Gordon and the anonymous reviewers for their helpful suggestions. This research is supported by NSF ITR ACI-0325297, DARPA FA8650-07-C-7737, AFRL FA8750-08-1-0088 and the Gigascale Systems Research Center.

References

Claudio Basile, Keith Whisnant, Zbigniew Kalbarczyk, and Ravi Iyer. Loose synchronization of multithreaded replicas. pages 250–255, 2002.

R. Bocchino, V. Adve, S. Adve, and M. Snir. Parallel programming must be deterministic by default. Technical Report UIUCDCS-R-2008-3012, University of Illinois at Urbana-Champaign, 2008. URL `http://dpj.cs.uiuc.edu/DPJ/Publications_files/paper.pdf`.

Guang-Ien Cheng, Mingdong Feng, Charles E. Leiserson, Keith H. Randall, and Andrew F. Stark. Detecting data races in cilk programs that use locks. In *proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '98)*, pages 298–309, Puerto Vallarta, Mexico, June 28–July 2 1998.

Joe Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. Explicitly parallel programming with shared-memory is insane: At least make it deterministic! In *proceedings of SHCMP 2008: Workshop on Software and Hardware Challenges of Manycore Platforms*, Beijing, China, June 22 2008.

Rachna Dhamija and Adrian Perrig. Déjà vu: a user study using images for authentication. In *SSYM'00: Proceedings of the*

9th conference on USENIX Security Symposium, pages 4–4, Berkeley, CA, USA, 2000. USENIX Association.

Jorg Domaschka, Franz J. Hauck, Hans P. Reiser, and Rutiger Kapitza. Deterministic multithreading for java-based replicated objects. In *proceedings of the International Conference on Parallel and Distributed Computing and Systems*, 2006.

Jorg Domaschka, Andreas I. Schmied, Hans P. Reiser, and Franz J. Hauck. Revisiting deterministic multithreading strategies. *International Parallel and Distributed Processing Symposium*, pages 1–8, 2007.

George W. Dunlap, Dominic G. Lucchetti, Michael A. Fetterman, and Peter M. Chen. Execution replay of multiprocessor virtual machines. In *VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 121–130, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-796-4.

Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 212–223, Montreal, Quebec, Canada, June 1998. Proceedings published ACM SIGPLAN Notices, Vol. 33, No. 5, May, 1998.

Derek R. Hower and Mark D. Hill. Rerun: Exploiting episodes for lightweight memory race recording. In *proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA)*, June 2008.

Wolfgang Karl, Markus Leberecht, and Michael Oberhuber. Forcing deterministic execution of parallel programs - debugging support through the smile monitoring approach. In *proceedings of the SCI-Europe*, September 1998.

Milind Kulkarni, Keshav Pingali, Ganesh Ramanarayanan, Bruce Walter, Kavita Bala, and L. Paul Chew. Optimistic parallelism benefits from data partitioning. *SIGARCH Comput. Archit. News*, 36(1):233–243, 2008. ISSN 0163-5964.

Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978. ISSN 0001-0782.

T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Trans. Comput.*, 36(4):471–482, 1987. ISSN 0018-9340.

Edward A. Lee. The problem with threads. *Computer*, 39(5):33–42, May 2006. ISSN 0018-9162.

Shan Lu, Joe Tucek, Feng Qin, and Yuanyuan Zhou. Avio: Detecting atomicity violations via access-interleaving invariants. In *proceedings of the International Conference on Architecture Support for Programming Languages and Operating Systems*, October 2006.

Shan Lu, Weihang Jiang, and Yuanyuan Zhou. A study of interleaving coverage criteria. In *proceedings of ESEC-FSE*, pages 533–536, New York, NY, USA, 2007a. ACM. ISBN 978-1-59593-811-4.

Shan Lu, Soyeon Park, Chongfeng Hu, Xiao Ma, Weihang Jiang, Zhenmin Li, Raluca A. Popa, and Yuanyuan Zhou. Muvi: Automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *proceedings of the 21st ACM Symposium on Operating Systems Principles*, October 2007b.

Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *ASPLOS XIII: proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 329–339, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-958-6.

Pablo Montesinos, Luis Ceze, and Josep Torrellas. Delorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In *proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA)*, June 2008.

Hans P. Reiser, Jorg Domaschka, Franz J. Hauck, Rudiger Kapitza, and Wolfgang Schroder-Preikschat. Consistent replication of multithreaded distributed objects. *25th IEEE Symposium on Reliable Distributed Systems*, pages 257–266, 2006. ISSN 1060-9857.

Jonathan Rose. Locusroute: a parallel global router for standard cells. In *DAC '88: Proceedings of the 25th ACM/IEEE conference on Design automation*, pages 189–195, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press. ISBN 0-8186-8864-5.

Mark Russinovich and Bryce Cogswell. Replay for concurrent non-deterministic shared-memory applications. In *proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, pages 258–266, New York, NY, USA, 1996. ACM. ISBN 0-89791-795-2.

Debashis Saha and Sourav K. Dutta. Specification of deterministic execution timing schema for parallel programs on a multiprocessor. *Computer, Communication, Control and Power Engineering*, pages 114–116 vol.1, 1993.

Jaswinder Pal Singh, Anoop Gupta, and Marc Levoy. Parallel visualization algorithms: Performance and architectural implications. *Computer*, 27(7):45–55, 1994. ISSN 0018-9162.

William Thies, Michal Karczmarek, and Saman Amarasinghe. Streamit: A language for streaming applications. In *International Conference on Compiler Construction*, Grenoble, France, April 2002.

Joseph Tucek, Shan Lu, Chengdu Huang, Spiros Xanthos, and Yuanyuan Zhou. Triage: diagnosing production run failures at the user's site. In *proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, pages 131–144, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-591-5.

Larry D. Wittie. Debugging distributed C programs by real time reply. *SIGPLAN Not.*, 24(1):57–67, 1989. ISSN 0362-1340.

S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *proceedings of 22nd Annual International Symposium on Computer Architecture News*, pages 24–36, June 1995.

Min Xu, Rastislav Bodik, and Mark D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. *SIGARCH Comput. Archit. News*, 31(2):122–135, 2003. ISSN 0163-5964.