Explicitly Parallel Programming with Shared-Memory is Insane: At Least Make it Deterministic!

Joe Devietti, Brandon Lucia, Luis Ceze and Mark Oskin
Department of Computer Science and Engineering
University of Washington

Abstract

The root of all (or most) evil in programming shared memory multiprocessors is that execution is not deterministic. Bugs are hard to find and non-repeatable. This makes debugging a nightmare and gives little assurance in the testing — there is no way to know how the program will behave in a different environment. Testing and debugging is already difficult with a single thread on uniprocessors. Pervasive parallel programs and chip multiprocessors will make this difficulty worse, and more widespread throughout our industry.

In this paper we make the case for fully deterministic shared memory multiprocessing. The idea is to make memory interleaving fully deterministic, in contrast to past approaches of simply replaying an execution based on a memory interleaving log. This causes the execution of a parallel program to be only function of its inputs, making a parallel program effectively behave like a sequential program. We show that determinism can be provided with reasonable performance cost and we also discuss the benefits. Finally, we propose and evaluate a range of implementation approaches.

1. Introduction

History has shown that developing multithreaded software is inherently more difficult than writing single-threaded code. Among the many new software engineering challenges multithreading creates is the fact that even correctly written code can execute non-deterministically: given the same input, threads can interleave their memory and I/O operations in unique ways each time an application executes. This fact seems both obvious and upon further reflection, ridiculous. What kind of programming model can be taken seriously that executes applications differently each time they are run?

Non-determinism in multithreaded execution arises from small perturbations in the execution environment from run to run. These changes include other processes executing simultaneously, differences in how the operating system allocates resources, minor differences in the cache, TLB, bus and resource priority control mechanism states, and differences in the initial micro-architectural states of the processor cores themselves. Non-determinism also enters into program execution from the operating system itself. Even the most basic system calls, such as READ can, *legitimately* return a different result from program run to program run.

Non-determinism complicates the software development process. Defective software might execute correctly hundreds of times before a subtle synchronization bug appears. The current solution to this is based on hope: simply execute the application several times and hope the significant thread interleavings that will occur, have occurred. Test-suites, the bed rock of reliable software development, have diminished value with multithreading. If a program executes differently each time it is run, what does any particular test suite actually test; i.e. what is the coverage? If the program output varies from run to run, is that program wrong, or is it just a different, legitimate outcome? Finally, how can developers have confidence that any particular thread interleavings that occur in a software product deployed in the wild, have been tested in-house?

In this paper we argue that shared memory multiprocessors can be deterministic, and with little performance penalty. To ground the discussion we begin by providing a precise definition for what it means to execute deterministically. A key insight of this definition is that multithreaded execution can be deterministic if the communication between threads is deterministic, and this leaves ample room in the microarchitecture design space in which to achieve deterministic behavior efficiently.

Next, we study what gives rise to non-deterministic execution, and explore just how non-deterministic shared memory multiprocessors actually are. We find, as anyone would have expected, that current generation multicore devices are highly non-deterministic. Program outcomes could diverge from run to run exponentially in the number of instructions they contain. The fact that they tend not to is because of synchronization between threads, and small perturbations in execution at the instruction-by-instruction level tend to cancel each other out over time, creating far less apparent non-determinism than is theoretically possible.

The remainder of this paper presents our study into how to build an efficient deterministic multiprocessor. We propose three basic mechanisms for doing so: (1) using locks to transform multithreaded execution into single-threaded execution, a naive, but useful technique; (2) using cache-line states to control when communication occurs between threads; (3) using transactions to speculatively communicate deterministically between threads.

This study is a limit study of these three techniques, and some useful implementation variations of them. From this study, we find that determinism can, theoretically at least, be had for practically no added performance cost. Stated differently, assuming efficient hardware can be built, then being deterministic in execution will not artificially serialize parallel execution.

2. Background

2.1. A Definition of Deterministic Parallel Execution

At a high level, our goal is to make multi-threaded execution as deterministic as single-threaded execution. Hence, given the same input to a multi-threaded program, it should produce the same output. How can this be achieved? Let us consider all memory and system call operations from all threads merged together into a global ordering. First, its not important what particular ordering is chosen, any ordering that is a valid program execution will do. For a multiprocessor to be deterministic it is only critical that the same ordering is achieved each time the same program is run with the same input.

Second, what about this global ordering makes a program produce the same output given the same input? For instance, if two adjacent memory operations from different threads operate on different memory addresses, can they be swapped in the order and not effect the program outcome? The answer is yes because that swap has no observable effect on thread execution. What turns out to be key for achieving deterministic execution in this global ordering is that each consumer (load instruction) of data read data from the same producer (write instruction). Moreover, I/O operations should be considered as having read and write sets, and in order to operate correctly with the outside non-deterministic environment, should be executed in the same order from program run to program run.

In summary, a multiprocessor is deterministic if two constrains are held: (1) Each dynamic instance of a consumer (load) instruction, regardless of thread, reads data written by the same dynamic instance of a producer (store) instruction, regardless of which thread it is executed in; (2) All system I/O calls occur in a global ordering and are considered both producers and consumers of data to all addresses. This latter constraint is clearly overly broad, but narrowing it is the subject of future work. Such a definition of execution provides for deterministic behavior, and, as we shall see in the next few sections, ample opportunity for efficient implementation.

2.2. Non-determinism in existing systems

An interesting question to ask is what are the sources of non-determinism in the execution environment that give rise to the non-determinism in program output; and how much non-determinism exists, quantitatively, in the execution environment? Here in this section, we attempt address these two questions.

2.2.1. Sources of non-determinism

Multiprocessor systems are non-deterministic in their execution environment for two broad reasons: (1) The software environment changes, from program run to program run; (2) Non-ISA micro-architectural state changes from program run to program run. Both of these effects manifest themselves as perturbations in the timings between events in different threads of execution. As the ordering of small events change, the effect on the application is to ultimately change which dynamic instance of a store instruction produces data for a dynamic instance of a load instruction. Once this occurs, the program execution diverges from previous execution runs at the ISA level, and the program output may vary. Below we enumerate just a few of the software and hardware sources of non-determinism.

Several aspects of the software environment create non-determinism in program output. Among them are: other processes executing concurrently and competing for resources, the state of memory page tables, disk and I/O buffers, and the state of any global speed-heuristic data structures (e.g hash tables) in the operating system. In addition, several operating system calls have interfaces with legitimate non-deterministic behavior. For example, the *read* system call can legitimately take a variable amount of time to complete and return a variable amount of data!

At the hardware level, a variety of non-ISA visible components vary from program run to program run. Among them are: the state of any physically mapped caches, the state of any predictor tables (branch, data, aliasing, etc), the state of any bus priority controllers, any micro-architectural state that may eventually manifest as a timing difference (physical register usage, etc). Certain hardware components, such as bus arbitrators can indeed change their outcome from program run to program purely from environmental factors. If the choice of priority is given on which signal is detected first, the outcome can vary with differing temperature and load characteristics.

Collectively, current generation software and hardware systems are not built to be deterministic. In the next section, we measure just how non-deterministic they are.

2.2.2. Quantifying non-determinism

clear cache?	same processor?	thread 0 wins
no	yes	99.89%
no	no	83.18%
yes	yes	64.05%
yes	no	33.92%

Table 1. Data race outcomes under various code/scheduling configurations

In this section, we quantify the amount of non-determinism that exists in ordinary multiprocessor systems. We begin with a simple experiment that illustrates how small changes to the initial state of the system can lead to entirely different program outcomes.

A simple illustration of non-deterministic behavior: Figure 2 depicts a simple program with a data-race. Table 1 illustrates various changes to the execution environment we induced and a measurement of program outcome. The point of this is that program behavior is, as we have all known, non-deterministic on a multi-core system.

Measuring non-determinism in real applications: The previous example illustrates how simple changes to the initial conditions of a multiprocessor system can lead to different program outcomes for a simple toy example. But how much non-determinism exists in real application execution?

To answer this question we first have to define a measurement technique. Returning to the definition of deterministic execution above, non-determinism is program execution occurs when a particular dynamic instance of a program *load* reads data created from a different *store* dynamic instance. We instrumented Splash2 to track communication among threads in order to quantify producer-consumer differences.

Figure 3 shows two illustrative examples of the results (*barnes*, *ocean-contig*). The x axis is time. The y axis is the percent of loads, in a 100,000 memory-instruction window, that source

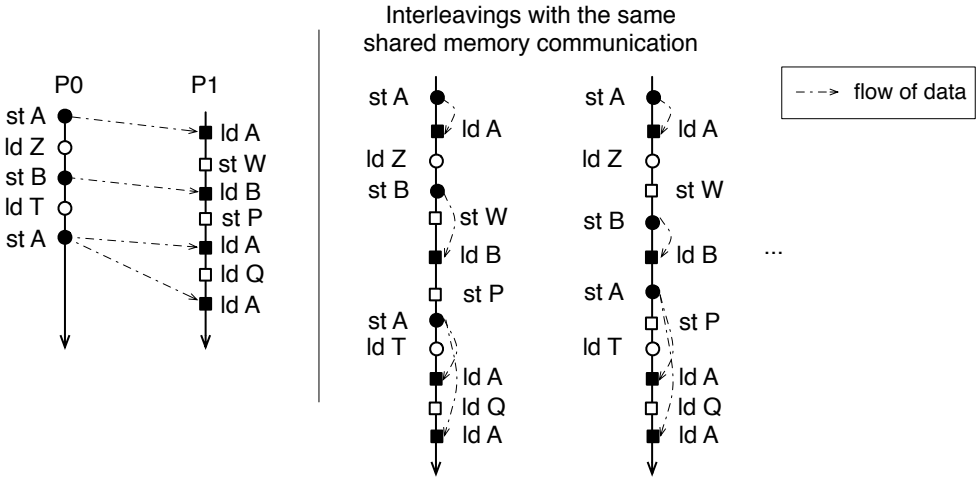

Figure 1. Making the flow of data between shared memory operations deterministic.

```
int race = -1; // global variable

void thread0() {
  if ( doClearCache )
    clearCache();
  barrier_wait();
  race = 0;
}

void thread1() {
  if ( doClearCache )
    clearCache();
  barrier_wait();
  race = 1;
}

return race;
```

Figure 2. Simple program with a data race between 2 threads.

their data from a *different* dynamic store instance. To compute the y axis we take two traces of loads and stores and between sets of producer-consumer store/loads we compute the edit-distance within each set. It is the average of this edit-difference (normalized against the total number of memory operations) that is plotted. Edit-distance is used because it correctly accounts for extra *load* instructions inserted for synchronization.

From this graphs two things are immediately apparent. First, programs have significant non-deterministic behavior. Second, the graphs have two additional properties. Both graphs depict phases of execution where non-determinism drops to nearly zero. These are created by barrier operations, which synchronize the threads and then subsequent execution is more deterministic. The second observation is that *ocean-contig* never shows 100% non-determinism, and in fact a significant fraction of *load* operations are deterministic. These *load* operations are private data. Both of these facts about program execution are significant and can be exploited in a system that actually is deterministic.

3. Enforcing Deterministic Multiprocessing

In this section we describe how to build a deterministic multiprocessor. We begin with a naive approach (which is still extremely useful for debugging), and then refine this simple technique into ever more efficient implementations.

3.1. Basic Idea

In the previous section we saw that the key to making multiprocessors deterministic was to ensure that the communication (via shared memory or otherwise) between threads was deterministic. Conceptually, the easiest way to do that is to allow only one processor to access memory at a time in a deterministic order. This process can be thought of as a memory access token being passed around between processors in a deterministic order. We call this *deterministic serialization* of a parallel execution, shown in Figure 4(b). Deterministic serialization guarantees that inter-thread communication is deterministic because it preserves all pairs of communicating memory instructions (see Section 2.1).

The simplest way of implementing such serialization is by having each processor synchronize before every memory operation to acquire the token and then, when the memory operation is completed, pass it to the next processor in the deterministic order. From now on we will call this token the *deterministic token*. A processor blocks whenever it needs access memory but does not have the deterministic token. This token can be implemented as a queue lock or flag variable.

Waiting for the token at every memory operation is certainly expensive and will cause significant performance degradation when compared to the original parallel execution (Figure 4(a)). The performance degradation stems from (i) overhead introduced by waiting and passing the deterministic token and (ii) the serialization itself, which removes the benefits of parallel execution.

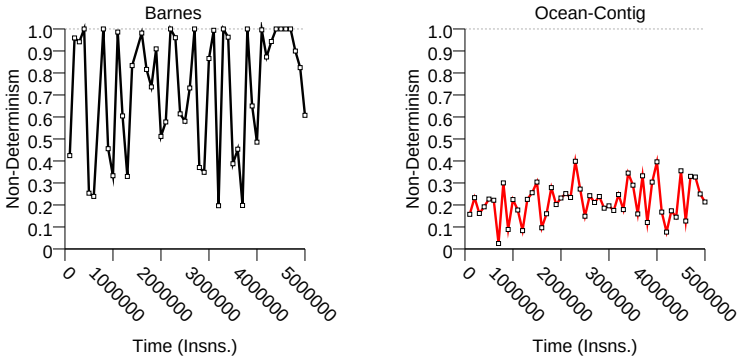

Figure 3. Profile of non-determinism over time in two applications from SPLASH2, Ocean-Contig and Barnes

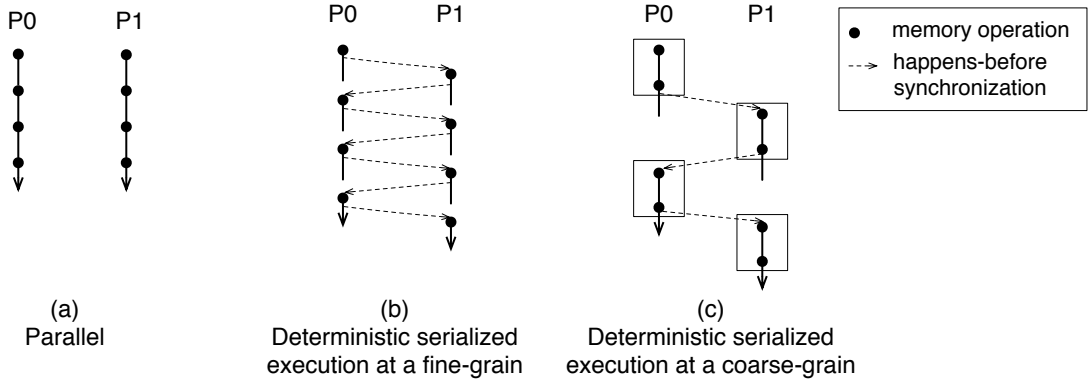

Figure 4. Deterministic serialization of memory operations.

The overhead of synchronization can be mitigated by synchronizing at a coarser granularity, allowing each processor to execute a finite, deterministic number of memory operations, henceforth called *quantum*, before passing the token to the next processor.

Reducing the impact of serialization requires enabling parallel execution while preserving the same execution behavior as deterministic serialization. We propose three main techniques to recover parallelism. The first technique exploits the fact that synchronization is only necessary before accesses to shared pieces of memory, allowing concurrent execution of private memory accesses. The second uses speculation to optimize parallel execution of quanta from different processors. The trade-off between these two techniques is one between performance, complexity and energy waste. The first technique requires fewer additions to a typical multiprocessor system and does not suffer from wasted work due to misspeculation but yields lower performance than using speculation, which, on the other hand is more complex to implement and uses more energy. Finally, the third main technique provides a more convenient break-down of program execution into quanta in order to reduce the lengthening of the critical path of execution of a parallel program. Below, we now describe each technique in detail.

3.2. Leveraging Private Data Information

The performance of deterministic parallel execution can be improved by leveraging the observation that only memory accesses to shared data need to be deterministically serialized. This implies that a thread only needs to wait for the token when it accesses shared data. If the thread is able to identify when an access is private, it can issue the access even if it does not hold the token. Figure 5 illustrates this concept. As a result, the system can execute memory operations that access private data concurrently with memory operations from other processors.

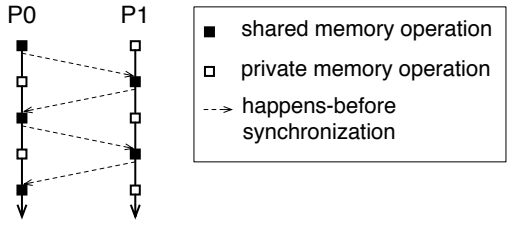

Figure 5. Recovering parallelism with deterministic serialization of shared memory operations only.

The key to making this efficient is to provide a low-overhead mechanism of determining what memory operations in a thread access private data. There are three main possibilities: (i) dynamically determining what pieces of data is shared my maintaining a sharing table (a table which tracks global cache-line states); (ii) statically determining private data access with a compiler and implementing ISA support for private loads. In this paper, we do not explore option (ii). Below we explore how to implement the first option.

A sharing table is a data structure in memory that contains sharing information for each memory position (usefully, but not necessarily, aggregated into cache line size blocks). A thread can access its own private data without holding the deterministic token. A thread can also read shared data without holding the token. However, in order to write to shared data or read data regarded as private by another thread, a thread needs to hold the token and wait until all other threads are blocked waiting for the token. This guarantees that the sharing information is kept consistent. When a thread t writes to a piece of data, the data is set to private with t as the owner. Similarly, when a thread t reads data for the first time, it is set as private owned by t. Figure 6 illustrates this process with a flowchart.

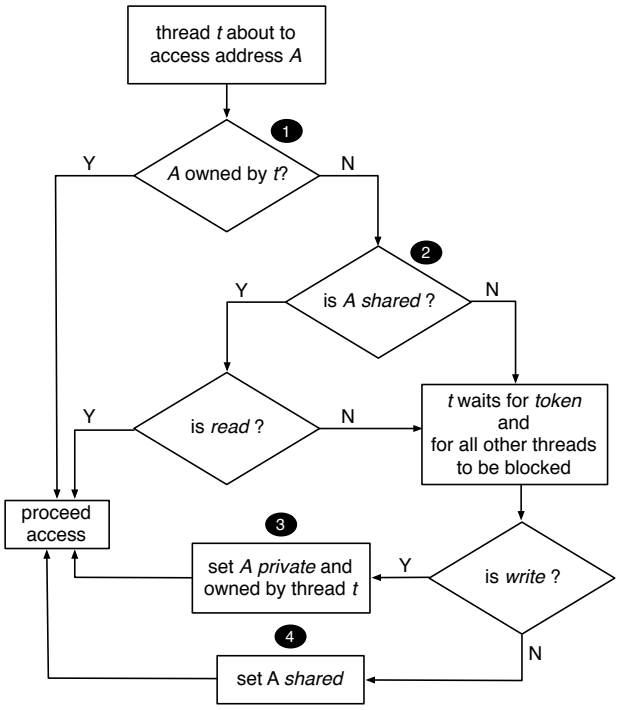

Figure 6. Flowchart for deterministic serialization of shared memory operations only.

The sharing information itself can be implemented in several ways. In some respects, the sharing table is the state of cache lines in a multiprocessor system. Determinism is achieved by carefully orchestrating in time, when a cache-line changes state from exclusive to shared or the reverse. One particular implementation detail of this table is, just like with the shared state of MESI, it is not important which processors have the cache-line, only that the cache-line is in a shared, read-only state. Similarly, a sharing table need only track whether a memory address is in an exclusive state (and which thread owns it), or that no-thread owns the address and it is shared. A sharing table has no need for the M (modified) or I (invalid) states. Figure 6 depicts a finite-state machine for processing a memory request with a sharing table.

The sharing table itself can be instrumented in either software or hardware. A software implementation simply inlines the decision tree from Figure 6 into the application. A hardware implementation should be piggy-backed onto the coherence protocol. The changes required to the base MESI protocol are straightforward: cache-lines cannot be converted from exclusive to shared or shared to exclusive, unless the processor holds the deterministic token. All other state transitions remain the same. A hybrid HW/SW approach is also possible. Instead of maintaining a separate sharing table for cache-lines, if a processor can simply trap to software if it misses in the cache, then a flexible software solution can be implemented, with the common-case cache-hit requiring no additional overhead.

3.3. Leveraging Support for Transactional Memory

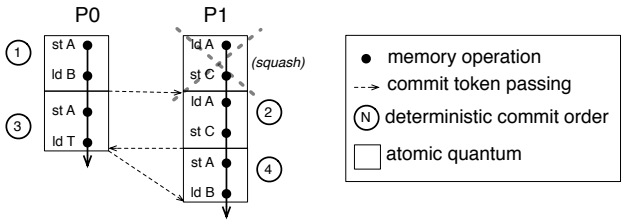

Figure 7. Recovering parallelism by executing quanta as atomic transactions.

It is easy to see that executing quanta atomically and in isolation in a deterministic total order would be equivalent to deterministic serialization of memory operations. This implies that as long as quanta *appear* to execute atomically and in isolation, the execution will be equivalent to deterministic serialization. Transactional memory systems can be leveraged for this purpose.

We can leverage support for transactional memory by treating each quantum as a transaction. In addition to support for transactional memory, we also need a mechanism to form quanta deterministically, as well as a mechanism to enforce a pre-defined commit order. As Figure 7 illustrates, a quantum runs concurrently with other quanta in the system as long as there are no overlapping memory accesses that would violate the original deterministic serialization of memory operations. In case a conflict happens, the quantum later in the total deterministic order gets squashed and re-executed. Note that the total deterministic order of quantum commits is a key component in guaranteeing deterministic serialization of memory operations.

As mentioned earlier, a quantum is a unit of work with a deterministic number of instructions. In order to achieve that, the process of breaking the execution of a thread into quanta has to be deterministic. The choice of where to break the execution can be done in software (compiler, or binary instrumentation) or by the architecture. A key fact here, however, is these transactions are not programmer-directed transactions. In our implementation, we keep a private counter for each thread. At the start of any hyperblock of execution, this counter is incremented, and when it reaches a predefined value, the transaction is committed and a new one started. Several alternatives are possible (tail end of loops, etc).

A hardware implementation would clearly be more efficient for forming transactions and can be implemented by a simple instruction completion count counter. Note, that for deterministic execution *bounded* TM systems can be used. So long as the bounds of the TM are reached *deterministically* for each trans-

action, then it is appropriate to simply end the transaction at the hardware bound, and begin a new one. This means even simple schemes that hold onto cache-lines are able to implement deterministic execution.

Having a total predefined commit order allows uncommitted (or speculative) data to flow between quanta. This can potentially save a large number of squashes in application that have more intensive inter-thread communication. The idea is allow a quantum to fetch speculative data from an uncommitted quantum that happened earlier in the deterministic total order. This is illustrated in Figure 8, where quantum (2) fetched an uncommitted version of A from (1). Note that without support for forwarding, (2) would have been squashed. In order to guarantee correctness, though, if a quantum that provided data to other quanta is later squashed, all later quanta also need to be squashed, since they might have consumed misspeculated data.

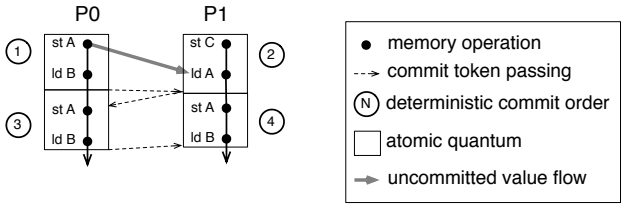

Figure 8. Avoiding unnecessary squashes with un-committed data forwarding.

3.4. Exploiting the Critical Path

The critical path of a parallel application is composed of multiple sections of different threads. It can be thought of as the path of a "criticality" token that gets passed between threads as the application execution progresses. Intuitively, the criticality token is passed around between threads as they synchronize and communicate with each other.

We can exploit knowledge of how programs typically are written to adapt the size of quanta to make more efficient progress on the critical path of execution. We devised two heuristics to create quanta that better match the critical path of a program. The first heuristic, called *sync-follow* simply ends a quantum when an unlock operation is performed. The rationale is that when a thread releases a lock, other threads might be spinning waiting for that lock, so the deterministic token should be sent forward as early as possible to allow waiting thread to make progress. Figure 9 illustrates such scenario.

The second heuristic relies on information about data sharing in order to identify when a thread has potentially completed work on shared data, and consequently ends a quantum at that time. It does so by determining when a thread hasn't issued memory operations to shared locations in some time; e.g. in the last 30 memory operations. The rationale is that when a thread is working on shared data, it is expected that others thread will access that data soon. By ending a quantum early and passing the deterministic token, the consumer thread potentially consumes the data earlier than if the quantum in the producer thread ran longer. This not only has an effect on performance, but also is likely to reduce the amount of work wasted by squashed in the transactional memory-based implementation discussed earlier in this section.

Other quanta breaking techniques are possible (system call boundaries, exposing the breaking to the programmer, etc). The key to any of them, however, is they must break in deterministic ways. The two described above do so, and any new ones explored must as well.

4. Experimental Setup

We evaluate the performance impact of deterministic execution with a simulator written as a tool for the PIN [3] binary instrumentation infrastructure from Intel. Our simulator monitors the execution of an application, builds quanta based on the instructions that are executed and then builds a schedule of their execution. The model includes the effects of serialization, memory conflicts and limited buffering in the transactional memory support.

We evaluate all strategies described in Section 3: *Lock* refers to the basic approach of complete serialization (Section 3.1); *SharingTable* refers to the approach that recovers parallelism by overlapping execution of private memory accesses (Section 3.2); *TM-Bounded* refers to the transactional-memory based approach with support for a single outstanding transaction; and *TM-Forward* refers to the transactional-memory based approach with unlimited buffering and support for speculative value forwarding. We also evaluate the effect of conflict detection granularity by supporting conflict detection at the granularity of word and 32-byte cache-line sizes.

The simulator also models different quantum builder strategies. The *Base* quantum builder just builds quanta based on instruction count (every 1,000 or 10,000 instructions). The *SharingMonitor* builds quanta based on the sharing monitoring heuristic described in Section 3.4. For the preliminary results included in this submission, the data for the *SyncFollow* quantum building was not ready.

For this preliminary study we chose applications from the SPLASH2 [8] benchmarks suite, although this tool supports any application running on a Linux-x86 machine. Table 2 describes the benchmarks in more detail.

Application	Description
fft	Fast Fourier transformation
lu	LU matrix factorization
ocean	Ocean movements simulation
radix	Sorting algorithm
volrend	Volume rendering
water-ns	Water molecule system simulation
water-sp	Water molecule system simulation

Table 2. Benchmarks used in our evaluation

5. Evaluation

Figure 10 shows the scalability of our techniques compared to the original parallel baseline. We ran the SPLASH-2 [8] parallel benchmark suite with 2, 4 and 8 threads, using word-granularity conflict detection and 10,000-instruction quanta. The simple lock-based deterministic scheme shows the poorest scalability, degrading nearly linearly with the number of threads for most benchmarks (as one would expect). The performance degradation is

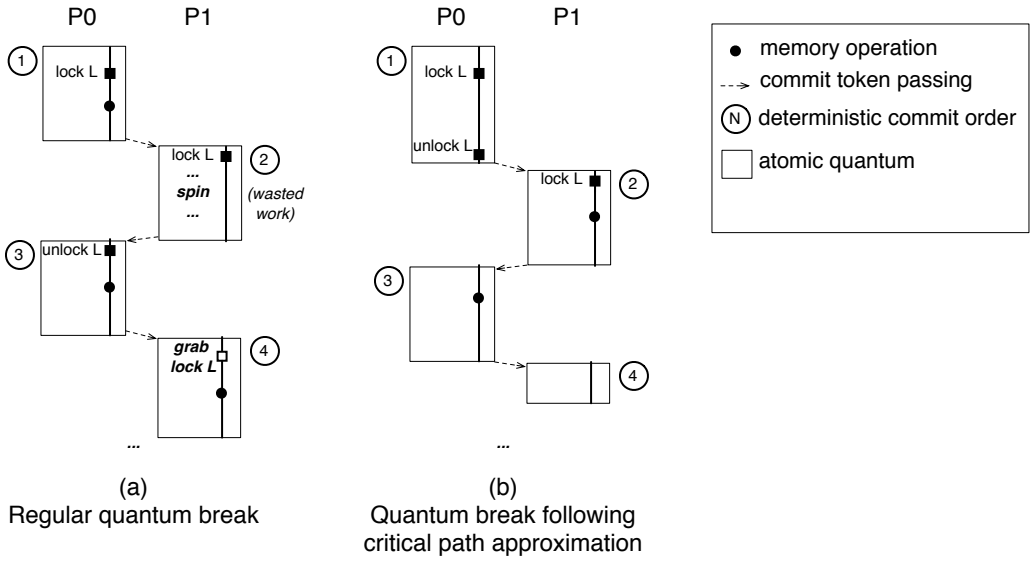

Figure 9. Example of a situation when better quantum breaking policies leads to better performance.

sublinear because our deterministic scheme affects only the multithreaded portion of an application's execution. Some of the benchmarks with a substantial amount of single-threaded work (e.g. `lu` and `fft`) exhibit this behavior. The scalability of the sharing table scheme depends on the amount of data sharing in the program, which is generally low for well-engineered applications like SPLASH2, but is quite high in the case of `radix` and `ocean`.

Using transactional memory helps reduce overheads substantially, by allowing "memory renaming" which avoids unnecessary conflicts on WAR and WAW dependences. The TM-Bounded scheme allows one transaction commit to be buffered, showing that high performance can be acheived with modest buffering requirements. The TM-Forward scheme represents a very aggressive design that speculatively forwards writes to reduce the latency of RAW conflicts. The overhead of the TM-based schemes is low ($< 50\%$ even with 8 threads), showing that for well-designed parallel programs, it is possible to obtain determinism and performance.

Decreasing the size of quanta from 10,000 to 1,000 instructions reduces overheads afor all deterministic schemes (Figure 11). We use 8 threads and word-granularity conflict detection. Smaller quanta would in practice increase the amortized overheads of starting and committing a quanta, but also lead to lower abort costs due to the decreased amount of lost work. Our model ignores these costs, but does take into account the reduced probability of conflicts due to fewer loads and stores being inside each quantum. The sharing table scheme, with its inability to speculatively proceed past potential conflicts, gains the most from smaller quanta. The performance of TM-Bounded converges with that of TM-Forward as the probability of having a conflicts drops; the latter scheme's ability to avoid conflicts is significant only with larger quanta.

Coarsening the granularity of conflict detection allows for a simpler hardware implementation, at the expense of a greater number of false conflicts. False conflicts in any scheme occur for two reasons. The first source of false conflicts is cache line aliasing of memory addresses. The second source is that in the non-TM based sharing monitor scheduling technique, WAR and WAW hazards are viewed as conflicts, when in fact they could safely be ignored with TM's "memory renaming". Figure 12 compares conflict detection at word (4-byte) and cache line (32-byte) granularity, with 8 threads and 10,000-instruction quanta. This affects the overhead of the TM-based schemes, but not to a large extent: none of the TM schemes have an overhead of more than 2x, and with speculative forwarding the overhead is less than 1.4x.

Using different quantum builders (Figure 13) affects overhead in a variable manner. We use 8 threads and 10,000-instruction quanta. Monitoring the flow of shared data and using heuristics to anticipate inter-thread communication helps some applications (e.g. `volrend`, `ocean`) while hurting others (`fmm`). As these results are preliminary, we do not evaluate the effect of producing quanta according to observed synchronization events.

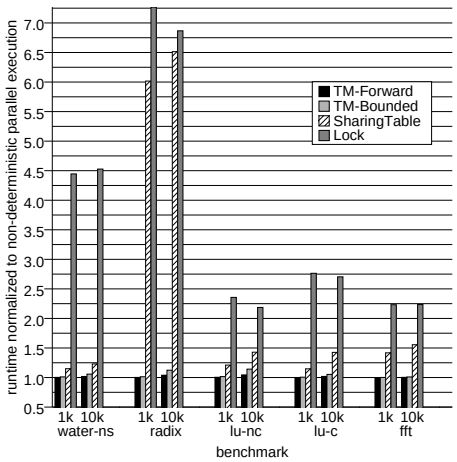

Figure 11. 1,000- versus 10,000-instruction quanta.

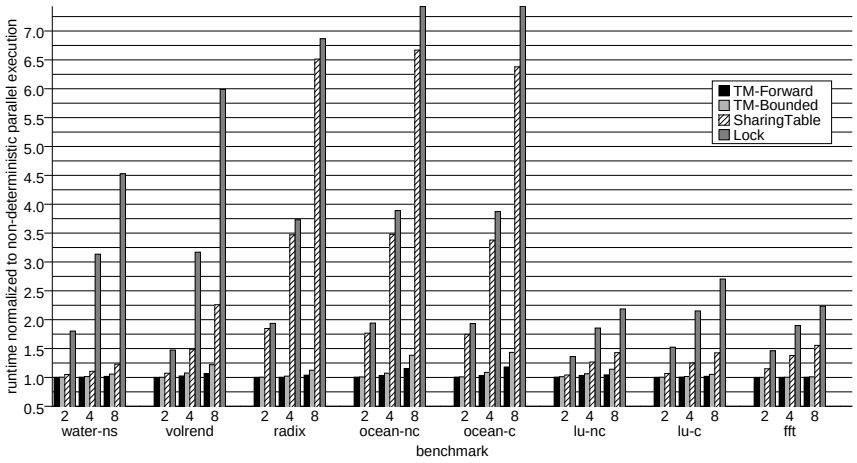

Figure 10. Runtime overheads with 2, 4 and 8 threads.

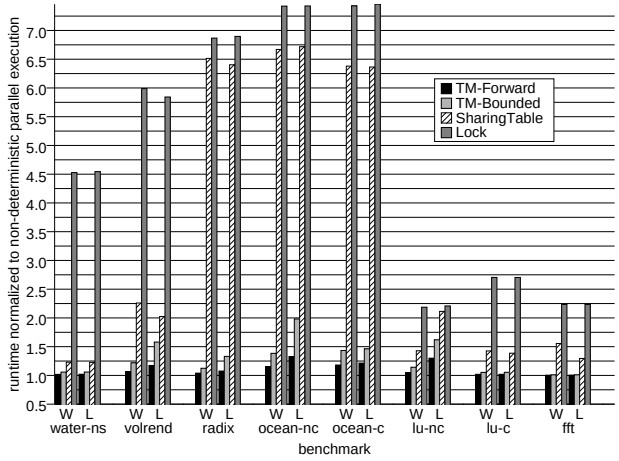

Figure 12. Word- versus line-granularity conflict detection.

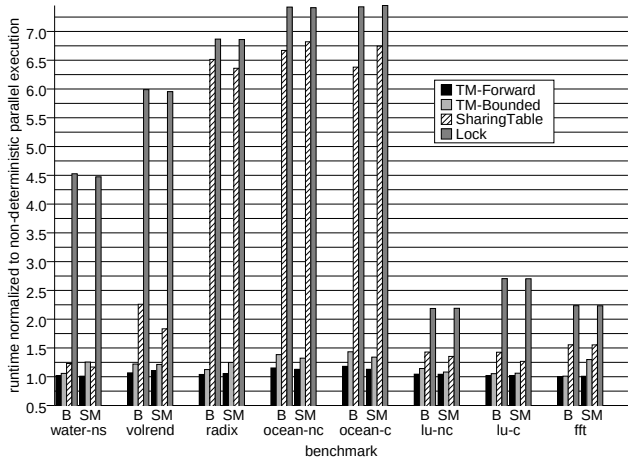

Figure 13. Base (B) versus SharingMonitor (SM) quantum builders.

6. Related Work

This work addresses the problem of eliminating non-determinism from parallel executions. In some sense an attempt at this goal was the advent of synchronization primitives — programmers eliminate troublesome non-determinism by preventing problematic interleavings with synchronization. Very little continued work has been done to truly eliminate non-determinism from the execution of parallel programs.

Architects and systems researchers have recognized the difficulty non-determinism presents to software developers. For this reason, a variety of replay techniques [9, 6, 4, 1, 2, 5] have been proposed. Replay allows programmers to execute applications in a special "debug mode", that enables reliable replaying of a particular sequence of execution. Among recent advances in hardware support for deterministic replay are ReRun [1] and DeLorean [4], which aim at reducing log size and hardware complexity. ReRun is a hardware memory race recording mechanism, which uses *episodes* to enable replay an execution. ReRun constructs episodes by recording portions of an execution during which no conflicts are detected. This novel strategy consumes little hardware state, and produces a small race log from which executions equivalent to the original can be replayed. DeLorean is another hardware approach to supporting deterministic replay, in which instructions are executed as blocks (or chunks), and the commit order of blocks of instructions are recorded, as opposed to each instruction. DeLorean is able to record this data efficiently on-the-fly and produces very small logs. DeLorean also uses pre-defined ordering to some extent to further reduce the memory ordering log.

Software approaches to deterministic replay have also shown a great degree of success. DejaVu [2] provides deterministic replay for applications running in a Java Virtual Machine, efficiently and independently of the thread scheduler being used in the underlying VM. RecPlay [7] combines memory access recording and replaying with online data-race detection. The combination of these techniques makes the record phase of this process very efficient, and permits data race detection to be postponed to the replay phase, also contributing to efficiency.

Overall, replaying has its place, but it also has its limitations. First among them, replay does not make multithreaded execution deterministic, it merely enables the deterministic replay of a particular non-deterministic execution. While far more useful than nothing at all, replay does not help give precise meaning to what a regression test means. Nor do they provide the ability to have con-

fidence that the interleavings that occur during deployment can be tested during development.

7. Conclusion

This paper makes a bold claim: the explicitly parallel shared memory programming model is fundamentally broken due to non-determinism. Program output should *not* vary each time it is executed, essentially at random. The reason it does is – up until now – because it was viewed as the only way to make efficient parallel processors. Making the execution model deterministic was thought to cost too much in terms of performance.

In this paper we have shown that conventional wisdom to be false: a deterministic multiprocessor can be built to be fast. The key is to understand what it really takes to be deterministic, which is deterministic communication between threads. Building on recent research into shared memory cache coherence systems and transactional memory support, this paper has outlined a variety of implementation strategies to build a fully deterministic multiprocessor.

Our results have shown a variety of viable techniques to achieve determinism. These techniques have various micro-architectural trade-offs, such as quanta, speculation versus locking, etc. These trade-offs will need to be more thoroughly explored and matched to whatever specific micro-architectural solution is implemented. But, the high-level result of this paper is that multiprocessor systems can be deterministic with no performance impact due intrinsically to deterministic behavior. Any performance impact will be because of micro-architectural implementation details. It is the subject of our future work to make these as minimal as possible. Given that they are largely based around transactional memory support and cache coherence system schemes, and these schemes have minimal impact, we expect this will be the case with determinism.

In the future, we expect all shared memory systems will be deterministic. This paper points the way. With little performance impact, and obvious benefits to software development and programmer sanity, there is no reason not to.

References

[1] D. Hower and M. Hill. Rerun: Exploiting Episodes for Lightweight Memory Race Recording. In *ISCA*, 2008.

[2] J. Choi and H. Srinivasan. Deterministic Replay of Java Multithreaded Applications. In *SIGMETRICS SPDT*, 1998.

[3] C. K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Janapa Reddi, and K. Hazelwood. PIN: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *PLDI*, 2005.

[4] P. Montesinos, L. Ceze, and J. Torrellas. DeLorean: Recording and Deterministically Replaying Shared-Memory Multiprocessor Execution Efficiently. In *ISCA*, 2008.

[5] S. Narayanasamy, C. Pereira, and B. Calder. Recording Shared Memory Dependencies Using Strata. In *ASPLOS*, 2006.

[6] S. Narayanasamy, G. Pokam, and B. Calder. BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging. In *ISCA*, Los Alamitos, CA, USA, 2005.

[7] M. Ronsee and K. De Bosschere. RecPlay: A Fully Integrated Practical Record/Replay System. *ACM TOCS*, 1999.

[8] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *ISCA*, 1995.

[9] M. Xu, R. Bodik, and M. Hill. A "Flight Data Recorder" for Enabling Full-System Multiprocessor Deterministic Replay. In *ISCA*, 2003.