Increasing Concurrency in Deterministic Runtimes with Conversion

Timothy Merrifield
University of Illinois at Chicago
tmerri4@uic.edu

Jakob Eriksson
University of Illinois at Chicago
jakob@uic.edu

ABSTRACT

Experimental results are presented for several benchmark programs, identifying quantum size imbalance as a major source of inefficiency in Dthreads. A two-pronged approach is proposed to address this problem. First, Dthreads is ported to a *versioned memory* subsystem, so that the fence may be removed. Second, we find that Dthreads' round-robin token order is more rigid than necessary for the revised memory model. We propose a more efficient, yet determinism-preserving order based on approximation of program execution time using a determinstic clock.

1. INTRODUCTION

Order is not always important, but when it is, it must not change. This statement captures the essence of deterministic concurrency. Prior work differs both in when order matters, and in what order to enforce, but the above statement holds for all. In this paper, we study its performance impact on a recent deterministic concurrency runtime (Dthreads [13]), and present ways to (a) significantly reduce the proportion of time when execution order is enforced, and (b) compute a more expedient, yet still deterministic, order of execution.

Recent work in deterministic concurrency has seen numerous proposals with the goal of making determinism a practical reality on concurrent systems. These include language-based techniques [6, 7], scheduler memoization [8], run-time enforcement [3, 13, 15, 11], and a combination of run-time enforcement and language constructs [10]. Some run-time enforcement strategies have focused on a limited programming model [4, 2], while others have aspired to be fully functional, drop-in replacements for existing threading libraries [13, 3, 15]. Given that this latter category can support a large portion of existing software, these are a particularly attractive choice for deterministic execution in the short-term.

Dthreads [13] wraps the popular `pthreads` library, making the process of turning a `pthreads` program into a deterministic program as easy as replacing a compiler flag. However, while Dthreads is a formidable feat of engineering demonstrating impressive performance, benchmarks still show several deterministic programs being as much as 4–5.5× slower than their non-deterministic counterparts.

In §2, we study this problem in some detail, and identify one major reason behind the slow-down: when a thread is done with one task (quantum), deterministic execution says that the thread must wait its turn before it may proceed with the next. In Dthreads, depending on what the other threads are doing, that wait can sometimes be rather tedious. Specifically, if one thread has short-duration quanta, and another has long-duration quanta, the former thread will spend most of its time waiting for the latter.

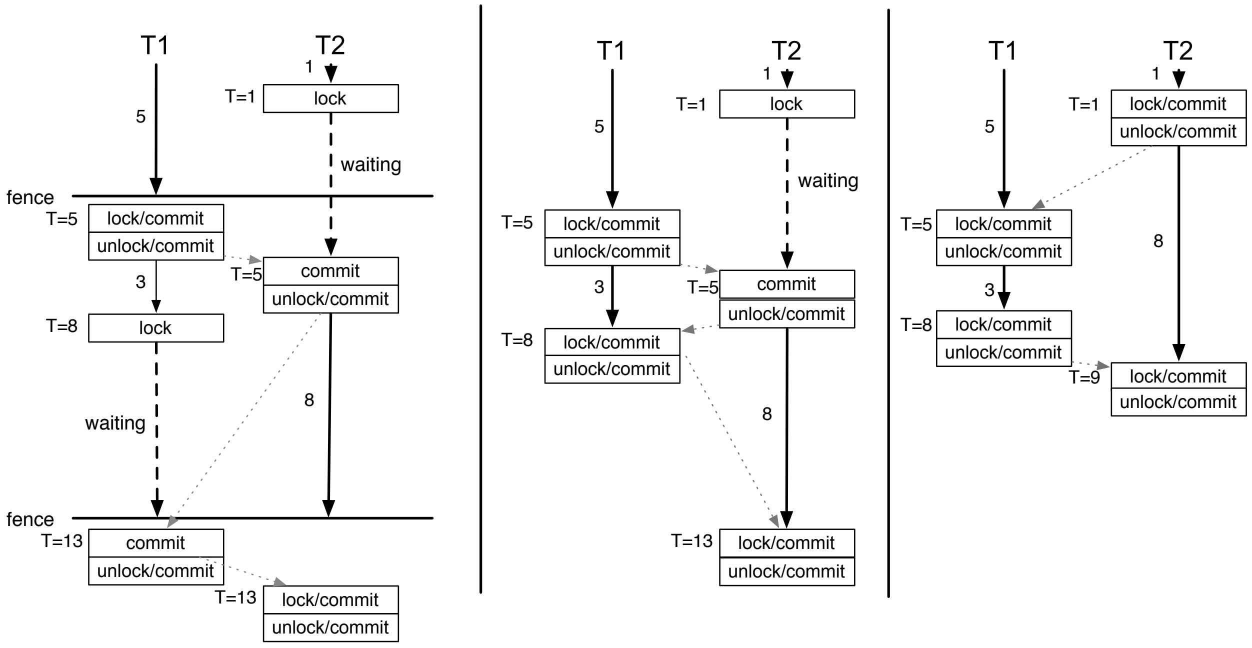
To address this problem, we first relax *when* order matters. We briefly describe CONVERSION (§3.1), a multi-version concurrency control system for main memory, and a port of Dthreads to the CONVERSION memory model. CONVERSION offers complete isolation between threads, which removes the need for the Dthreads fence between transactions. This allows each process to do twice as much work before waiting its turn, resulting in up to 2× improvement in performance.

Second, in §4 we address what is the most expedient order to enforce. A new, deterministic token ordering scheme is proposed, which replaces the current round-robin token order with one based on quantum runtime estimates. This ordering promises to essentially eliminate the slow-down effect identified in §2.

2. QUANTUM LENGTH IMBALANCE IS BEHIND DTHREADS SLOW-DOWN

Dthreads provides a coarse-grained execution model, using `pthreads` operations to mark the end of a parallel phase (quantum) of execution and the beginning of a sequential commit phase. When threads arrive at a pthreads operation, they first wait at the *fence* (a barrier for a variable number of threads) for the arrival of the other executing threads that are active in the current round. The purpose of the fence is to ensure that commits to shared memory are not visible to threads that are still executing in the parallel phase. Similar to the COREDET approach, threads commit their changes to shared memory during the sequential phase, in a round-robin fashion. In Dthreads, this ordering is preserved using a *token* which is deterministically passed between threads.

By using pthreads operations to bookend quanta, Dthreads provides clear semantics—execution behavior is only based on the placement of pthreads function calls. However, as these quanta are of undetermined length, Dthreads quanta cannot exhibit the even lengths that COREDET achieves by



(a) Original Dthreads

(b) Dthreads w/ Conversion
Round-robin token passing

(c) Dthreads w/ Conversion
ideal token order

Figure 1: Thread execution timing diagrams for three different execution models. (a) Shows the original model of Dthreads, (b) describes Dthreads with Conversion and (c) still utilizes Conversion but with a better token passing technique. The goal here is to reduce waiting (shown with dashed lines) and keep threads busy.

delimiting quanta based on estimated execution times.

Fig. 1(a) illustrates this problem more concretely. Here, Dthreads is executing with two threads, T1 and T2, with T1 holding the token initially. At time 1 (T=1), thread T2 arrives at a pthreads operation and now waits at the fence for T1. Thread T1 arrives at the fence at time 5 and commits its changes, executes a short critical section (taking no time in this example) and passes the token to thread T2 which does the same. At time 8, T1 performs another pthreads operation and begins waiting for T2 to arrive at the fence, which finally happens at time 13.

This simple example reveals a challenge for deterministic runtimes with significant variance in quantum length. In the example in Fig. 1, the threads spend approximately 30−40% of their total execution time just waiting at the fence. This problem is further intensified as more threads are added and the time spent in the sequential phase is factored in.

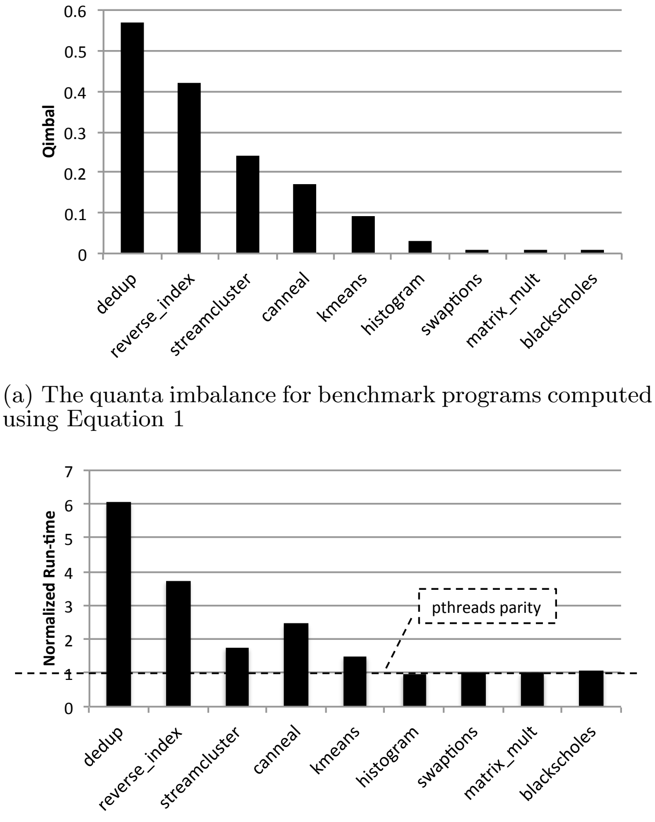
2.1 Quantum Length Imbalance in Benchmark Programs

To better quantify the impact of quantum length imbalance, we studied the existing imbalance in several benchmark programs, to see how this relates to overall Dthreads performance. We define the quantum length imbalance for a given Dthreads parallel round r as,

$$Qimbal_r = \frac{1}{N-1}\sum_{q=1}^{N}\frac{len_{max} - len_q}{len_{max}}, \qquad (1)$$

where N is the number of threads in the round and max is the longest quantum for that round. Intuitively, $Qimbal$ expresses the mean fraction of each round that a thread, other than the longest-quantum thread, spends waiting at the barrier. Fig. 2(a) shows the mean quantum imbalance over all rounds for Dthreads on a subset of the Parsec [5] and Phoenix [16] benchmark suites. For example, the benchmark "dedup" shows a quantum imbalance of 0.58, meaning that on average, every thread but one was spending 58% of its time in the parallel phase waiting at the barrier. In other words, with 4 threads and 4 cores, one thread was running at 100% while the other three on average were running at 42% of capacity, for a total effective utilization of $\frac{1+.42+.42+.42}{4} = 56\%$. This is without taking into account any other inefficiencies present in Dthreads.

For comparison purposes, Fig. 2(b) provides the execution time of these benchmarks, normalized to the pthreads execution time. Here the striking similarity suggests that quantum length imbalance is one of the main factors, and perhaps even the dominant factor in the observed Dthreads slow-down. One noticeable exception is the `streamcluster` program. In looking through this program, we found that `streamcluster` makes heavy use of barriers, effectively creating a fence in the pthreads version as well. In essence, Dthreads was not unusually efficient in this benchmark; instead pthreads was unusually inefficient.



(a) The quanta imbalance for benchmark programs computed using Equation 1

(b) The execution times for Dthreads, normalized to the execution times for pthreads.

Figure 2: The relationship between quanta imbalance and Dthreads performance.

3. REVISED MEMORY MODEL

From the discussion above it is clear that quantum imbalance leads to threads waiting at the fence and results in reduced performance. Because the fence exists to ensure that commits to shared memory aren't made visible to threads still executing a quantum, a revised memory model that allowed commits but shielded their results from executing threads would allow us to *remove the fence entirely.* To support this functionality, we replace the memory model of Dthreads with Conversion [14].

3.1 A Conversion Primer

Conversion provides support for multi-version concurrency control for main memory segments. Conversion semantics are similar to a version control system, where each process operates on a *working copy* of the segment, isolated from updates by other processes. This technique is similar to the CVS-based version-controlled memory described in [9]. A process can retrieve any updates to the trunk by calling `update()` and a call to `commit()` pushes any local changes to the trunk. Like the original Dthreads memory model, Conversion utilizes virtual memory protection to provide isolation.

However, unlike Dthreads, Conversion is built as a Linux kernel module and operates directly in kernel space. After an application marks a **mmap()**'d segment as Conversion-enabled, copy-on-write page faults are passed to the Conversion module by the Linux page fault handling code. Conversion tracks these faults and on `commit()`, creates a new version, causing subsequent writes to trigger page faults. A call to `update()` retrieves the latest version, performing a batch update of all required page table entries.

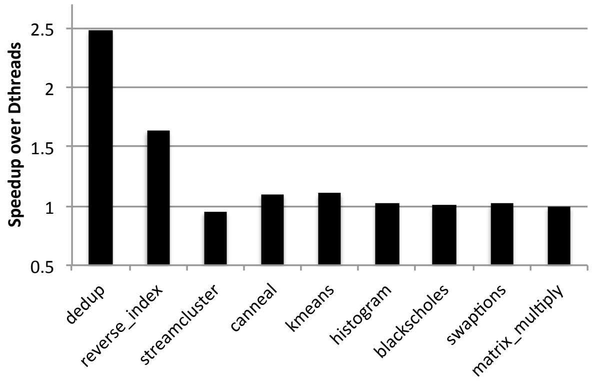


Figure 3: Speedup of over Dthreads when using Conversion with the fence removed.

The fact that Conversion resides in the kernel allows it to handle page faults without signaling user space, which improved the latency of copy-on-write page faults by 2x over Dthreads in our experiments. Additionally, Conversion performs `update()` operations by directly updating process page tables, which provides increased isolation over Dthreads. Unlike the CVS-based memory [9], Conversion is *very* fast, performing per-page operations in less than 1 μs (with a constant overhead of 2-6 μs). Additionally, using a Conversion memory model reduces the memory management code in Dthreads by over 80%.

3.2 Removing the Fence

With Conversion in place, we can now remove the fence and rely on token order to preserve determinism. The new execution model is shown in Figure 1(b). Again, T2 invokes a pthreads operation at time 1 but this time is not waiting at the fence but instead is waiting on token acquisition. T1 arrives at time 5 and acquires the token, executes its critical section and commits. After T2 commits, T1 arrives again at time 8 but this time *commits immediately* and exits. Finally T2 arrives at time 13 and commits. In this example, T1 finished 5 time units earlier, reducing the total time spent waiting from 9 time units to 4. The impact of this improved concurrency is further discussed in [14] and the results are summarized in Figure 3.

While removing the fence can improve the performance of benchmarks that show quanta imbalance, we note that T2 is still spending 4 time units waiting for the token, which is passed in round-robin fashion. Clearly, a different ordering may produce better results. It is important to note here that the order of acquisition remains deterministic as long as acquisition happens in the same order for each execution of the program. So if we can deterministically enforce a better order, then we can improve performance. This insight drives our discussion of token ordering below.

4. IMPROVED TOKEN ORDERING FOR GREATER CONCURRENCY

Prior to using Conversion, the order in which the token was passed between threads was a non-issue, as a thread could not continue beyond the fence even if it owned the token. Therefore, a simple round-robin token passing scheme was sufficient for the original Dthreads. However, with Conversion a better token order can substantially increase concurrency. We return to Figure 1(b), where the fence-less execution uses a round-robin token passing order. In this example, thread T2 arrives at a synchronization point first after performing an operation of cost 1. However, using the round-robin token passing it must wait for T1 to arrive and commit before it can acquire the token. T1 arrives at time 5, commits and passes the token to T2. Notice that this is still a fence-less strategy, as T1 continues to execute its next quantum while T2 performs its commit. T1 then commits at time 8 while T2 later commits at time 15.

However, the round-robin token order is arbitrary and could be substantially improved. The result of the alternative token order { T2, T1, T1, T2 } is shown in Figure 1(c). Upon arrival at time 1, T2 immediately acquires the token, commits and continues on to execute the next quantum. T1 also acquires the token without waiting and performs a commit at time 5. In this version of events no waiting is necessary and the program terminates at time 9.

The basic strategy for producing an expedient token order is to always give the token to the thread that needs it first. Unfortunately, token order cannot simply be based on arrival time, as this is inherently non-deterministic. Instead, we must find an alternative, *deterministic clock* that deterministically approximates execution time. This is similar to the approach described in [15]. Using such a clock, we modify the token acquisition process as follows: a token acquisition by a thread is only successful if that thread has the lowest *deterministic time* value of all "live" threads in the system (by "live", we mean processes that are not waiting on an event such as a join or a conditional variable). If this criterion is not met, then the thread must wait for the condition to be true. The challenge is to find a technique that can provide an accurate estimate of execution time, while remaining deterministic.

5. A TICK-BASED TOKEN PASSING SCHEME

To implement the *deterministic clock* that we describe above, we approximate execution time using "clock ticks" deterministically triggered in suitable locations throughout the program. Our system maintains a tick-counter stored in a global array $ticks[]$, indexed using a thread's *threadid*. The program code is instrumented with calls to *tick_add*(*int*), which increases the tick-counter for the given process. When the process attempts to acquire the token, it is successful if it has the lowest tick-count of all the "live" processes in the system. Intuitively, if it has the lowest count, no thread can ever have a lower count in the future. Since counts are deterministically increased, token acquisition based on the lowest count is also deterministic. We describe the details of our technique below, however we leave evaluation of this technique for future work.

5.1 Notification of Waiting Threads

A thread attempting to acquire a token may not have the lowest tick-count, but in the future it may become the lowest as other threads continue to execute. For example, if T1 and T2 have tick-counts of 50 and 10 respectively and T1 attempts to acquire the token, it clearly must wait. If T2 continues to execute and reaches a tick-count greater than 50, then it is now safe for T1 to commit. Therefore, we need a process by which T2 can notify T1 that it now has the lowest tick-count in the system.

To facilitate this notification, we add two additional fields to each object in the $ticks[]$ array, $notify\_tick\_count$ and *notify_thread_id*. Returning to our example, when T1 attempts to acquire the token it notices that a running thread (T2) has a lower tick-count. It sets the $notify\_tick\_count$ for T2 to be 50, sets the *notify_thread_id* to itself (T1) and goes to sleep. As T2 continues to execute, with each call to *tick_add*() it checks to see if its tick-count has exceeded the value of $notify\_tick\_count$. If it has, it now wakes up the thread T1 so that it can acquire the token, and sets $notify\_tick\_count$ to $LONG\_MAX$.

5.2 Avoiding Thread Starvation

Threads that are waiting on pthreads events (including conditional variables and joins) pose a unique challenge because they are no longer increasing their tick-count. Once a thread T1 wakes up from such an event, they can potentially have a far lower tick-count than all other threads in the system. This can cause starvation, as other threads cannot acquire the token until T1's tick-count increases beyond their own.

In order to avoid the starvation issue, once a thread T1 wakes up from such an event they acquire the token normally, but are now forced to increase their tick-count to the tick-count of the last thread that acquired the token (notice that this is the tick-count *when* it acquired the token, not its current tick-count). This essentially resets T1's tick-count and ensures that it is not drastically lower than others in the system. Given that the pthread event that woke up the thread T1 is deterministic, and the prior commit ordering was deterministic, we maintain determinism.

5.3 Instrumentation of Programs

We currently instrument simple programs by hand but we envision future implementations using compiler support to insert calls to *tick_add*(). As was described in Coredet [3], the time between calls to *tick_add*() is a trade-off between the granularity of our deterministic clock, and overhead due to increased instrumentation. Assuming accurate tick-counts, a larger number of calls to *tick_add*() increases the possibility of notification to a waiting thread.

To provide accurate ticks, there are two major approaches. The first is to use the compiler techniques described in [3]. This would provide a per-instruction estimate during compilation and allow us to generate calls to *tick_add*() with an estimated value of execution time. A problem with this approach involves how to handle calls to external libraries such as libc. Coredet, dependent on compiler instrumentation for correctness, simply executes external libraries sequentially given that they are not instrumented. However, the *correctness* of our scheme is not adversely affected by calls to libraries that are not instrumented with calls to *tick_add*(). We would only suffer a performance penalty in this case. For some well-known libc functions it may be possible to provide instrumentation that estimates, to some reasonable degree of accuracy, the latency of these operations.

An alternative approach is to use hardware performance counters to build a deterministic clock. However, as described in Kendo [15], many of these counters are non-deterministic and therefore cannot be used to provide a highly accurate deterministic clock implementation. Kendo uses only the retired instruction count, which the authors deem (through experimentation) to be deterministic.

Another approach is to use both of these techniques. For code compiled under our control, we instrument with calls to *tick_add*(), utilizing per-instruction latency estimates. When a program makes a call to an external library, we insert a call to *tick_add*() just after the function that will base its tick-counter increase on the retired instruction count.

6. RELATED WORK

The seminal work in Coredet [3] provided the formulation of many of the original concepts associated with deterministic runtime systems. In [3], there is a thorough discussion of quanta formation and the importance of balance to maintaining concurrency. The Coredet runtime is further expanded upon in [11] and [10].

Dthreads improved the performance of Coredet by removing the instrumentation for loads and stores and replacing it with virtual memory protection. This memory model is further described in Grace [4] and Sheriff [12]. Kendo [15] uses a deterministic clock to provide order to synchronization mechanisms, but only supports programs that are already race-free.

Conversion shares similarities to other deterministic consistency models such as Hobbes version-controlled memory [9] and Workspace Consistency (WC) [1]. While Hobbes introduced the concept of version controlled shared memory, no implementation or evaluation was provided. And unlike WC, Conversion does not enforce up-front ordering of thread communication but does allow this to be implemented at higher levels (as described here and in [14]).

7. CONCLUSION

We have presented a discussion and analysis of concurrency issues within the Dthreads deterministic runtime system. We have shown through benchmark programs that quanta imbalance is the major source of this problem and that the use of an alternative, Conversion-based memory model can improve performance. Further, we discussed an alternative token ordering technique that allows threads to acquire the token based on arrival at synchronization points. We believe this will greatly improve concurrency over the current round-robin approach, however future implementation and evaluation is necessary.

8. ACKNOWLEDGMENTS

This material is based upon work supported by the U.S. National Science Foundation under Grants CNS-1017877 and CNS-1149989.

9. REFERENCES

[1] A. Amittai, B. Ford, and Y. Zhang. Workspace Consistency: A Programming Model for Shared Memory Parallelism. 2nd WoDet, 2011.

[2] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient System-Enforced Deterministic Parallelism. OSDI'10, 2010.

[3] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. CoreDet: A Compiler and Runtime System for Deterministic Multithreaded Execution. *ACM SIGARCH Computer Architecture News*, 38(1):53–64, 2010.

[4] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: Safe Multithreaded Programming for C/C++. OOPSLA '09, 2009.

[5] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *PACT*, PACT '08, 2008.

[6] R. Bocchino Jr, V. Adve, D. Dig, S. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A Type and Effect System for Deterministic Parallel Java. In *ACM SIGPLAN Notices*, volume 44, pages 97–116. ACM, 2009.

[7] S. Burckhardt, A. Baldassin, and D. Leijen. Concurrent Programming with Revisions and Isolation Types. OOPSLA '10, 2010.

[8] H. Cui, J. Wu, C. Tsai, and J. Yang. Stable Deterministic Multithreading Through Schedule Memoization. *9th OSDI*, 2010.

[9] D. Hower and M. Hill. Hobbes: CVS for Shared Memory. In *Workshop on Determinism and Correctness in Parallel Programming*, 2011.

[10] J. Devietti, D. Grossman, and L. Ceze. The Case For Merging Execution-and Language-level Determinism with MELD. In *Workshop on Determinism and Correctness in Parallel Programming.*

[11] J. Devietti, J. Nelson, T. Bergan, L. Ceze, and D. Grossman. RCDC: A Relaxed Consistency Deterministic Computer. *ACM SIGARCH Computer Architecture News*, 39(1):67–78, 2011.

[12] T. Liu and E. Berger. SHERIFF: Precise Detection and Automatic Mitigation of False Sharing. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, pages 3–18. ACM, 2011.

[13] T. Liu, C. Curtsinger, and E. D. Berger. Dthreads: efficient deterministic multithreading. SOSP '11, 2011.

[14] T. Merrifield and J. Eriksson. Conversion, Multi-Version Concurrency Control for Main-Memory Segments. In *Proceedings of the 8th ACM european conference on Computer Systems*, EuroSys '13.

[15] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: Efficient Deterministic Multithreading in Software. In *ACM Sigplan Notices*, volume 44, pages 97–108. ACM, 2009.

[16] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *HPCA*, pages 13–24, 2007.