DMP: Deterministic Shared Memory Multiprocessing

Joseph Devietti Brandon Lucia Luis Ceze Mark Oskin

Computer Science & Engineering, University of Washington

{devietti,blucia0a,luisceze,oskin}@cs.washington.edu

Abstract

Current shared memory multicore and multiprocessor systems are nondeterministic. Each time these systems execute a multithreaded application, even if supplied with the same input, they can produce a different output. This frustrates debugging and limits the ability to properly test multithreaded code, becoming a major stumbling block to the much-needed widespread adoption of parallel programming.

In this paper we make the case for fully deterministic shared memory multiprocessing (DMP). The behavior of an arbitrary multithreaded program on a DMP system is only a function of its inputs. The core idea is to make inter-thread communication fully deterministic. Previous approaches to coping with nondeterminism in multithreaded programs have focused on replay, a technique useful only for debugging. In contrast, while DMP systems are directly useful for debugging by offering repeatability by default, we argue that parallel programs should execute deterministically in the field as well. This has the potential to make testing more assuring and increase the reliability of deployed multithreaded software. We propose a range of approaches to enforcing determinism and discuss their implementation trade-offs. We show that determinism can be provided with little performance cost using our architecture proposals on future hardware, and that software-only approaches can be utilized on existing systems.

Categories and Subject Descriptors C [*0*]: GENERAL — Hardware/software interfaces; C [*1*]: 2 Multiple Data Stream Architectures (Multiprocessors) — Multiple-instruction-stream, multiple-data-stream processors

General Terms performance, design, reliability

Keywords multicores, determinism, parallel programming, debugging

1. Introduction

Developing multithreaded software has proven to be much more difficult than writing singlethreaded code. One of the major challenges is that current multicores execute multithreaded code nondeterministically [13]: given the same input, threads can interleave their memory and I/O operations differently in each execution.

Nondeterminism in multithreaded execution arises from small perturbations in the execution environment, e.g., other processes executing simultaneously, differences in the operating system resource allocation, state of caches, TLBs, buses and other microarchitectural structures. The ultimate consequence is a change in program behavior in each execution. Nondeterminism complicates the software development process significantly. Defective software might execute correctly hundreds of times before a subtle synchronization bug appears, and when it does, developers typically cannot reproduce it during debugging. In addition, nondeterminism also makes it difficult to test multithreaded programs, as good coverage requires both a wide range of program inputs and wide range of possible interleavings. Moreover, if a program might behave differently each time it is run with the same input, it becomes hard to assess test coverage. For that reason, a significant fraction of the research on testing parallel programs is devoted to dealing with nondeterminism.

While there were significant efforts in addressing the problem of nondeterminism, most of it has been on deterministically replaying multithreaded execution based on a previously generated log [9, 15, 16, 19, 23]. In contrast, this paper makes the case for *fully deterministic shared memory multiprocessing* (DMP). We show that, with hardware support, arbitrary shared memory parallel programs can be executed deterministically with very little performance penalty. In addition to the obvious benefits deterministic multiprocessing has for debugging, i.e., repeatability, we argue that parallel programs should always execute deterministically in the field. This has the potential to make testing more assuring, to allow a more meaningful collection of crash information, and to increase the reliability of multithreaded software deployed in the field, since execution in the field will better resemble in-house testing. Moreover, the same techniques we propose to enable efficient deterministic execution can also be used to directly manipulate thread interleavings and allow better testing of multithreaded programs. Overall, deterministic multiprocessing positively impacts both development and deployment of parallel programs.

This paper is organized as follows. We begin by providing a precise definition of what it means to execute deterministically (Section 2). A key insight of this definition is that multithreaded execution is deterministic if the communication between threads is deterministic. We then discuss what gives rise to nondeterminism and provide experimental data that shows, as one would expect, that current shared memory multiprocessors are indeed highly nondeterministic. After that we describe how to build an efficient deterministic multiprocessor. We propose three basic mechanisms (Section 3) for doing so: (1) using serialization to transform multithreaded execution into single-threaded execution; (2) restricting accesses to shared data to guarantee deterministic communication; and (3) using transactions to speculatively communicate between threads, rolling back if determinism has been violated. Moreover, we propose a series of optimizations to further reduce the performance overheads to a negligible amount. We discuss the trade-offs of hardware implementations and contrast them to software implementations (Section 4). We then move on to describe our experimental infrastructure (Section 5), composed of a simulator for the hardware implementation and a fully functional software-based implementation using the LLVM [11] compiler infrastructure. We present performance and characterization results (Section 6) for both the hardware and compiler-based implementations. We then discuss trade-offs and system issues such as interaction with the

ASPLOS'09, March 7–11, 2009, Washington, DC, USA.

operating system and debugger (Section 7). Finally, we discuss related work (Section 8) and then conclude (Section 9).

2. Background

2.1 A Definition of Deterministic Parallel Execution

We define a deterministic shared memory multiprocessor system as a computer system that: (1) executes multiple threads that communicate via shared memory, and (2) will produce the same program output if given the same program input. This definition implies that a parallel program running on a DMP system is as deterministic as a single-threaded program.

The most direct way to guarantee deterministic behavior is to preserve the same global interleaving of instructions in every execution of a parallel program. However, several aspects of this interleaving are irrelevant for ensuring deterministic behavior. It is not important which global interleaving is chosen, as long as it is always the same. Also, if two instructions do not communicate, their order can be swapped with no observable effect on program behavior. What turns out to be the key to deterministic execution is that all communication between threads must be precisely the same for every execution. This guarantees that the program always behaves the same way if given the same input.

Guaranteeing deterministic communication between threads requires that each dynamic instance of an instruction (consumer) read data produced from the same dynamic instance of another instruction (producer). Producer and consumer need not be in the same thread, so this communication happens via shared memory. Interestingly, there are multiple global interleavings that lead to the same communication between instructions, they are called *communication-equivalent interleavings*, illustrated in Figure 1. In summary, any communication-equivalent interleaving will yield the same program behavior. To guarantee deterministic behavior, then, we need to carefully control only the behavior of load and store operations that cause communication between threads. This insight is key to efficient deterministic execution.

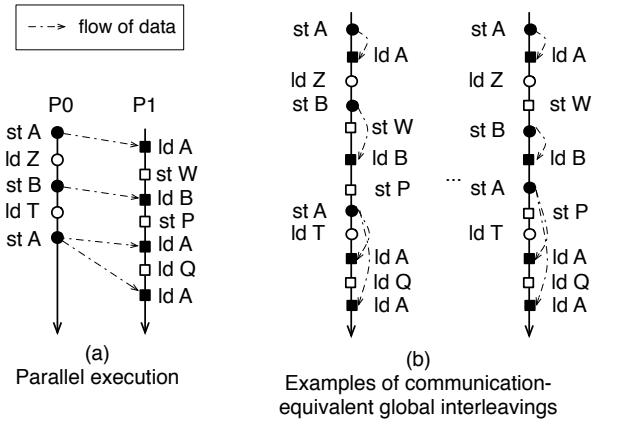

Figure 1. An illustration of a parallel execution (a) and two of its multiple communication-equivalent interleavings (b). Solid markers represent communicating instructions, while hollow markers represent instructions that do not communicate between threads.

2.2 Nondeterminism in existing systems

Current generation software and hardware systems are not built to behave deterministically. Multiprocessor systems execute programs nondeterministically because the software environment and the non-ISA microarchitectural state change from execution to execution. These effects manifest themselves as perturbations in the timing between events in different threads of execution, causing the final global interleaving of memory operations in the system to be different. Ultimately, the effect on the application is a change in which dynamic instance of a store instruction produces data for a dynamic instance of a load instruction. Once this occurs, program execution behavior may diverge from previous executions at the ISA level, and, consequently, program output may vary.

2.2.1 Sources of nondeterminism

The following are some of the software and hardware sources of nondeterminism.

Software Sources. Several aspects of the software environment create nondeterminism: other processes executing concurrently and competing for resources; the state of memory pages, power savings mode, disk and I/O buffers; and the state of global data structures in the OS. In addition, several operating system calls have interfaces with legitimate nondeterministic behavior. For example, the *read* system call can legitimately take a variable amount of time to complete and return a variable amount of data.

Hardware Sources. A number of non-ISA-visible components vary from program run to program run. Among them are the state of any caches, predictor tables and bus priority controllers, in short, any microarchitectural structure. In fact, certain hardware components, such as bus arbiters, can change their outcome with each execution purely due to environmental factors; e.g., if the choice of priority is based on which signal is detected first, the outcome could vary with differing temperature and load characteristics.

Collectively, current generation software and hardware systems are not built to behave deterministically. In the next section, we measure how nondeterministically they behave.

2.2.2 Quantifying nondeterminism

We first show a simple experiment that illustrates how simple changes to the initial conditions of a multiprocessor system can lead to different program outcomes for a simple toy example. Then we measure how much nondeterminism exists in the execution of real applications.

A simple illustration of nondeterministic behavior. Figure 2 depicts a simple program with a data-race where two threads synchronize on a barrier and then write different values to the same shared variable. The table inside the figure also shows the frequency of the outcome of 1000 runs on a Intel Core 2 Duo machine. For the "clear cache" runs we had each thread set a 2MB buffer to zero before entering the barrier. As expected, the behavior is nondeterministic and varies significantly depending on the configuration being run.

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

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

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

clear cache?	thread 0 wins
no	83.2%
yes	33.9%

Figure 2. Simple program with a data race between two threads and the outcome of the race with 1000 runs.

Measuring nondeterminism in real applications. As mentioned previously, nondeterminism in program execution occurs when a

particular dynamic instance of a load reads data created from a different dynamic instance of a store. To measure this degree of nondeterminism, we start by building a set containing all pairs of communicating dynamic instructions (dynamic store $\rightarrow$ dynamic load) that occurred in an execution. We call this the *communication set*, or C. A dynamic memory operation is uniquely identified by its instruction address, its instance number[1], and the id of the thread that executed it. Assume there are two executions, $ex1$ and $ex2$, whose communicating sets are C_{ex1} and C_{ex2}, respectively. The symmetric difference between the communication sets, $C_{ex1} \triangle C_{ex2} = (C_{ex1} \cup C_{ex2}) - (C_{ex1} \cap C_{ex2})$, yields the communicating pairs that were *not* present in both executions, which quantifies the execution difference between $ex1$ and $ex2$. Finally, we define the amount of nondeterminism between $ex1$ and $ex2$ as: $ND_{ex1,ex2} = \frac{|C_{ex1} \triangle C_{ex2}|}{|C_{ex1}| + |C_{ex2}|}$ (Eq. 1), which represents the proportion of all communicating pairs that were *not* in common between the two executions. Note that this number correctly accounts for extra instructions spent in synchronization, as they will form some communicating pair.

We developed a tool using PIN [14] that collects the communication sets for multithreaded program executions. The tool takes a single sample at a specified point in the execution. We cover the whole application by running the tool twice at a number of different points in each execution, and then computing the nondeterminism between corresponding points in each run. Figure 3 shows the results for *barnes* and *ocean-contig* from the SPLASH-2 suite [22] running on an Intel Core 2 Duo machine. The plots show that programs have significant nondeterministic behavior. Moreover, they reveal two interesting properties. First, both plots depict phases of execution where nondeterminism drops to nearly zero. These are created by barrier operations that synchronize the threads and then make subsequent execution more deterministic. Second, *ocean-contig* never shows 100% nondeterminism, and, in fact, a significant fraction of pairs are the same, i.e., typically those from private data accesses. These aspects of program execution can be exploited in designing a system that is actually deterministic.

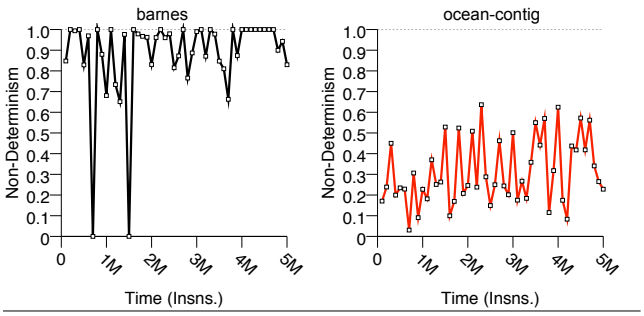

Figure 3. Amount of nondeterminism over the execution of *barnes* and *ocean-contig*. The x axis is the position in the execution where each sample of 100,000 instructions was taken. The y axis is ND (Eq. 1) computed for each sample in the execution.

3. Enforcing Deterministic Shared Memory Multiprocessing

In this section we describe how to build a deterministic multiprocessor system. We focus on the key mechanisms without discussing specific implementations, which we address in Section 4. We begin with a basic naive approach, and then refine this simple technique into progressively more efficient organizations.

3.1 Basic Idea — *DMP-Serial*

As seen earlier, making multiprocessors deterministic depends upon ensuring that the communication between threads is deterministic. The easiest way to accomplish this is to allow only one processor at a time to access memory in a deterministic order. This process can be conceptualized as a memory access token being deterministically passed among processors. We call this *deterministic serialization* of a parallel execution, shown in Figure 4(b). Deterministic serialization guarantees that inter-thread communication is deterministic by preserving all pairs of communicating memory instructions.

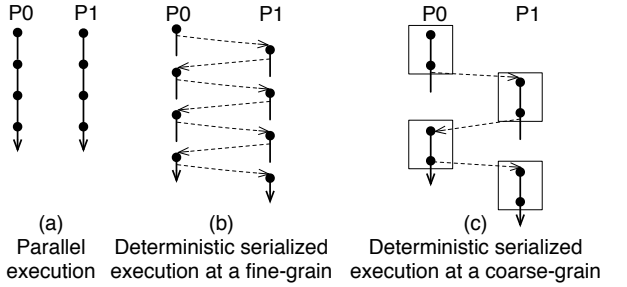

Figure 4. Deterministic serialization of memory operations. Dots represent memory operations and dashed arrows represent happens-before synchronization.

The simplest way to implement such serialization is to have each processor obtain the memory access token (henceforth called *deterministic token*) and, when the memory operation is completed, pass it to the next processor in the deterministic order. A processor blocks whenever it needs to access memory but does not have the deterministic token.

Waiting for the token at every memory operation is likely to be expensive and will cause significant performance degradation compared to the original parallel execution (Figure 4(a)). Performance degradation stems from overhead introduced by waiting and passing the deterministic token and from the serialization itself, which removes the benefits of parallel execution. Synchronization overhead can be mitigated by synchronizing at a coarser granularity (Figure 4(c)), allowing each processor to execute a finite, deterministic number of instructions, or a *quantum*, before passing the token to the next processor. A system with serialization at the granularity of quanta is called *DMP-Serial*. The process of dividing the execution into quanta is called *quantum building*: the simplest way to build a quantum is to break execution up into fixed instruction counts (e.g. 10,000). We call this simple quantum building policy *QB-Count*.

Note that DMP does not interfere (e.g., introduce deadlocks) with application-level synchronization. DMP offers the same memory access semantics as traditional nondeterministic systems. Hence, the extra synchronization imposed by a DMP system resides below application-level synchronization and the two do not interact.

3.2 Recovering Parallelism

Reducing the impact of serialization requires enabling parallel execution while preserving the same execution behavior as deterministic serialization. We propose two techniques to recover parallelism.

[1] The instance number is obtained by keeping a per-thread counter of the number of dynamic memory operations executed by each instruction pointer.

The first technique exploits the fact that threads do not communicate all the time, allowing concurrent execution of communication-free periods. The second technique uses speculation to allow parallel execution of quanta from different processors, re-executing quanta when determinism might have been violated.

3.2.1 Leveraging Communication-Free Execution — *DMP-ShTab*

The performance of deterministic parallel execution can be improved by leveraging the observation that threads do not communicate all the time. Periods of the execution that do not communicate can execute in parallel with other threads. Thread communication, however, must happen deterministically. With *DMP-ShTab*, we achieve this by falling back to deterministic serialization only while threads communicate. Each quantum is broken into two parts: a communication-free prefix that executes in parallel with other quanta, and a suffix, from the first point of communication onwards, that executes serially. The execution of the serial suffix is deterministic because each thread runs serially in an order determined by the deterministic token, just as in DMP-Serial. The transition from parallel execution to serial execution is deterministic because it occurs only when all threads are blocked – each thread will block either at its first point of inter-thread communication or, if it does not communicate with other threads, at the end of its current quantum. Thus, each thread blocks during each of its quanta (though possibly not until the end), and each thread blocks at a deterministic point within each quantum because communication is detected deterministically (described later).

Inter-thread communication occurs when a thread writes to shared (or non-private) pieces of data. In this case, the system must guarantee that all threads observe such writes at a deterministic point in their execution. Figure 5 illustrates how this is enforced in DMP-ShTab. There are two important cases: (1) reading data held private by a remote processor, and (2) writing to shared data (privatizing it). Case (1) is shown in Figure 5(a): when quantum 2 attempts to read data that is held private by a remote processor P0, it must first wait for the deterministic token and for all other threads to be blocked waiting for the deterministic token. In this example, the read cannot execute until quantum 1 finishes executing. This is necessary to guarantee that quantum 2 always gets the same data, since quantum 1 might still write to A before it completes executing. Case (2) is shown in Figure 5(b): when quantum 1, which already holds the deterministic token, attempts to write to a piece of shared data, it must also wait for all other threads to be blocked waiting for the deterministic token. In this example, the store cannot execute until quantum 2 finishes executing. This is necessary to guarantee that all processors observe the change of the state of A (from shared to privately held by a remote processor) at a deterministic point in their execution. Note that each thread waits to receive the token when it reaches the end of a quantum before starting its next quantum. This periodically (and deterministically) allows a thread waiting for all other threads to be blocked to make progress.

To detect writes that cause communication, DMP-ShTab needs a global data-structure to keep track of the sharing state of memory positions. A *sharing table* is a conceptual data structure that contains sharing information for each memory position; it can be kept at different granularities, e.g., line or page. Figure 6 shows a flowchart of how the sharing table is used. Some accesses can freely proceed in parallel: a thread can access its own private data without holding the deterministic token (1) and it can also read shared data without holding the token (2). However, in order to write to shared data or read data regarded as private by another thread, a thread needs to wait for its turn in the deterministic total order, when it holds the token and all other threads are blocked also waiting for the token (3). This guarantees that the sharing information is kept consistent and its state transitions are deterministic. When a thread writes to a piece of data, it becomes the owner of the data (4). Similarly, when a thread reads data not yet read by any thread, it becomes the owner of the data. Finally, when a thread reads data owned by another thread, the data becomes shared (5).

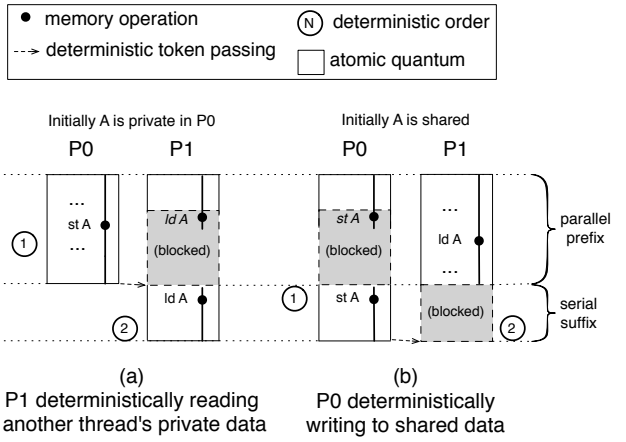

Figure 5. Recovering parallelism by overlapping communication-free execution.

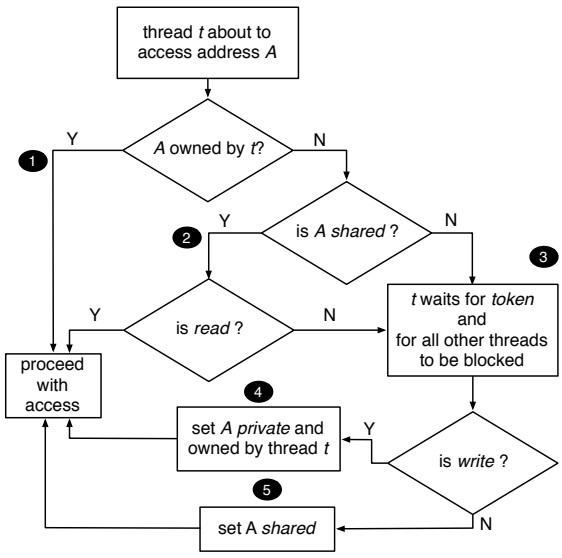

Figure 6. Deterministic serialization of shared memory communication only.

In summary, DMP-ShTab lets threads run concurrently as long as they are not communicating. As soon as they attempt to communicate, the sharing table deterministically serializes communication.

3.2.2 Leveraging Support for Transactional Memory — *DMP-TM* and *DMP-TMFwd*

Executing quanta atomically and in isolation in a deterministic total order is equivalent to deterministic serialization of memory operations. To see why, consider a quantum executed atomically and in isolation as a single instruction in the deterministic total order,

which is the same as DMP-Serial. Transactional Memory [5, 6] can be leveraged to make quanta *appear* to execute atomically and in isolation. This, coupled with a deterministic commit order, makes execution equivalent to deterministic serialization while recovering parallelism.

We use TM support by encapsulating each quantum inside a transaction, making it appear to execute atomically and in isolation. In addition, we need a mechanism to form quanta deterministically and another to enforce a deterministic commit order. As Figure 7(a) illustrates, speculation allows a quantum to run 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 of conflict, the quantum later in the deterministic total order gets squashed and re-executed (2). Note that the deterministic total order of quantum commits is a key component in guaranteeing deterministic serialization of memory operations. We call this system *DMP-TM*.

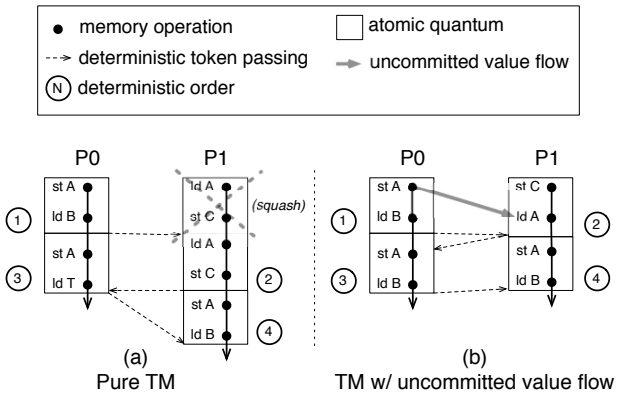

Figure 7. Recovering parallelism by executing quanta as memory transactions (a). Avoiding unnecessary squashes with uncommitted data forwarding (b).

Having a deterministic commit order also allows isolation to be selectively relaxed, further improving performance by allowing uncommitted (or speculative) data to be forwarded between quanta. This can potentially save a large number of squashes in applications that have more inter-thread communication. To do so, we allow a quantum to fetch speculative data from another uncommitted quantum earlier in the deterministic order. This is illustrated in Figure 7(b), where quantum 2 fetches an uncommitted version of A from quantum 1. Note that without support for forwarding, quantum 2 would have been squashed. To guarantee correctness, if a quantum that provided data to other quanta is squashed, all subsequent quanta must also be squashed since they might have consumed incorrect data. We call a DMP system that leverages support for TM with forwarding *DMP-TMFwd*.

Another interesting effect of pre-defined commit ordering is that memory renaming can be employed to avoid squashes on write-after-write and write-after-read conflicts. For example, in Figure 7(a), if quanta 3 and 4 execute concurrently, the store to A in (3) need not squash quantum 4 despite their write-after-write conflict.

3.3 Exploiting the Critical Path — *QB-SyncFollow*, *QB-Sharing* and *QB-SyncSharing*

The most basic quantum building policy, QB-Count, produces quanta based on counting instructions and breaking a quantum when a deterministic, target number of instructions is reached. However, instruction-count based quantum building does not capture the fact that threads do not have the same rate of progress. It also does not capture the fact that multithreaded programs have a critical path. Intuitively, the critical thread changes as threads communicate with each other via synchronization operations and data sharing.

We now describe how to exploit typical program behavior to adapt the size of quanta and lead to more efficient progress on the critical path of execution. We devised three heuristics to do so. The first heuristic, called *QB-SyncFollow* simply ends a quantum when an unlock operation is performed. As Figure 8 shows, the rationale is that when a thread releases a lock (P0), other threads might be spinning waiting for that lock (P1), so the deterministic token should be sent forward as early as possible to allow the waiting thread to make progress. In addition, QB-SyncFollow passes the token forward immediately if a thread starts spinning on a lock.

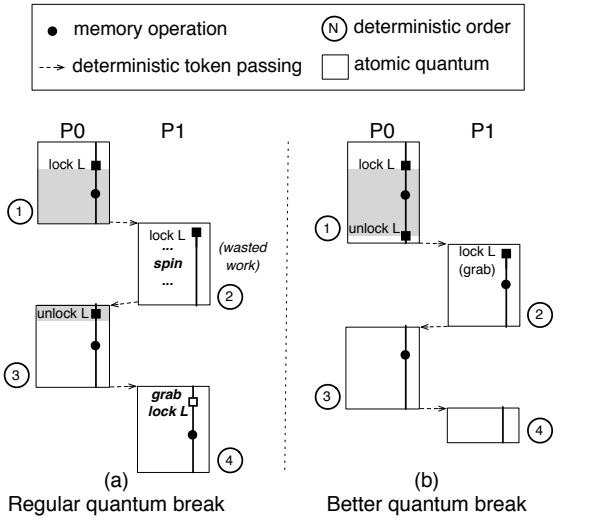

Figure 8. Example of a situation when better quantum breaking policies leads to better performance.

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 other threads 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 in all DMP techniques, but also reduces the amount of work wasted by squashes in DMP-TM and DMP-TMFwd. We call this quantum building heuristic *QB-Sharing*.

In addition, we explore a combination of QB-SyncFollow and QB-Sharing, which we refer to as *QB-SyncSharing*. This quantum building strategy monitors synchronization events and sharing behavior. QB-SyncSharing determines the end of a quantum whenever either of the other two techniques would have decided to do so.

4. Implementation Issues

As seen in the previous section, implementing a DMP system requires a mechanism to deterministically break the execution into

quanta and mechanisms to guarantee the properties of deterministic serialization. A DMP system could have all these mechanisms completely in hardware, completely in software or even as a mix of hardware and software components. The trade-off is one of complexity versus performance. A hardware-only implementation offers better performance but requires changes to the multiprocessor hardware. Conversely, a software-only implementation performs worse but does not require special hardware. In this section, we discuss the relevant points in each implementation.

4.1 Hardware-Only Implementation

Quantum Building: The simplest quantum building policy, QB-Count, is implemented by counting dynamic instructions as they retire and placing a quantum boundary when the desired quantum size is reached. QB-SyncFollow requires access to information about synchronization, which is obtained by a compiler or annotations in the synchronization libraries. QB-Sharing requires monitoring memory accesses and determining whether they are to shared data or not, which is done using the sharing table (Section 3.2.1), discussed later in this section. Finally, QB-SyncSharing is exactly a logical *OR* of the decision made by QB-SyncFollow and QB-Sharing. Regardless of the quantum building policy used, depending upon the consistency model of the underlying hardware, threads must perform a memory fence at the edge of a quantum. It is at these points that inter-thread communication occurs.

Hw-DMP-Serial: DMP-Serial is implemented in hardware with a token that is passed between processors in the deterministic order. The hardware supports multiple tokens, allowing multiple deterministic processes at the same time — each process has its own token.

Hw-DMP-ShTab: The sharing table data-structure used by DMP-ShTab keeps track of the sharing state of data in memory. Our hardware implementation of the sharing table leverages the cache line state maintained by a MESI cache coherence protocol. A line in exclusive or modified state is considered private by the local processor so, as the flowchart in Figure 6 shows, it can be freely read or written by its owner thread without holding the deterministic token. The same applies for a read operation on a line in shared state. Conversely, a thread needs to acquire the deterministic token before writing to a line in shared state, and moreover, all other threads must be at a deterministic point in their execution, e.g., blocked. The state of the entries in the sharing table corresponding to lines that are not cached by any processor is kept in memory and managed by the memory controller, much like a directory in directory-based cache coherence. Note, however, that we do not require directory-based coherence per se. This state is transferred when cache misses are serviced. Nevertheless, directory-based systems can simplify the implementation of Hw-DMP-ShTab even further.

We now address how the state changes of the sharing table happen deterministically. There are three requirements: (1) speculative instructions cannot change the state of the sharing table; (2) a coherence request that changes the state of a cache line can only be performed if the issuer holds the deterministic token; and (3) all nodes need to know when the other nodes are blocked waiting for the deterministic token — this is necessary to implement step 3 in Figure 6. To guarantee (1), speculative instructions that need to change the sharing table can only do so when they are not speculative anymore. To guarantee (2), a coherence request carries a bit that indicates whether the issuer holds the deterministic token: if it does, the servicing node processes the request; if it does not, the servicing node nacks the request if it implies a change in its cache lines, e.g., a downgrade. Finally, (3) is guaranteed by having all processors broadcast when they block or when they unblock.

Alternatively, sharing table implementations can use memory tagging, where the tags represent the sharing information. Moreover, our evaluation (Section 6.2) shows that tracking sharing information at a page granularity does not degrade performance excessively. This suggests a page-level implementation, which is simpler than a line-level implementation.

Hw-DMP-TM and Hw-DMP-TMFwd: On top of standard TM support, a hardware implementation of DMP-TM needs a mechanism to enforce a specific transaction commit order — the deterministic commit order of quanta encapsulated inside transactions. Hw-DMP-TM does that by allowing a transaction to commit only when the processor receives the deterministic token. After a single commit, the processor passes the token to the next processor in the deterministic order. DMP-TMFwd requires more elaborate TM support to allow speculative data to flow from uncommitted quanta earlier in the deterministic order. This is implemented by making the coherence protocol be aware of the data version of quanta, very similarly to versioning protocols used in Thread-Level Speculation (TLS) systems [3]. One interesting aspect of Hw-DMP-TM is that, if a transaction overflow event is made deterministic, it can be used as a quantum boundary, making a bounded TM implementation perfectly suitable for a DMP-TM system. Making transaction overflow deterministic requires making sure that updates to the speculative state of cache lines happen strictly as a function of memory instruction retirement, i.e., updates from speculative instructions cannot be permitted. In addition, it also requires all non-speculative lines to be displaced before an overflow is triggered, i.e., the state of non-speculative lines cannot affect the overflow decision.

The implementation choices in a hardware-only DMP system also have performance versus complexity trade-offs. DMP-TMFwd is likely to offer better performance, but it requires mechanisms for speculative execution, conflict detection and memory versioning, whereas DMP-ShTab performs a little worse but does not require speculation.

4.2 Software-Only Implementation

A DMP system can also be implemented using a compiler or a binary rewriting infrastructure. The implementation details are largely similar to the hardware implementations. The compiler builds quanta by sparsely inserting code to track dynamic instruction count in the control-flow-graph — quanta need not be of uniform size as long as the size is deterministic. This is done at the beginning and end of function calls, and at the tail end of CFG back edges. The inserted code tracks quantum size and, when the target size has been reached, it calls back to a runtime system, which implements the various DMP techniques. Sw-DMP-Serial is supported in software by implementing the deterministic token as a queuing lock. For Sw-DMP-ShTab, the compiler instruments every load and store to call back to the runtime system, and the runtime system implements the logic shown in Figure 6 and the table itself is kept in memory. It is also possible to implement a DMP system using software TM techniques, but we did not evaluate such a scheme in this paper.

5. Experimental Setup

We evaluate both hardware and software implementations of a DMP system. We use the SPLASH2 [22] and PARSEC [2] benchmark suites and run the benchmarks to completion. Some benchmarks were not included due to infrastructure problems such as out of memory errors and other system issues (e.g., lack of 64-bit compatibility). Note that the input sizes for the software implementation experiments are typically larger than the ones used in the sim-

ulation runs due to simulation time constraints. We ran our native experiments on a machine with dual Intel Xeon E5462 quad-core 64-bit processors (8 cores total) clocked at 2.8 GHz, with 8GB of memory running Linux 2.6.24. In the sections below we describe the evaluation environment for each implementation category.

5.1 Hw-DMP

We assess the performance trade-offs of the different hardware implementations of a DMP systems with a simulator written using PIN [14]. The model includes the effects of serialized execution[2], quantum building, memory conflicts, speculative execution squashes and buffering for a single outstanding transaction per thread in the transactional memory support. Note that even if execution behavior is deterministic, performance may not be deterministic. Therefore we run our simulator multiple times, average the results and provide error bars showing the 90% confidence interval for the mean. While the model simplifies some microarchitectural details, the specifics of the various Hw-DMP system implementations are modeled in detail. To reduce simulation time, our model assumes that the IPCs (including squashed instructions) of all the different DMP modes are the same. This reasonable assumption allows us to compare performance between different Hw-DMP schemes using our infrastructure. Note that the comparison baseline (nondeterministic parallel execution) also runs on our simulator.

5.2 Sw-DMP

We evaluate the performance impact of a software-based DMP system by using a compiler pass written for LLVM v2.2 [11]. Its main transformations are described in Section 4.2. The pass is executed after all other compiler optimizations. Once the object files are linked to the runtime environment, LLVM does another complete link-time optimization pass to inline the runtime library with the main object code.

The runtime system provides a custom pthread-compatible thread management and synchronization API. Finally, the runtime system allows the user to control the maximum quantum size and the granularity of entries in the sharing table. We configured these parameters on a per-application basis, with quantum sizes varying from 10,000 to 200,000 instructions, and sharing table entries from 64B to 4KB. Our Sw-DMP experiments run on real hardware, so we took multiple runs, averaged their running time and provided error bars in the performance plot showing the 90% confidence interval for the mean. The focus of this paper is on the DMP concepts and their hardware implementations, so we omit a detailed description and evaluation of our software-only implementation.

6. Evaluation

We first show the scalability of our hardware proposals: in the best case, Hw-DMP-ShTab has negligible overhead compared to nondeterministic parallel execution with 16 threads, while the more aggressive Hw-DMP-TMFwd reduces Hw-DMP-ShTab's overheads by 20% on average. We then examine the sensitivity of our hardware proposals to changes in quantum size, conflict detection granularity, and quantum building strategy, presenting only a subset of benchmarks (and averages) for space reason. Finally, we show the scalability of our software-only implementation of DMP-ShTab and demonstrate that Sw-DMP-ShTab does not unduly limit performance scalability. We believe that Hw-DMP-ShTab represents a good trade-off between performance and complexity, and that Sw-DMP-ShTab is fast enough to be useful for debugging and, depending on the application, deployment purposes.

[2] Note that the simulation actually serializes quanta execution functionally, which affects how the system executes the program. This accurately models the effects of quanta serialization on application behavior.

6.1 Hw-DMP: Performance and Scalability

Figure 9 shows the scalability of our techniques compared to the nondeterministic, parallel baseline. We ran each benchmark with 4, 8 and 16 threads, and QB-SyncSharing producing 1,000-instruction quanta. As one would expect, Hw-DMP-Serial exhibits slowdown nearly linear with the number of threads. The degradation can be sublinear because DMP affects only the parallel behavior of an application's execution. Hw-DMP-ShTab has 38% overhead on average with 16 threads, and in the few cases where Hw-DMP-ShTab has larger overheads (e.g. `lu-nc`), Hw-DMP-TM provides much better performance. For an additional cost in hardware complexity, Hw-DMP-TMFwd, with an average overhead of only 21%, provides a consistent performance improvement over Hw-DMP-TM. Hw-DMP-ShTab and the TM-based schemes all scale sublinearly with the number of processors. The overhead for TM-based schemes is flat for most benchmarks, suggesting that a TM-based system would be ideal for larger DMP systems. Thus, with the right hardware support, the performance of deterministic execution can be very competitive with nondeterministic parallel execution.

6.2 Hw-DMP: Sensitivity Analysis

Figure 10 shows the effects of changing the maximum number of instructions included in a quantum. Again, we use the QB-SyncSharing scheme, with line-granularity conflict detection. Increasing the size of quanta consistently degrades performance for the TM-based schemes, as larger quanta increase the likelihood and cost of aborts since more work is lost. The ability of Hw-DMP-TMFwd to avoid conflicts helps increasingly as the quanta size gets larger. With Hw-DMP-ShTab, the effect is more application dependent: most applications (e.g. `vlrend`) do worse with larger quanta, since each quantum holds the deterministic token for longer, potentially excluding other threads from making progress. For `lu-nc`, however, the effect is reversed: `lu-nc` has relatively large communication-free regions per quantum, allowing each thread to make progress without holding the deterministic token. Smaller quanta sizes force a thread to wait for the deterministic token sooner, lessening this effect. On average, however, Hw-DMP-Serial is less affected by quantum size.

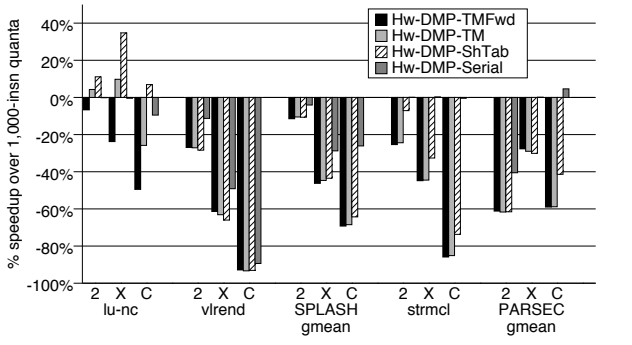

Figure 10. Performance of 2,000 (2), 10,000 (X) and 100,000 (C) instruction quanta, relative to 1,000 instruction quanta.

Figure 11 compares conflict detection at cache line (32-byte) and page (4096-byte) granularity. Increasing the conflict detection granularity decreases the performance of the TM-based schemes, as they suffer more (likely false) conflicts. The gap between Hw-DMP-TMFwd and Hw-DMP-TM grows as the former can avoid some of the conflicts by forwarding values. Hw-DMP-Serial is unaffected, because it does no conflict detection. With Hw-DMP-ShTab, a coarser granularity can lead to more blocking (e.g. `lu-nc`

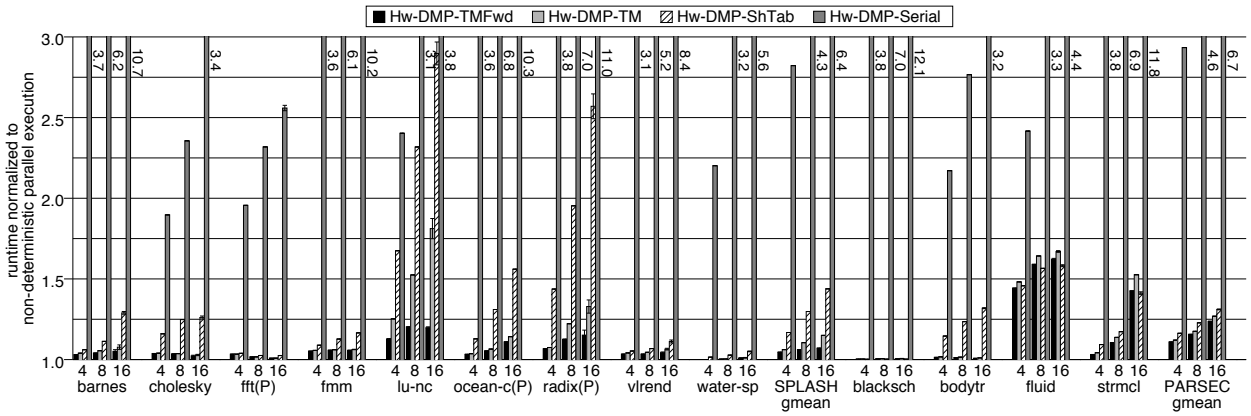

Figure 9. Runtime overheads with 4, 8 and 16 threads. (P) indicates page-level conflict detection; line-level otherwise.

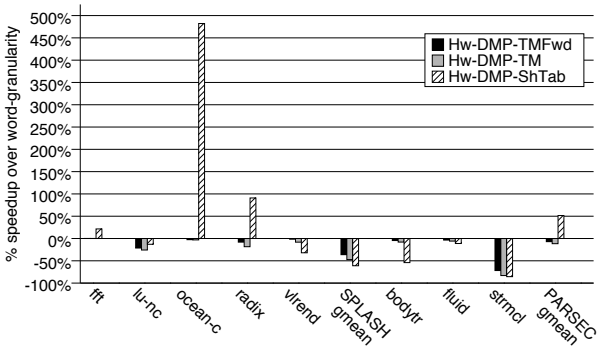

Figure 11. Performance of page-granularity conflict detection, relative to line-granularity.

and `streamcluster`) but can also, surprisingly, improve performance by making it faster to privatize or share large regions of memory (e.g. `radix`, `ocean-c`). This suggests a pro-active privatization/sharing mechanism to improve the performance of Hw-DMP-ShTab. On average, our results show that exploiting existing virtual memory support to implement Hw-DMP-ShTab could be quite effective.

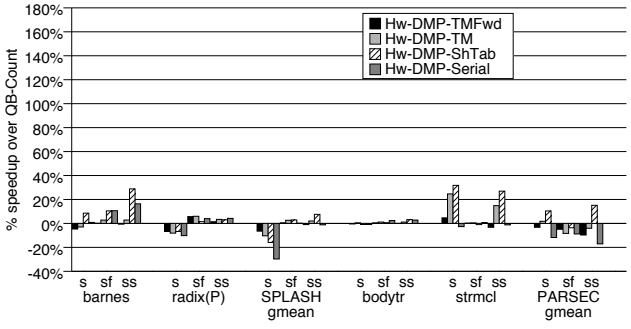

Figure 12. Performance of QB-Sharing (s), QB-SyncFollow (sf) and QB-SyncSharing (ss) quantum builders, relative to QB-Count, with 1,000-insn quanta.

Figure 12 shows the performance effect of different quantum building strategies. Smarter quantum builders generally do not improve performance much over the QB-Count 1,000-instruction baseline, as QB-Count produces such small quanta that heuristic breaking cannot substantially accelerate progress along the application's critical path. With 10,000-instruction quanta (Figure 13), the effects of the different quantum builders are more pronounced. In general, the quantum builders that take program synchronization into account (QB-SyncFollow and QB-SyncSharing) outperform those that do not. Hw-DMP-Serial and Hw-DMP-ShTab perform better with smarter quantum building, while the TM-based schemes are less affected, as TM-based schemes recover more parallelism. QB-Sharing works well with Hw-DMP-ShTab and PARSEC, and works synergistically with synchronization-aware quantum building: QB-SyncSharing often outperforms QB-SyncFollow (as with `barnes`).

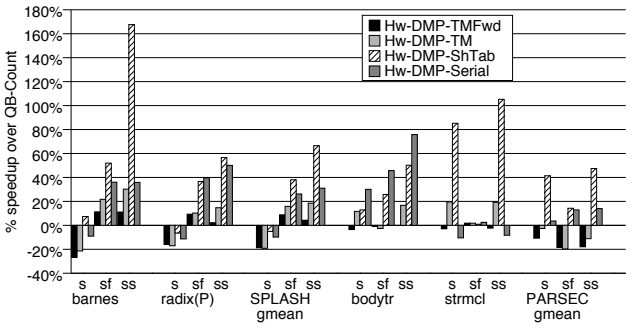

Figure 13. Performance of quantum building schemes, relative to QB-Count, with 10,000-insn quanta.

6.3 Hw-DMP: Characterization

Table 1 provides more insight into our sensitivity results. For Hw-DMP-TM, with both line- and page-level conflict detection, we give the average read- and write-set sizes (which show that our TM buffering requirements are modest), and the percentage of quanta that suffer conflicts. The percentage of conflicts is only roughly correlated with performance, as not all conflicts are equally expensive. For the Hw-DMP-ShTab scheme, we show the amount of execution overlap of a quantum with other quanta (parallel prefix), as a percentage of the average quantum size. This metric is highly correlated with performance: the more communication-free work exists at the beginning of each quantum, the more progress a thread can make before needing to acquire the deterministic token. Finally, we give the average quantum size and the percentage of quanta breaks caused by the heuristic of each of the quantum builders, with

a 10,000-instruction maximum quanta size. The average quantum size for QB-Count is uniformly very close to 10,000 instructions, so we omit those results. Since the average quantum size for QB-SyncFollow is generally larger than that for QB-Sharing, and the former outperforms the latter, we see that smaller quanta are not always better: it is important to choose quantum boundaries well, as QB-SyncFollow does.

6.4 Sw-DMP: Performance, Scalability and Characterization

Figure 14 shows the performance and scalability of Sw-DMP-ShTab compared to the parallel baseline. We see two classes of trends, slowdowns that increase with the number of threads (e.g. `barnes`) and slowdowns that don't increase much with the number of threads (e.g. `fft`). For benchmarks in the latter class, adding more threads substantially improves raw performance. Even for benchmarks in the former class, while adding threads does decrease raw performance compared to the corresponding parallel baseline, the slowdown is sublinear in the number of threads. Thus, adding threads still results in an improvement in raw performance. In summary, this data shows that Sw-DMP-ShTab does not unduly limit performance scalability for multithreaded applications.

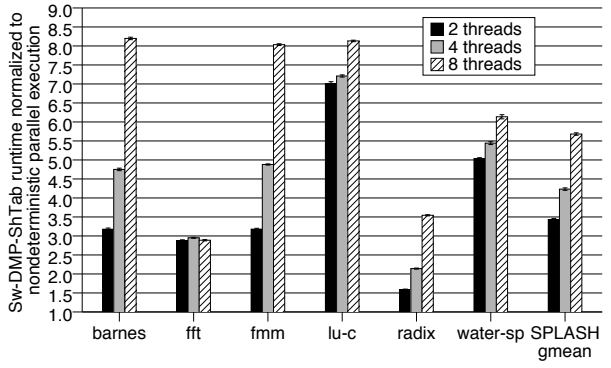

Figure 14. Runtime of Sw-DMP-ShTab with 2, 4 and 8 threads relative to nondeterministic parallel execution.

7. Discussion

Our evaluation of the various DMP schemes leads to several conclusions. At the highest level, the conclusion is that deterministic execution in a multiprocessor environment *is* achievable on future systems with little, if any, performance degradation. The simplistic Hw-DMP-Serial has a geometric mean slowdown of 6.5X on 16 threads. By orchestrating communication with Hw-DMP-ShTab, this slowdown reduces to a geometric mean of 37% and often less than 15%. By using speculation with Hw-DMP-TM, we were able to reduce the overhead to a geometric mean of 21% and often less than 10%. Through the addition of forwarding with Hw-DMP-TMFwd, the overhead of deterministic execution is less than 15% and often less than 8%. Finally, software solutions can provide deterministic execution with a performance cost suitable for debugging on current generation hardware, and depending upon the application, deployment.

Now that we have defined what deterministic shared memory multiprocessing is, shown how to build efficient hardware support for it, and demonstrated that software solutions can be built for current generation hardware, we now discuss several additional points. These are: (1) performance, complexity and energy tradeoffs; (2) support for debugging; (3) interaction with operating system and I/O nondeterminism; and (4) making deterministic execution portable for deployment.

Implementation Trade-offs. Our evaluation showed that using speculation in the hardware-based implementation pays off in terms of performance. However, speculation potentially wastes energy, requires complex hardware and has implications in system design, since some code, such as I/O and parts of an operating system, cannot execute speculatively. Fortunately, DMP-TM, DMP-ShTab and DMP-Serial can coexist in the same system. One easy way to coexist is to switch modes at a deterministic boundary in the program (e.g., the edge of a quanta). More interestingly, a DMP system can be designed to support multiple modes co-existing simultaneously. This allows a DMP system to use the most convenient approach depending on what code is running, e.g., using speculation (DMP-TM) in user code and avoiding it (DMP-ShTab) in kernel code.

DMP systems could also be built in a hybrid hardware-software fashion, instead of a purely hardware or software implementation. A hybrid DMP-TM system, for example, could leverage modest hardware TM support while doing quantum building and deterministic ordering more flexibly in software with low performance cost. A hybrid DMP-ShTab system could efficiently implement the sharing table by leveraging modern cache coherence protocols; exposing coherence state transitions could enable a simple high performance hybrid DMP-ShTab implementation in the near future.

Supporting Debugging Instrumentation. In order to enable a debugging environment in a DMP system, we need a way of allowing the user to instrument code for debugging while preserving the interleaving of the original execution. To accomplish this, implementations must support a mechanism that allows a software development system to mark code as being inserted for instrumentation purposes only; such code will not affect quantum building, and thus preserves the original behavior.

Dealing with nondeterminism from the OS and I/O. There are many sources of nondeterminism in today's systems, as we discussed in Section 2.2.1. A DMP system hides most of them, allowing many multithreaded programs to run deterministically. Besides hiding the nondeterminism from the microarchitecture structures, a DMP system also hides the nondeterminism of OS thread scheduling by using the deterministic token to provide low-level deterministic thread scheduling. This causes threads to run in the same order on every execution. Nevertheless, challenging sources of nondeterminism remain.

One challenge is that parallel programs can use the operating system to communicate between threads. A DMP system needs to make that communication deterministic. One way to address the problem is to execute OS code deterministically, which was discussed earlier in this section. Alternatively, a layer between the operating system and the application can be utilized to detect communication and synchronization via the kernel and provide it within the application itself. This is the solution employed by our software implementation.

Another challenge is that many operating system API calls allow nondeterministic outcomes. System calls such as *read* may lead to variations in program execution from run to run, as their API specification itself permits such variation. There are two ways to handle nondeterministic API specifications: ignore them — any variation in outcome could also have occurred to sequential code; or fix them, by providing alternative APIs that are deterministic. With *read*, a solution is to always return the maximum amount of data requested until EOF.

The final, and perhaps most difficult challenge is that the real world is simply nondeterministic. Ultimately, programs interact with remote systems and users, which are all nondeterministic and

	Hw-DMP Implementation – 1,000-insn quanta						QB Strategy – 10,000-insn quanta†					
	TM				ShTab							
	Line		Page		Line	Page	SyncFollow		Sharing		SyncSharing	
Benchmark	R/W set sz.	% conf.	R/W set sz.	% conf.	% Q overlap	% Q overlap	Avg. Q sz.	% sync brk.	Avg. Q sz.	% shr. brk.	Avg. Q sz.	% sync brk.
barnes	27/9	37	9/2	64	47	46	5929	42	4658	67	5288	54
cholesky	14/6	23	3/1	39	31	38	6972	30	3189	94	6788	35
fft	22/16	25	3/4	26	19	39	9822	1	3640	62	4677	49
fmm	30/6	51	7/1	69	33	29	8677	15	4465	65	5615	50
lu-nc	47/33	71	6/4	77	14	16	7616	24	6822	37	6060	42
ocean-c	46/15	28	5/2	34	5	46	5396	49	3398	73	3255	73
radix	16/20	7	3/7	13	31	42	8808	15	3346	71	4837	57
vlrend	27/8	38	7/1	50	41	39	7506	28	7005	45	6934	38
water-sp	32/19	19	5/1	45	40	37	7198	5	5617	30	6336	20
SPLASH amean	30/16	31	5/2	44	29	35	7209	27	4987	57	5363	48
blacksch	28/9	8	14/1	10	48	48	10006	<1	9163	10	9488	7
bodytr	11/4	16	3/2	28	39	19	7979	25	7235	31	6519	37
fluid	41/8	76	8/2	75	43	40	871	98	2481	95	832	99
strmcl	36/5	28	10/2	91	60	12	9893	1	1747	79	2998	77
PARSEC amean	29/6	36	9/1	51	45	30	7228	19	5156	54	3880	64

Table 1. Characterization of hardware-DMP results. † Same granularity as used in Figure 9

might affect thread interleaving. It may appear, then, that there is no hope for building a deterministic multiprocessor system because of this, but that is not the case. When multithreaded code synchronizes, that is an opportunity to be deterministic from that point on, since threads are in a known state. Once the system is deterministic and the interaction with the external world is considered part of the input, it is much easier to write, debug and deploy reliable software. This will likely encourage programmers to insert synchronization around I/O, in order to make their applications more deterministic, and hence more reliable.

Support for deployment. We contend that deterministic systems should not just be used for development, but for deployment as well. We believe systems in the field should behave like systems used for testing. The reason is twofold. First, developers can have a higher confidence their programs will work correctly once deployed. Second, if the program does crash in the field, then deterministic execution provides a meaningful way to collect and replay crash history data. Supporting deterministic execution across different physical machines places additional constraints on the implementation. Quanta must be built the same across all systems. This means machine-specific effects cannot be used to end quanta (e.g., micro-op count, or a full cache-set for bounded TM-based implementations). Furthermore, passing the deterministic token across processors must be the same for all systems. This suggests that DMP hardware should provide the core mechanisms, and leave the quanta building and scheduling control up to software.

8. Related Work

One way to obtain a deterministic parallel program is to write it using a deterministic parallel programming model, like stream programming languages such as StreamIt [21] and implicitly parallel languages such as Jade [18]. In fact, the need to improve multiprocessor programmability due to the shift to multicore architectures is causing renewed interest in deterministic implicitly parallel programming models [8]. However, several of these models tend to be domain-specific. For generality, our focus is on dealing with nondeterminism in *arbitrary* shared memory parallel programs.

Past work on dealing with nondeterminism in shared memory multiprocessors has focused primarily on deterministically replaying an execution for debugging. The idea is to record a log of the ordering of events that happened in a parallel execution and later replay the execution based on the log. There are several previously proposed software-based systems [9, 12, 19] for recording and replaying a multithreaded execution. The software-based approaches typically suffer from high overheads or limitations on the types of events recorded — for example, synchronization events only, which is not enough to reproduce the outcome of races. Several researchers have addressed these limitations and overheads by devising hardware-based record and replay mechanisms [1, 17, 23]. Hardware based approaches actually record memory events at the level of memory operations more efficiently, allowing more accurate and faster record and replay functionality.

Most of the follow-up research has been on further reducing the log size and performance impact of deterministic replay. Examples of such efforts are Strata [16] and FDR2 [24], which exploit redundancy in the memory race log. Very recent advances are ReRun [7] and DeLorean [15], which aim at reducing log size and hardware complexity. ReRun is a hardware memory race recording mechanism that records periods of the execution without memory communication (the same phenomenon leveraged by DMP-ShTab). This new strategy requires little hardware state and produces a small race log. In DeLorean, instructions are executed as blocks (or chunks), and the commit order of blocks of instructions is recorded, as opposed to each instruction. In addition, DeLorean uses pre-defined commit ordering (albeit with chunk size logging in special cases) to further reduce the memory ordering log.

DMP makes memory communication deterministic, therefore it *completely eliminates* memory communication logging, and consequently does not need to provide hardware to record and replay execution at all — execution will always be the same as long as the inputs are the same. Since DeLorean attempts to control nondeterminism to reduce log size to a minimum, it is similar in spirit to our work. However, DeLorean still has a log and requires speculation, whereas DMP-ShTab, for example, has no need for speculation. Finally, while a chunk in DeLorean and a quantum in DMP are a similar concept, in DeLorean chunks are created blindly, whereas in DMP we propose to create quanta more intelligently to better match application sharing and synchronization characteristics.

Some of the mechanisms used in DMP-TM (and DMP-TMFwd) are similar to mechanisms employed in Thread-Level Speculation [4, 10, 20] systems, though DMP employs them for almost the opposite purpose. These mechanisms are: ordered quantum commits in DMP-TM, and the speculative value forwarding optimization in DMP-TMFwd. TLS starts with a (deterministic) sequential program and then speculatively executes it in parallel while guaran-

teeing the original sequential semantics of the program. DMP-TM does the opposite: it starts with a (nondeterministic) explicitly parallel program, effectively serializes (making it deterministic) and then uses speculative execution to efficiently recover parallelism while preserving deterministic behavior.

In summary, DMP is not a record and replay system, it provides fully deterministic shared memory communication. As such, it can be used for debugging because it offers repeatability by default. We argue, however, that deterministic execution is useful for more than just debugging: it can be deployed as well.

9. Conclusion

In this paper we made the case for fully deterministic shared memory multiprocessing. After quantitatively showing how nondeterministic current systems are, we have shown that the key requirement to support deterministic execution is deterministic communication via shared memory. Fortunately, this requirement still leaves room for efficient implementations. We described a range of implementation alternatives, in both hardware and software, with varying degrees of complexity and performance cost. Our simulations show that a hardware implementation of a DMP system can have negligible performance degradation over nondeterministic systems. We also briefly described our compiler-based software-only DMP system and show that while the performance impact is significant, it is quite tolerable for debugging.

While the benefits for debugging are obvious, we suggest that parallel programs should always run deterministically. Deploying deterministic execution in the field has the potential to increase reliability of parallel code, as the system in the field would behave similarly to in-house testing environments, and to allow a more meaningful collection of crash information.

In conclusion, we have shown that, perhaps contrary to popular belief, a shared memory multiprocessor system can execute programs deterministically with little performance cost. We believe that deterministic multiprocessor systems are a valuable goal, as they, besides yielding several interesting research questions, abstract away several difficulties in writing, debugging and deploying parallel code.

Acknowledgments

We thank the anonymous reviewers for their invaluable comments. We also thank Karin Strauss, Dan Grossman, Susan Eggers, Martha Kim, and Andrew Putnam from the University of Washington for their feedback on the manuscript. Finally, we thank Jim Larus and Shaz Qadeer from Microsoft Research for very helpful discussions. Brandon Lucia was partially supported by the Clairmont L. Egtvedt Fellowship and The Faithful Steward Endowed Fellowship.

References

[1] D. Bacon and S. Goldstein. Hardware-Assisted Replay of Multiprocessor Programs. In *Workshop on Parallel and Distributed Debugging*, 1991.

[2] C. Bienia, S. Kumar, J. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. Technical report, Princeton University, January 2008.

[3] S. Gopal, T. N. Vijaykumar, J. E. Smith, and G. S. Sohi. Speculative Versioning Cache. In *International Symposium on High Performance Computer Architecture*, 1998.

[4] L. Hammond, M. Willey, and K. Olukotun. Data Speculation Support for a Chip Multiprocessor. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[5] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional Memory Coherence and Consistency. In *International Symposium on Computer Architecture*, 2004.

[6] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *International Symposium on Computer Architecture*, 1993.

[7] D. Hower and M. Hill. Rerun: Exploiting Episodes for Lightweight Memory Race Recording. In *International Symposium on Computer Architecture*, 2008.

[8] W. Hwu, S. Ryoo, Sain-Zee Ueng, J.H. Kelm, I. Gelado, S.S. Stone, R.E. Kidd, S.S. Baghsorkhi, A.A. Mahesri, S.C. Tsao, N. Navarro, S.S. Lumetta, M.I. Frank, and S.J. Patel. Implicitly Parallel Programming Models for Thousand-Core Microprocessors. In *Design Automation Conference*, 2007.

[9] J. Choi and H. Srinivasan. Deterministic Replay of Java Multithreaded Applications. In *SIGMETRICS Symposium on Parallel and Distributed Tools*, 1998.

[10] V. Krishnan and J. Torrellas. A Chip-Multiprocessor Architecture with Speculative Multithreading. *IEEE Transactions on Computers*, September 1999.

[11] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *International Symposium on Code Generation and Optimization*, 2004.

[12] T.J. Leblanc and J.M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE Transactions on Computers*, April 1987.

[13] E. A. Lee. The problem with threads. *IEEE Computer*, May 2006.

[14] 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 *Conference on Programming Language Design and Implementation*, 2005.

[15] P. Montesinos, L. Ceze, and J. Torrellas. DeLorean: Recording and Deterministically Replaying Shared-Memory Multiprocessor Execution Efficiently. In *International Symposium on Computer Architecture*, 2008.

[16] S. Narayanasamy, C. Pereira, and B. Calder. Recording Shared Memory Dependencies Using Strata. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[17] S. Narayanasamy, G. Pokam, and B. Calder. BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging. In *International Symposium on Computer Architecture*, 2005.

[18] M. Rinard and M. Lam. The design, implementation, and evaluation of Jade. *ACM Transactions on Programming Languages and Systems*, May 1988.

[19] M. Ronsee and K. De Bosschere. RecPlay: A Fully Integrated Practical Record/Replay System. *ACM Transactions on Computer Systems*, 1999.

[20] G. S. Sohi, S. E. Breach, and T. N. Vijayakumar. Multiscalar Processors. In *International Symposium on Computer Architecture*, June 1995.

[21] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A Language for Streaming Applications. In *International Conference on Compiler Construction*, 2002.

[22] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *International Symposium on Computer Architecture*, 1995.

[23] M. Xu, R. Bodik, and M. Hill. A "Flight Data Recorder" for Enabling Full-System Multiprocessor Deterministic Replay. In *International Symposium on Computer Architecture*, 2003.

[24] M. Xu, M. Hill, and R. Bodik. A Regulated Transitive Reduction (RTR) for Longer Memory Race Recording. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.