DMP: Deterministic Shared-Memory Multiprocessing

Shared-memory multicore and multiprocessor systems are nondeterministic, which frustrates debugging and complicates testing of multithreaded code, impeding parallel programming's widespread adoption. The authors propose fully deterministic shared-memory multiprocessing that not only enhances debugging by offering repeatability by default, but also improves the quality of testing and the deployment of production code. They show that determinism can be provided with little performance cost on future hardware.

Joseph Devietti
Brandon Lucia
Luis Ceze
Mark Oskin
University of Washington

Developing multithreaded software has proven to be much more difficult than writing single-threaded code. One major challenge is that current multicore processors execute multithreaded code nondeterministically: 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—for example, other processes executing simultaneously, differences in the operating system's resource allocation, the state of caches, TLBs, buses, and other microarchitectural structures. 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 can't reproduce it during debugging. Because a program can behave differently each time it's run with the same input, assessing test coverage is difficult. For that reason, much of the research on testing parallel programs is devoted to dealing with nondeterminism.

Although researchers have made significant efforts to address nondeterminism, they've mostly focused on deterministically replaying multithreaded execution based on a previously generated log.[1-5] (See the "Related work on dealing with nondeterminism in multithreaded software" sidebar for more information.) In contrast, we make the case for fully deterministic shared-memory multiprocessing (DMP). We show that, with hardware support, we can execute arbitrary shared-memory parallel programs deterministically, with scant performance penalty. For a discussion of determinism's benefits, see the "How determinism benefits multithreaded software development" sidebar.

In this article, we describe a multiprocessor architecture for determinism. This work has led to several follow-on projects; for example, we recently published work on pure software approaches to determinism.[6]

Published by the IEEE Computer Society

Related work on dealing with nondeterminism in multithreaded software

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 occurred in a parallel execution and later replay the execution on the basis of the log. Software-based systems[1-3] typically suffer from high overhead or limitations on the types of events recorded—for example, only synchronization events, which are insufficient to reproduce the outcomes of races. Hardware-based record and replay mechanisms[4-6] address these issues by enabling the recording of low-level events more efficiently, allowing faster and more accurate record and replay.

Most follow-up research has been on further reducing deterministic replay's log size and performance impact. Strata[7] and FDR2[8] exploit redundancy in the memory race log. Recent advances are ReRun[9] and DeLorean,[10] which aim to reduce 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 that DMP-ShTab leverages), requiring little hardware state and producing a small race log. DeLorean executes instructions as blocks (similar to quanta), and only needs to record the commit order of blocks of instructions. Additionally, DeLorean uses predefined commit ordering (albeit with block size logging in special cases) to further reduce the memory ordering log. In contrast, DMP makes memory communication deterministic by construction, eliminating the need for logging.

Similarly to DMP, Kendo,[11] which was concurrently developed with our work, provides log-free deterministic execution. Kendo leverages an application's data-race freedom to provide determinism efficiently and purely in software via a deterministic synchronization library. In contrast, DMP uses hardware support to provide determinism for arbitrary programs, including those with data races.

In this work, we explore enforcing determinism at the execution level, providing generality but requiring architecture modifications. A body of related work proposes a different trade-off: moving to new, deterministic languages such as StreamIt,[12] Jade,[13] and DPJ,[14] while continuing to run on commodity hardware.

References

1. J.-D. Choi and H. Srinivasan, ''Deterministic Replay of Java Multithreaded Applications,'' Proc. *SIGMETRICS Symp. Parallel and Distributed Tools,* ACM Press, 1998, pp. 48-59.
2. T.J. Leblanc and J.M. Mellor-Crummey, ''Debugging Parallel Programs with Instant Replay,'' *IEEE Trans. Computers,* vol. 36, no. 4, 1987, pp. 471-482.
3. M. Ronsee and K. De Bosschere, ''RecPlay: A Fully Integrated Practical Record/Replay System,'' *ACM Transactions on Computer Systems,* vol. 17, no. 2, 1999, pp. 133-152.
4. D.F. Bacon and S.C. Goldstein, ''Hardware-Assisted Replay of Multiprocessor Programs,'' *Proc. 1991 ACM/ONR Workshop Parallel and Distributed Debugging,* ACM Press, 1991, pp. 194-206.
5. S. Narayanasamy, G. Pokam, and B. Calder, ''BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging,'' *Proc. 32nd Int'l Symp. Computer Architecture* (ISCA 05), IEEE CS Press, 2005, pp. 284-295.
6. M. Xu, R. Bodik, and M.D. Hill, ''A 'Flight Data Recorder' for Enabling Full-System Multiprocessor Deterministic Replay,'' *Proc. 30th Int'l Symp. Computer Architecture* (ISCA 03), IEEE CS Press, 2003, pp. 122-135.
7. S. Narayanasamy, C. Pereira, and B. Calder, ''Recording Shared Memory Dependencies Using Strata,'' *Proc. 12th Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 06), ACM Press, 2006, pp. 229-240.
8. M. Xu, M. Hill, and R. Bodik, ''A Regulated Transitive Reduction (RTR) for Longer Memory Race Recording,'' *Proc. 12th Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 06), ACM Press, 2006, pp. 49-60.
9. D.R. Hower and M.D. Hill, ''Rerun: Exploiting Episodes for Lightweight Memory Race Recording,'' *Proc. 35th Int'l Symp. Computer Architecture* (ISCA 08), IEEE CS Press, 2008, pp. 265-276.
10. P. Montesinos, L. Ceze, and J. Torrellas, ''DeLorean: Recording and Deterministically Replaying Shared-Memory Multiprocessor Execution Efficiently,'' *Proc. 35th Int'l Symp. Computer Architecture* (ISCA 08), IEEE CS Press, 2008, pp. 289-300.
11. M. Olszewski, J. Ansel, and S. Amarasinghe, ''Kendo: Efficient Deterministic Multithreading in Software,'' *Proc. 14th Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 09), ACM Press, 2009, pp. 97-108.
12. W. Thies, M. Karczmarek, and S.P. Amarasinghe, ''StreamIt: A Language for Streaming Applications,'' *Proc. 11th Int'l Conf. Compiler Construction,* LNCS 2304, Springer, 2002, pp. 179-196.
13. M.C. Rinard and M.S. Lam, ''The Design, Implementation, and Evaluation of Jade,'' *ACM Trans. Programming Languages and Systems,* vol. 20, no. 3, 1998, pp. 483-545.
14. R.L. Bocchino Jr. et al., ''A Type and Effect System for Deterministic Parallel Java,'' *Proc. 24th ACM SIGPLAN Conf. Object Oriented Programming Systems Languages and Applications* (OOPSLA 09), 2009, pp. 97-116.

Defining deterministic parallel execution

We define a DMP system as a computer system that

- executes multiple threads that communicate via shared memory, and
- produces 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.

How determinism benefits multithreaded software development

Deterministic execution could potentially improve the entire multithreaded software development process (see Figure A). In a development environment, determinism allows for repeatable debugging, which in turn enables reversible debugging for multithreaded programs. Determinism also acts as a powerful lever for thread analysis tools (such as race detectors) because any properties verified for an execution with a given input will continue to hold for all future executions of that input. Determinism simplifies testing by eliminating the need to stress test a program by running it repeatedly with the same inputs to expose different thread interleavings; instead, thread interleavings can be controlled in a more principled fashion by modifying inputs. However, the greatest rewards accrue when parallel programs always execute deterministically in the field. Deployed determinism has the potential to make testing more valuable, as execution in the field will better resemble in-house testing, and to allow for easier reproduction of bugs from the field back in a debugging environment. Ultimately, determinism will enable the deployment of more reliable parallel software.

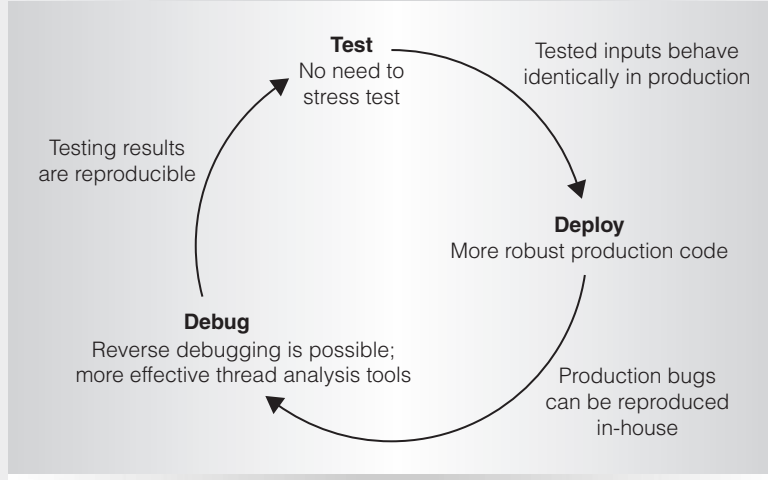

Figure A. The benefits of determinism throughout the development cycle for multithreaded software.

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, this interleaving has several aspects that are irrelevant for ensuring deterministic behavior. It's not important which particular global interleaving is chosen, as long as it's always the same. Also, if two instructions don't communicate, the system can swap their execution order with no observable effect on program behavior. The key to deterministic execution is that all communication between threads is precisely the same for every execution.

To guarantee deterministic interthread communication, each dynamic instance of a load (*consumer*) must read data produced from the same dynamic instance of another store (*producer*). The producer and consumer need not be in the same thread, so this communication happens via shared memory. Interestingly, multiple global interleavings can lead to the same communication between instructions—that is, they are communication-equivalent interleavings. Any communication-equivalent interleaving will yield the same program behavior. To guarantee deterministic behavior, then, we must carefully control only the behavior of load and store operations that cause communication between threads. This insight is the key to efficient deterministic execution.

Enforcing Deterministic Execution

In this section, we describe how to build a deterministic multiprocessor system, focusing on the key mechanisms. We begin with a basic naïve approach and then refine this

simple technique into progressively more efficient organizations. We discuss specific implementations later.

Basic idea: DMP-Serial

As we noted earlier, making multiprocessors deterministic depends on ensuring that the communication between threads is deterministic. The easiest way to accomplish this is to take a nondeterministic parallel execution (see Figure 1a) and allow only one processor at a time to access memory deterministically. We call this the *deterministic serialization of a parallel execution* (see Figure 1b).

The simplest way to implement such serialization is to have each processor obtain a deterministic token before performing a memory operation and, when the memory operation is complete, pass the token to the next processor deterministically. A processor blocks whenever it needs to access memory but doesn't have the token.

Waiting for the token at every memory operation will cause significant performance degradation for two reasons. First, the processor must wait for and pass the deterministic token. Second, the serialization removes the performance benefits of parallel execution.

We can mitigate the synchronization overhead of token passing by synchronizing at a coarser granularity, letting each processor execute a finite, deterministic number of instructions, or a *quantum*, before passing the token to the next processor (Figure 1c). We call such a system *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 into fixed instruction counts (for example, 10,000).

Notably, DMP doesn't interfere with application-level synchronization (for example, DMP introduces no deadlocks). DMP offers the same memory access semantics as traditional nondeterministic systems. The extra synchronization a DMP system imposes resides below application-level synchronization, and the two don't interact.

Recovering parallelism

Reducing serialization's impact requires enabling parallel execution while preserving determinism. We propose two techniques to recover parallelism from communication-free periods of an execution (Figure 1d). The first technique leverages an invalidation-based cache coherence protocol to identify when interthread communication is about to occur. The second technique uses speculation to allow parallel execution of quanta from different processors, re-executing quanta when determinism might have been violated.

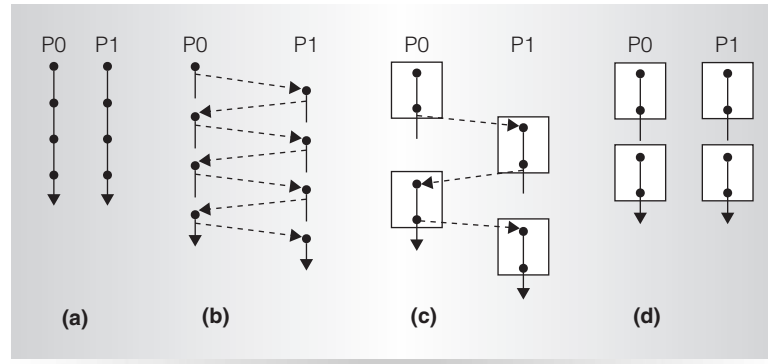

Figure 1. Deterministic serialization of memory operations: nondeterministic parallel execution (a), fine-grain deterministic serialized execution (b), coarse-grain deterministic serialized execution (c), and overlapping communication-free execution periods (d). Dots are memory operations, and dashed arrows are happens-before synchronization.

Leveraging the cache coherence protocol: DMP-ShTab. With DMP-ShTab, we improve deterministic parallel execution's performance by serializing only while threads communicate. We divide each quantum into two parts: a communication-free prefix that executes in parallel with other quanta, and a suffix that, from the first point of communication onward, executes serially. The serial suffix's execution is deterministic because each thread runs serially in an order established 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 interthread communication or, if it doesn't 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

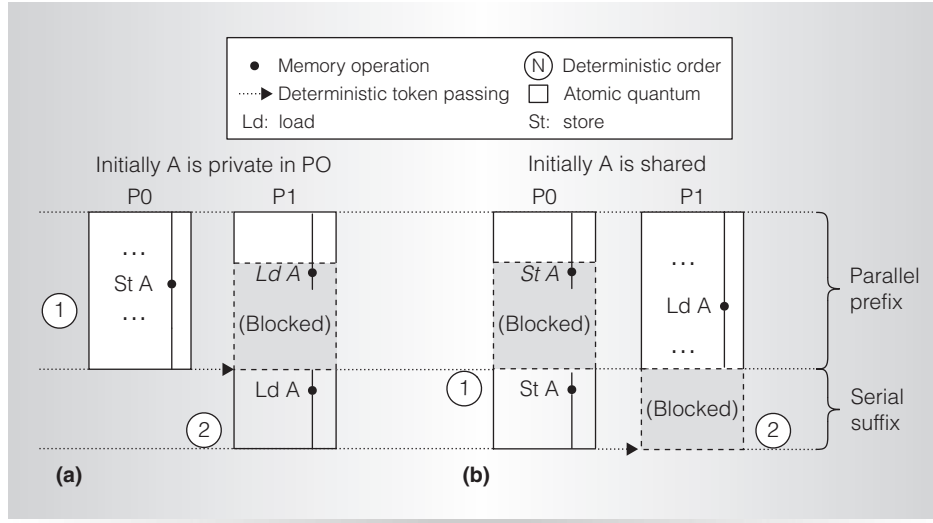

Figure 2. Recovering parallelism by overlapping communication-free execution: remote processor P1 deterministically reads another thread's private data (a), and remote processor P0 deterministically writes to shared data (b).

each quantum because communication is detected deterministically (we'll describe this later).

Interthread communication occurs when a thread writes to shared (or nonprivate) pieces of data. In this case, the system must guarantee that all threads observe such writes at a deterministic point in their execution. Figure 2 illustrates how DMP-ShTab enforces this. There are two important cases: reading data held private by a remote processor, and writing to shared data (privatizing it). Figure 2a shows the first case: when quantum 2 attempts to read data that a remote processor P0 holds private, 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 can't execute until quantum 1 finishes executing. This guarantees that quantum 2 always gets the same data, because quantum 1 might still write to location A before it finishes executing.

Figure 2b shows the second case: 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 can't execute until quantum 2 finishes executing. This is necessary to guarantee that all processors observe the change in location A's state (from shared to privately held by a remote processor) at a deterministic point in their execution. When each thread reaches the end of a quantum, it waits to receive the token before starting its next quantum, which periodically (and deterministically) allows a thread that's waiting for all other threads to be blocked to make progress.

Leveraging support for transactional memory: DMP-TM and DMP-TMFwd. Executing quanta atomically, in isolation, and in a deterministic total order is equivalent to deterministic serialization. To see why, consider a quantum, executed atomically and in isolation, to be a single instruction in the deterministic total order (which is the same as DMP-Serial). Leveraging transactional memory[7,8] can make quanta appear to execute atomically and in isolation. Coupled with a deterministic commit order, this makes execution functionally equivalent to deterministic serialization, while recovering parallelism. Additionally, we need mechanisms to form quanta deterministically and to enforce a deterministic commit order. As Figure 3a shows, speculation lets a quantum run concurrently with other quanta in the system, as long as

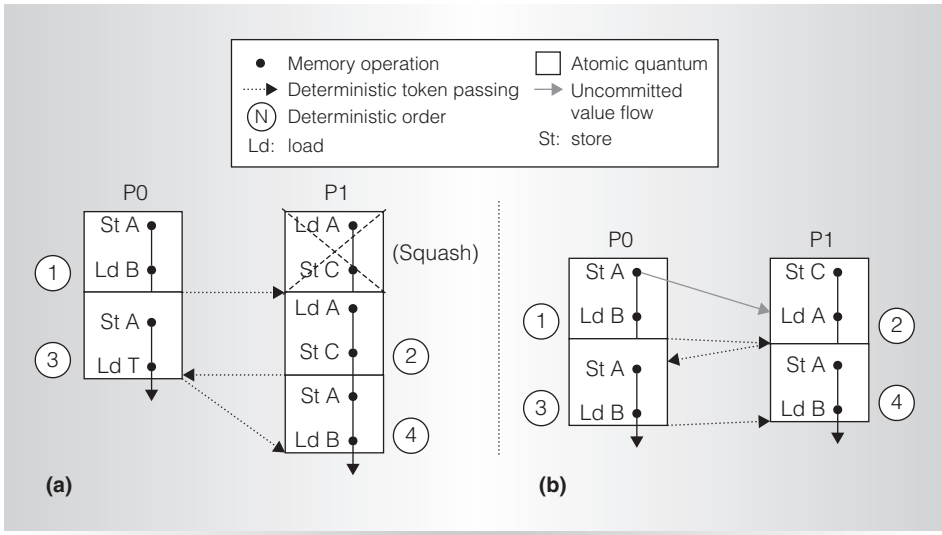

Figure 3. Recovering parallelism by executing quanta as memory transactions (a). Avoiding unnecessary squashes with uncommitted data forwarding (b).

there are no overlapping memory accesses that would violate the original deterministic serialization of memory operations. In case of conflict, the quantum gets squashed and re-executed later in the deterministic total order (see quantum 2 in Figure 3a). Notably, the deterministic total order of quantum commits is a key component in guaranteeing deterministic behavior. We call this system *DMP-TM*.

Another interesting effect of predefined commit ordering is that it lets us use memory renaming to avoid squashes on write-after-write and write-after-read conflicts. For example, in Figure 3a, if quanta 3 and 4 execute concurrently, the store to A in quantum 3 need not squash quantum 4, despite their write-after-write conflict. Having a deterministic commit order also lets us selectively relax isolation, further improving performance by letting quanta forward uncommitted (or speculative) data. This could save a large number of squashes in applications that have more interthread communication. To avoid spurious squashes, we let a quantum fetch speculative data from another uncommitted quantum earlier in the deterministic order. This is shown in Figure 3b, where quantum 2 fetches an uncommitted version of A from quantum 1. Recall that, without forwarding support, 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, because they might have directly or indirectly consumed incorrect data. We call a DMP system that leverages support for transactional memory with forwarding *DMP-TMFwd*.

Building quanta more intelligently

We devised several heuristics to modify our simplistic quantum building algorithm for higher performance. For example, we can end a quantum when an unlock operation occurs. The rationale is that when a thread releases a lock, other threads might be waiting for that lock, so the deterministic token should be sent forward as early as possible to let a waiting thread progress. We discuss other heuristics based on data-sharing patterns in the full conference paper.[9]

Implementing DMP

DMP-ShTab's sharing table data structure 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. The local processor considers a line in an "exclusive" or "modified" state to be private, so its owner thread can freely read or write it without holding the deterministic token.

The same applies for a read operation on a line in shared state. Conversely, a thread must 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 (that is, blocked). The memory controller keeps and manages the state of the entries in the sharing table corresponding to lines that aren't cached by any processor—much like a directory in directory-based cache coherence. However, we don't require directory-based coherence per se. When the system services cache misses, this state is transferred. Nevertheless, directory-based systems can simplify DMP-ShTab's implementation even further.

Both DMP-TM and DMP-TMFwd require, on top of standard transactional memory support, a small amount of machinery to enforce a deterministic transaction commit order by allowing a transaction to commit only when the processor receives the deterministic token. DMP-TMFwd additionally requires more elaborate transactional memory support to allow speculative data to flow from uncommitted quanta earlier in the deterministic order. We accomplish this by making the coherence protocol aware of the data version of quanta, much like in versioning protocols used in thread-level speculation systems.[10] One interesting aspect of DMP-TM is that if we make a transaction overflow event deterministic, we can use it as a quantum boundary, making a bounded transactional memory implementation perfectly suitable for a DMP-TM system. Making transaction overflow deterministic requires that updates to the speculative state of cache lines happen strictly as a function of memory instruction retirement (that is, updates from speculative instructions aren't permitted). In addition, it requires all nonspeculative lines to be displaced before an overflow is triggered (that is, the state of nonspeculative lines can't affect the overflow decision).

Experimental setup

We assess the performance trade-offs of the different hardware implementations of DMP systems with a simulator written using Pin.[11] The model includes the effects of serialized execution, quantum building, memory conflicts, speculative execution squashes, and buffering for a single outstanding transaction per processor. (Our simulations fully serialize quanta execution, affecting how the system executes the program. This accurately models quanta serialization's effects on application behavior). To reduce simulation time, our model assumes that all the different DMP modes have the same IPC (including squashed instructions). This reasonable assumption lets us compare performance between different DMP schemes using our infrastructure. Even if execution behavior is deterministic, performance might not be deterministic. Therefore, we run our simulator multiple times, averaging the results and providing error bars showing the 90 percent confidence interval for the mean. The comparison baseline (nondeterministic parallel execution) also runs on our simulator.

For workloads, we use the Splash2[12] and Parsec[13] benchmark suites and run the benchmarks to completion. We exclude some benchmarks due to infrastructure problems, such as out-of-memory errors and lack of 64-bit compatibility.

Evaluation

Figure 4 shows DMP's scalability compared to the nondeterministic, parallel baseline. We ran each benchmark with 16 threads, and a heuristic-based quantum building strategy producing 1,000-instruction quanta while leveraging synchronization events and sharing information. As you might expect, DMP-Serial exhibits slowdown that's nearly linear with the number of threads. The degradation can be sublinear because DMP affects only the parallel portion of an application's execution. DMP-ShTab has 38 percent overhead on average with 16 threads, and in the few cases where DMP-ShTab has larger overheads (for example, the lu-nc benchmark; see Figure 4), DMP-TM provides much better performance. For an additional hardware complexity cost, DMP-TMFwd, with an average overhead of only 21 percent, provides a consistent performance improvement over DMP-TM. 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

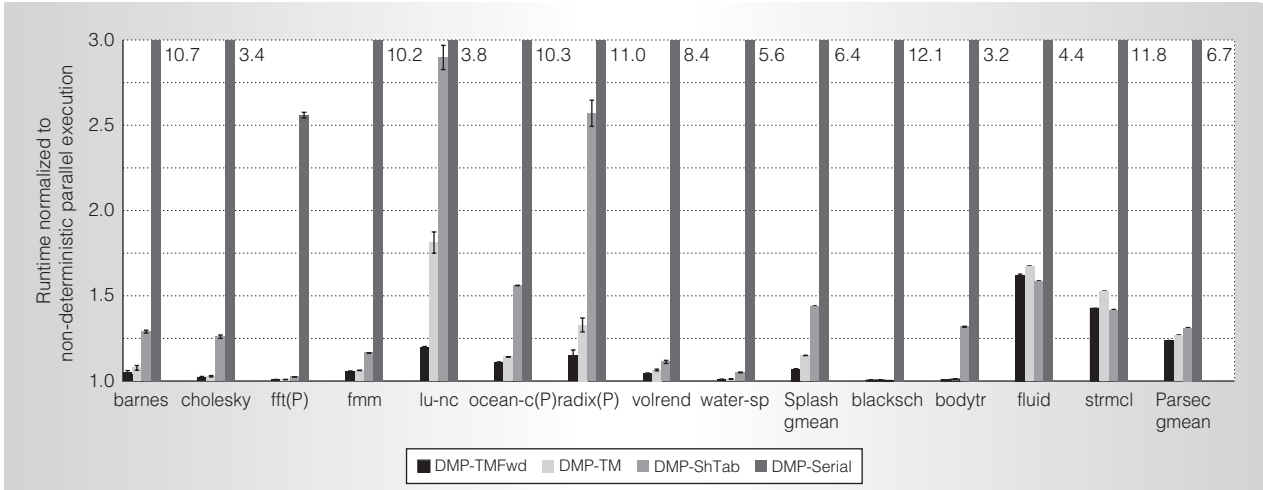

Figure 4. Runtime overheads with 16 threads. (P) indicates page-level conflict detection; otherwise, the conflict detection is at line-granularity.

right hardware support, deterministic execution's performance can compete with nondeterministic parallel execution.

Full-system implications

Beyond the correctness and performance considerations we've presented, we must also consider the DMP architecture's full-system implications.

Implementation trade-offs

Our evaluation showed that using speculation pays off in terms of performance. However, speculation potentially wastes energy, requires complex hardware, and has implications in system design, because some code (such as I/O and parts of an operating system) can't 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 (such as the edge of a quantum).

Supporting debugging instrumentation

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 original execution's interleaving. To accomplish this, implementations must support a mechanism that lets a compiler 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 operating system and I/O

A DMP system hides most sources of nondeterminism in today's systems, allowing many multithreaded programs to run deterministically. Slight modifications to existing operating system interfaces can handle some remaining sources of nondeterminism. For example, parallel programs that use the operating system to communicate between threads must do so deterministically, either by executing operating system code deterministically or detecting communication and synchronization via the kernel and providing it within the application itself. Our simulations use the latter approach. Another fixable source of nondeterminism is operating system APIs that allow unnecessarily nondeterministic outcomes. For example, the return value of a `read` system call might vary from run to run even if the file being read doesn't change. A solution for `read` is to always return the maximum amount of data requested until end-of-file.

Ultimately, however, the real world is simply nondeterministic. Programs interact with remote systems and users, which are all nondeterministic and can affect thread

interleaving. That might seem to imply that there's no hope for building a deterministic multiprocessor system, but that's not true. When multithreaded code synchronizes, that's an opportunity to be deterministic from that point forward because 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's much easier to write, debug, and deploy reliable multithreaded software.

Support for deployment

Deterministic systems shouldn't just be used for development, but for deployment as well, so that systems in the field will behave like systems used for testing. The reason is twofold. First, developers will be more confident that their programs will work correctly when deployed. Second, if the program does crash in the field, 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 (such as micro-op count, or a full cache-set for bounded TM-based implementations) can't be used to end quanta. 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.

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; besides yielding several interesting research questions, they abstract away several difficulties in writing, debugging, and deploying parallel code. MICRO

Acknowledgments

We thank Karin Strauss, Dan Grossman, Susan Eggers, Martha Kim, Andrew Putnam, Tom Bergan, and Owen Anderson from the University of Washington for their feedback on this article. Finally, we thank Jim Larus and Shaz Qadeer from Microsoft Research for very helpful discussions.

Other members of our group involved in determinism-related projects are Owen Anderson, Tom Bergan, Nick Hunt, Steve Gribble, and Dan Grossman.

References

1. D.F. Bacon and S.C. Goldstein, ''Hardware-Assisted Replay of Multiprocessor Programs,'' *Proc. 1991 ACM/ONR Workshop Parallel and Distributed Debugging,* ACM Press, 1991, pp. 194-206.
2. P. Montesinos, L. Ceze, and J. Torrellas, ''DeLorean: Recording and Deterministically Replaying Shared-Memory Multiprocessor Execution Efficiently,'' *Proc. 35th Int'l Symp. Computer Architecture* (ISCA 08), IEEE CS Press, 2008, pp. 289-300.
3. S. Narayanasamy, C. Pereira, and B. Calder, ''Recording Shared Memory Dependencies Using Strata,'' *Proc. 12th Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 06), ACM Press, 2006, pp. 229-240.
4. M. Ronsee and K. De Bosschere, ''RecPlay: A Fully Integrated Practical Record/Replay System,'' *ACM Trans. Computer Systems,* vol. 17, no. 2, 1999, pp. 133-152.
5. M. Xu, R. Bodik, and M.D. Hill, ''A 'Flight Data Recorder' for Enabling Full-System Multiprocessor Deterministic Replay,'' *Proc. 30th Int'l Symp. Computer Architecture* (ISCA 03), IEEE CS Press, 2003, pp. 122-135.
6. T. Bergan et al., ''CoreDet: A Compiler and Runtime System for Deterministic Multithreaded Execution,'' to be presented at 15th Int'l Conf. Architectural Support for Programming Languages and Operating Systems, 2010, www.cs.washington.edu/homes/tbergan/papers/asplos10-coredet.pdf.
7. L. Hammond et al., ''Transactional Memory Coherence and Consistency,'' *Proc. 31st Int'l Symp. Computer Architecture* (ISCA 04), IEEE CS Press, 2004, pp. 102-114.
8. M. Herlihy and J.E.B. Moss, ''Transactional Memory: Architectural Support for Lock-Free Data Structures,'' *Proc. 20th Int'l Symp. Computer Architecture* (ISCA 93), IEEE CS Press, 1993, pp. 289-300.
9. J. Devietti et al., ''DMP: Deterministic Shared Memory Multiprocessing,'' *Proc. 14th Int'l*

Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS 09), ACM Press, 2009, pp. 85-96.
10. S. Gopal et al., ''Speculative Versioning Cache,'' *Proc. 4th Int'l Symp. High-Performance Computer Architecture* (HPCA 98), IEEE CS Press, 1998, pp. 195-205.
11. C.K. Luk et al., ''Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation,'' *Proc. 2005 ACM SIGPLAN Conf. Programming Language Design and Implementation,* ACM Press, 2005, pp. 190-200.
12. S. Woo et al., ''The SPLASH-2 Programs: Characterization and Methodological Considerations,'' *Proc. 22nd Int'l Symp. Computer Architecture* (ISCA 95), IEEE CS Press, 1995, pp. 24-36.
13. C. Bienia et al., ''The PARSEC Benchmark Suite: Characterization and Architectural Implications,'' *Proc. 17th Int'l Conf. Parallel Architectures and Compilation Techniques,* ACM Press, 2008, pp. 72-81.

Joseph Devietti is a PhD candidate in the Computer Science and Engineering Department at the University of Washington. His research interests include parallel-programming models and the intersection of architecture and programming languages. Devietti has an MS in computer science and engineering from the University of Washington. He is a member of the ACM.

Brandon Lucia is a PhD candidate in the Computer Science and Engineering Department at the University of Washington. His research interests include hardware support for dynamic analysis of concurrent programs, with a focus on debugging and increasing software reliability. Lucia has an MS in computer science and engineering from the University of Washington. He is a member of the ACM.

Luis Ceze is an assistant professor in the Computer Science and Engineering Department at the University of Washington. His research interests include computer architecture, programming languages, and compilers and operating systems to improve programmability and reliability of multiprocessors. He is a cofounder of PetraVM, a start-up company formed around the DMP technology. Ceze has a PhD in computer science from the University of Illinois at Urbana-Champaign. He is a member of the ACM and the IEEE.

Mark Oskin is an associate professor in the Computer Science and Engineering Department at the University of Washington. He is currently on leave at PetraVM, a start-up commercializing many of the ideas behind DMP. His research interests include parallel computing, communication networks, and silicon manufacturing. Oskin has a PhD in computer science from the University of California at Davis.

Direct questions and comments to Joseph Devietti, Computer Science & Engineering, Univ. of Washington, Box 352350, Seattle, WA 98195-2350; devietti@cs.washington.edu.