

# RCDC: A Relaxed Consistency Deterministic Computer

Joseph Devietti Jacob Nelson Tom Bergan Luis Ceze Dan Grossman

University of Washington, Department of Computer Science & Engineering

{devietti,nelson,tbergan,luisceze,djg}@cs.washington.edu

<http://sampa.cs.washington.edu>

## Abstract

*Providing deterministic execution significantly simplifies the debugging, testing, replication, and deployment of multithreaded programs. Recent work has developed deterministic multiprocessor architectures as well as compiler and runtime systems that enforce determinism in current hardware. Such work has incidentally imposed strong memory-ordering properties. Historically, memory ordering has been relaxed in favor of higher performance in shared memory multiprocessors and, interestingly, determinism exacerbates the cost of strong memory ordering. Consequently, we argue that relaxed memory ordering is vital to achieving faster deterministic execution.*

*This paper introduces RCDC, a deterministic multiprocessor architecture that takes advantage of relaxed memory orderings to provide high-performance deterministic execution with low hardware complexity. RCDC has two key innovations: a hybrid HW/SW approach to enforcing determinism; and a new deterministic execution strategy that leverages data-race-free-based memory models (e.g., the models for Java and C++) to improve performance and scalability without sacrificing determinism, even in the presence of races. In our hybrid HW/SW approach, the only hardware mechanisms required are software-controlled store buffering and support for precise instruction counting; we do not require speculation. A runtime system uses these mechanisms to enforce determinism for arbitrary programs.*

*We evaluate RCDC using PARSEC benchmarks and show that relaxing memory ordering leads to performance and scalability close to nondeterministic execution without requiring any form of speculation. We also compare our new execution strategy to one based on TSO (total-store-ordering) and show that some applications benefit significantly from the extra relaxation. We also evaluate a software-only implementation of our new deterministic execution strategy.*

**Categories and Subject Descriptors** D.1.3 [Programming Languages]: Concurrent Programming—Parallel Programming; C.1.4 [Processor Architectures]: Parallel Architectures; D.3.4 [Programming Languages]: Processors—Run-time environments

**General Terms** Reliability, Design, Performance

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ASPLOS'11, March 5–11, 2011, Newport Beach, California, USA.  
Copyright © 2011 ACM 978-1-4503-0266-1/11/03...\$10.00

## 1. Introduction

Even if given the same exact input, a multithreaded program may not produce the same output. Nondeterminism in current multiprocessor systems severely complicates debugging, testing, replication, and deployment of multithreaded programs. Once a bug is found, it is hard to reproduce its behavior during debugging. When a multithreaded program is tested, there are no guarantees that it will behave the same way after deployment. Moreover, nondeterminism complicates fault-tolerant systems, since multithreaded replicas may have divergent behavior.

Deterministic multiprocessing promises to eliminate nondeterminism in the execution of multithreaded programs, effectively eliminating the problems described above. As such, determinism provides a large step toward improved programmability of multi-core systems.

Recent research has begun to explore ways to execute multithreaded programs deterministically. Kendo [26] is a software-only approach that provides determinism for race-free programs, but offers no guarantee for programs with data races. The essential idea is to allow a thread to complete a synchronization operation only when all other threads have completed more total instructions. DMP [14] proposed two alternate strategies using novel hardware: the first uses data-ownership tracking and periodic barriers to guarantee that all data movement between threads happens deterministically; the second uses support for transactional memory (TM) to speculate that regions of execution do not have any inter-thread communication, rolling-back and re-executing if inter-thread communication does happen. Neither strategy assumes race freedom, thus they both provide determinism for arbitrary programs. However, both strategies produce only sequentially consistent executions. This property likely affects performance and scalability of the first strategy, and while the TM-based strategy has high performance, it entails always-on speculation with resultant complexity and energy costs. Subsequently, CoreDet [4] argued that this TM-based approach is a poor fit for software TM, and thus needs hardware support. CoreDet improved scalability by proposing a deterministic execution strategy and a software-only implementation that uses deterministic store buffers to weaken the memory consistency model from sequential consistency to total-store-order (TSO) and consequently improves scalability without using speculation. While CoreDet demonstrated reasonable scalability, the high cost of software-based store buffers led to significant overheads.

Relaxing memory ordering has proven instrumental in improving performance and scalability in conventional nondeterministic shared memory multiprocessor architectures [1, 17]. While speculation alleviates some of the costs of strong ordering [11, 18, 32, 34] in complex architectures, we still relax memory ordering to allow compiler optimizations and to simplify hardware. Interestingly, strong memory ordering has a much higher cost in deterministic multiprocessing than in nondeterministic multiprocessing. There-

fore we argue that, in deterministic multiprocessors, it is even more important to give up strong memory ordering in favor of higher performance and lower complexity.

This paper proposes RCDC, a relaxed consistency deterministic multiprocessor computer system. RCDC improves prior work in two ways:

- RCDC implements a *new deterministic execution algorithm*, called DMP-HB (for “happens-before”), which relaxes memory consistency even further than TSO while still supporting data-race-free-based memory models (e.g., those of Java and C++). This improves performance and scalability by requiring fewer costly fences, which leads to less serialization. DMP-HB does not employ speculation and does not sacrifice determinism in the presence of races.
- RCDC uses a *lower complexity hybrid hardware/software implementation* in which the hardware provides only two simple mechanisms, software-controlled store buffering and instruction counting, leaving the rest of the implementation to software. Implementing store buffering in hardware has the pleasant side effect of reducing the effects of false sharing. RCDC can be implemented on a commodity multiprocessor architecture and does not interfere with software (e.g., the OS) that does not choose to use it.

The rest of this paper is organized as follows. We first discuss prior deterministic execution algorithms and how they relate to memory ordering, and also explain our new algorithm for deterministic execution (Section 2). We then provide an overview of how RCDC works, highlighting the responsibilities of hardware and software (Section 3). We follow with implementation details of RCDC’s hardware components, protocols, and software runtime system (Section 4). We then discuss system issues such as page swapping and context switches (Section 5). We evaluate RCDC in comparison with prior approaches and also include an evaluation of a software-only implementation of DMP-HB (Section 6). We end with related work and closing remarks (Sections 7 and 8).

## 2. Relaxed-Consistency Deterministic Execution

This section presents our new deterministic execution algorithm, DMP-HB. For expository purposes, we first describe two related deterministic execution algorithms from prior work (DMP-SERIAL [14] and DMP-TSO [4]<sup>1</sup>).

### 2.1 DMP-SERIAL

The simplest way to run a multithreaded program deterministically is to serialize its execution in a deterministic way. For example, we can schedule threads in a round-robin fashion so that execution is serial. Each thread is scheduled for one finite *logical* time slice, or *quantum*; a *round* consists of all threads executing one quantum each. To ensure determinism it suffices to ensure that the length of each quantum and the scheduling order are both deterministic.

### 2.2 DMP-TSO: Store Buffering

One way to recover parallelism is to isolate threads using store buffers. In this approach, each round is divided into three modes, a *parallel mode*, a *commit mode*, and a *serial mode*. During parallel mode, all stores are buffered in a thread-local store buffer, giving each thread a private view of shared memory. After parallel mode, all threads enter a commit mode in which the local store buffers are published to the global memory space. This commit happens

deterministically. The effect is a serial commit order, but the implementation uses parallelism to avoid a sequential bottleneck. After commit mode is a short serial mode in which threads execute in a deterministic serial order and operate on shared memory directly. Serial mode is used to execute atomic synchronization operations, as described below.

Figure 1a illustrates one round of execution in DMP-TSO. Each thread executes one quantum per round, where, as in DMP-SERIAL, a quantum is some deterministic number of instructions. DMP-TSO is deterministic due to four properties: (1) quantum lengths are deterministic; (2) threads are isolated in parallel mode, preventing nondeterministic interference from other threads; (3) commit mode ensures that writes to shared memory happen in a deterministic order; and (4) serial mode ensures that atomic synchronization happens in a deterministic order. Note that the deterministic guarantee offered by DMP-TSO does not depend on a race-free assumption — data races are resolved deterministically as a result of the isolation provided by parallel mode, combined with the deterministic order on writes provided by commit mode.

Notice that execution under DMP-TSO is not sequentially consistent; stores are not globally visible until commit mode, effectively reordering them after loads in the same quantum. This reveals the need for serial mode — without serial mode, synchronization would not happen atomically. Further, it reveals the need to define the semantics of a memory fence. In DMP-TSO, thread  $T$  ends its parallel mode when it reaches a memory fence. This flushes  $T$ ’s local store buffer, implementing the semantics of a full memory fence. Because DMP-TSO does not distinguish between different types of memory fences, it implements the total-store-order (TSO) memory model.

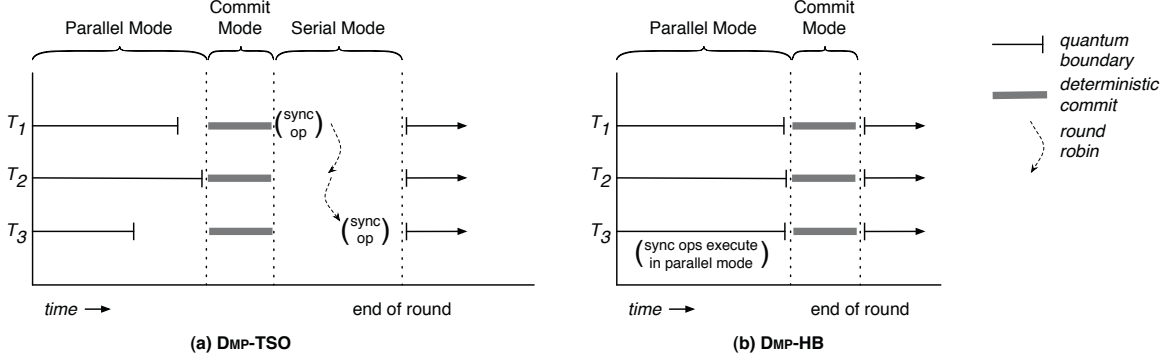
DMP-TSO achieves high performance when serial mode is empty and parallel mode is *balanced*, meaning that all quanta in a round execute in the same amount of real time. Serial mode is empty when synchronization is rare; prior work [4] has shown how to use instruction counting to achieve balanced parallel modes when serial mode is empty. When synchronization does happen, it forces DMP-TSO into serial mode, whereby every synchronization operation causes global coordination. Synchronization not only causes serialization but also *imbalance* in parallel mode, which results in additional lost parallelism due to excess waiting. When synchronization is frequent, the effects of serialization and imbalance dominate and performance suffers.

### 2.3 DMP-HB: Leveraging Data-Race-Free Memory Models

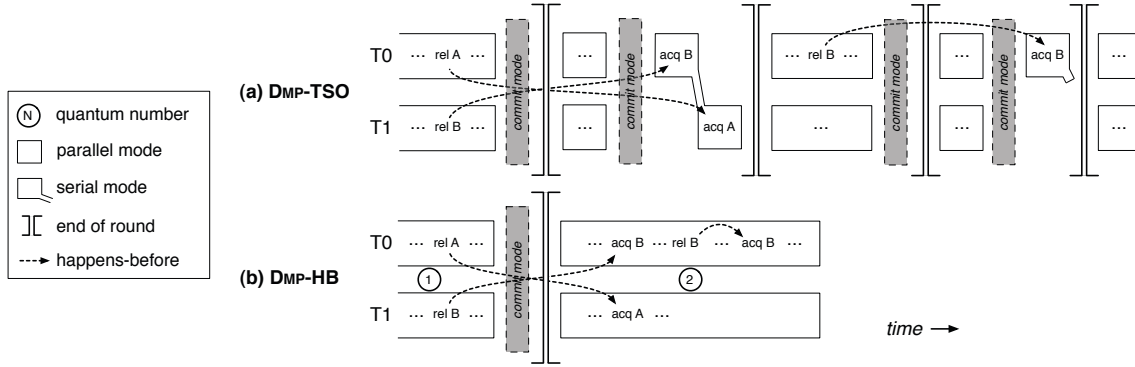
DMP-HB addresses the major weakness of DMP-TSO: synchronization. Like DMP-TSO, DMP-HB uses deterministic store buffers and divides execution into quantum rounds with parallel modes and commit modes. However, DMP-HB introduces a new approach to deterministic synchronization that improves on DMP-TSO in two respects. First, DMP-HB implements a data-race-free [1] (DRF) relaxed memory model based on the *happens-before* relation between threads (hence “DMP-HB”). This model requires fewer memory fences than TSO, which makes parallel mode less likely to end early, thus increasing parallelism. However, weakening consistency alone does not remove all impediments to scalability, as DMP-TSO requires that synchronization execute in a globally serialized fashion. To remove this further bottleneck, DMP-HB eliminates the need for an explicit serial mode by using the Kendo algorithm [26] to execute synchronization directly in parallel mode, while still providing determinism even for programs with races. Overall, these optimizations let DMP-HB execute with less serialization and less imbalance than DMP-TSO, leading to improved parallelism as illustrated in Figure 1b.

The key observation of DMP-HB is that language-level memory models have weaker consistency guarantees than TSO. Specif-

<sup>1</sup>In [4], DMP-TSO was named DMP-B; this paper uses the name DMP-TSO for improved contrast with DMP-HB.



**Figure 1.** Timeline of a quantum round in DMP-TSO and DMP-HB, showing the division of each round into parallel, serial, and commit modes. DMP-HB improves DMP-TSO by allowing synchronization to happen in parallel mode, eliminating the need for serial mode.



**Figure 2.** A comparison of execution under DMP-HB with execution under DMP-TSO, showing how DMP-HB extracts more parallelism from programs with frequent synchronization.

ically, Java [22] and C++ [10] define consistency models based on the data-race-free model [1]. From the programmer’s perspective, it does not matter that the execution layer (*e.g.*, the hardware) provides TSO when other layers of the system (*e.g.*, the compiler) guarantee only DRF. Further, the need to precisely control memory visibility causes nondeterministic processor-local fences to become global operations in deterministic systems like DMP-TSO, which suggests that strong memory ordering has a much higher cost in deterministic systems than in nondeterministic systems. Both these observations imply that deterministic systems should relax consistency as much as possible. As DRF-based models are specified by high-level languages, they represent the limit to which memory ordering can be relaxed.

### 2.3.1 Synchronization in DMP-HB

As DMP-TSO is a deterministic version of a TSO consistency model, DMP-HB is a deterministic version of a DRF consistency model. DMP-HB differs from DMP-TSO in its approach to synchronization. The rest of this section presents the basic ideas of DMP-HB. We describe details of the DMP-HB synchronization algorithm along with our synchronization library in Section 4.4.

Consider mutex locks in a language with a DRF-like model, such as Java or C++. In these languages, the visibility of stores is guaranteed only along happens-before edges, which can arise from program order between consecutive operations in a thread or from synchronization operations across threads. When thread  $T$  acquires lock  $L$ , this creates a happens-before edge  $E$  from the previous releaser of  $L$  to  $T$ . The DRF model guarantees that from this point forward,  $T$  will see stores that transitively happen-before its acquire of  $L$ . Other stores need not be visible. Therefore, in DRF models,  $T$  needs a memory fence after acquiring lock  $L$  *only* when happens-

before edge  $E$  is not redundant. When  $E$  is redundant, the fence can be elided.

DMP-HB exploits two happens-before redundancies: (1) thread-local edges, and (2) cross-quantum edges. First consider thread-local redundancies: if  $T$  was the previous releaser of  $L$ , then lock  $L$  has not been handed off to another thread, and we say that happens-before edge  $E$  is *local* to thread  $T$ . A fence is not needed in this case because edge  $E$  is redundant with program order. Prior work has had this same insight but in the context of nondeterministic systems, and furthermore has shown that lock locality is very common in Java programs [30, 33].

Cross-quantum redundancies are more interesting. They follow from the observation that all quanta in round  $N$  are connected to all quanta in round  $N + 1$  by implicit happens-before edges. These implicit edges arise from the bulk-synchronous style of execution used by DMP-HB, illustrated in Figure 1b. The important result is that an explicit fence is not necessary when synchronization is separated by a quantum boundary. Thus, by matching quantum length with the frequency of synchronization, DMP-HB can eliminate many unnecessary fences, increasing performance and scalability.

Figure 2b demonstrates both the above redundancies. An example of a redundant cross-quantum edge is shown when thread  $T_0$  acquires lock  $B$  in quantum 2: this creates a happens-before edge with the release of lock  $B$  by thread  $T_1$  in quantum 1. Because this edge crosses a quantum boundary (*i.e.*, it crosses a *commit mode*),  $T_0$  does not need to execute an explicit fence when acquiring lock  $B$ . In contrast, note that under DMP-TSO,  $T_0$  must execute a fence, *i.e.*, end its quantum, before acquiring lock  $B$ .

An example of a thread-local redundancy is also shown in quantum 2, where  $T_0$  reacquires lock  $B$ . As  $T_0$ ’s updates are automatically visible to itself, there is no need for a fence. The

extra serialization necessary to enforce the stronger TSO memory model is illustrated in Figure 2a.

Further, Figure 2b shows that DMP-HB does not require a serial mode, in contrast to DMP-TSO, which executes all lock acquires in serial mode. Even with the weaker DRF memory model, serializing all synchronization eliminates the ability to exploit thread-local redundant fences. Recall that DMP-TSO uses a serial mode to guarantee both atomicity and a deterministic order of synchronization. For correctness and determinism, DMP-HB must make these same two guarantees. Our solution is to use the Kendo algorithm [26] to impose a deterministic total order on all synchronization within a single quantum round. This algorithm allows synchronization to operate directly on the global memory space, bypassing the store buffer so the operation happens atomically. We describe this algorithm along with our synchronization library in Section 4.4.

### 2.3.2 Language Memory Models

Even though DRF does not specify the semantics of races, DMP-HB’s deterministic guarantees hold even for programs with data races. DMP-HB’s DRF memory model naturally matches the C++ memory model; however, the Java memory model specifies some behavior for data races, *e.g.*, to prevent “out-of-thin-air” values. DMP-HB does not itself introduce any potential “out-of-thin-air” values because it does not employ any form of speculation. A Java compiler must (still) ensure that its optimizations do not violate the Java memory model and also must ensure that proper synchronization and fences are inserted. Therefore, compiling Java code for DMP-HB’s memory model is no more complex than compiling for a weak ordering system.

## 3. RCDC System Overview

RCDC provides an efficient implementation of DMP-HB through a combination of hardware and software mechanisms, as summarized in Figure 3. The four main components of RCDC are (1) a precise instruction-count mechanism to divide each thread’s instruction stream into balanced quanta efficiently, (2) a store-buffer mechanism that allows threads to execute in isolation from other threads, (3) a deterministic commit mechanism that concludes each quantum round, and (4) a custom synchronization library that implements a pthreads interface while enforcing DMP-HB’s memory-consistency model. These components are implemented as a combination of hardware and software designed for maximal flexibility and minimal hardware complexity.

**Quantum formation** is an ideal use case for hardware, as counting instructions involves nearly zero overhead in hardware but causes substantial slowdown in software. The hardware instruction-counting mechanism simply counts instructions as they retire, triggering a user-level `QuantumReached` trap when a pre-defined total is reached. This trap is responsible for actually starting the commit process that makes buffered data visible.

Our synchronization library requires the ability to disable and enable instruction counting for the local processor, and also to read the current instruction counts of remote processors. For this purpose the instruction-count mechanism can be controlled and queried via the `StopInsnCount`, `StartInsnCount`, and `ReadInsnCount` instructions.

Deterministic quantum formation beyond simply counting instructions is also possible. Section 4.1 describes a more advanced strategy that uses opcodes and store buffer hit/miss information to construct quanta with better balance.

The **store buffer** mechanism is also a natural fit for hardware, where processor-private caches can isolate an executing thread from other threads in the system (like in hardware transactional memory, but without an abort mechanism) with additional bits of cache line state. We rely on simple compiler modifications or bi-

nary rewriting to replace existing store instructions with our new `BufferedStore` instruction. With more sophisticated analysis, ordinary non-buffered store instructions can, without any loss of determinism, replace buffered stores to locations that are provably thread-private. This increases the effective capacity of the store buffer without additional hardware resources.

The **deterministic commit** mechanism is triggered by software, via a new `Commit` instruction. The actual commit process is implemented in hardware, as described in Section 4.3. The commit process is invoked by software in response to `QuantumReached` and `BufferFull` traps, as well as to enforce the memory consistency requirements of DMP-HB. Hardware triggers a `BufferFull` trap immediately when a store buffer overflows. Note that our cache replacement policy, described in Section 4.2, is designed to ensure that store buffer overflows happen deterministically.

Finally, our custom **synchronization library** acts as a drop-in replacement for pthreads. It uses the instructions described above to enforce DMP-HB’s consistency model. Because the decision of *when* to commit is left to software, our synchronization library can easily be modified to implement other consistency models, *e.g.*, DMP-TSO.

## 4. Implementation

We now discuss implementation details for the major hardware and software components described in Section 3.

### 4.1 Quantum Formation

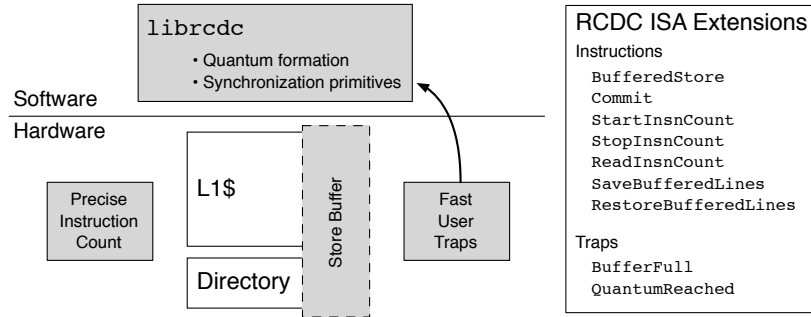
For quantum formation, the **hardware** does instruction counting and sets a trap after a software-defined quantum budget is exhausted. This mechanism is initialized by system **software** when a process is created. This code registers a user-level trap that is invoked whenever the quantum size is reached, establishes the size of quanta, and executes the `StartInsnCount` instruction.

Thus far, we have discussed quantum formation in terms of counting instructions. However, it is possible to provide better quantum balance by giving instructions non-uniform weights, *e.g.*, based on opcode. One useful optimization leverages the deterministic contents of the store buffer. Memory accesses deterministically either hit in the store buffer (*i.e.*, hit to a written cache line) or miss the store buffer (which means either hitting to a non-written cache line or missing to the next-level cache or beyond). Knowing whether the memory operation is a load or a store lets us, in many cases, accurately assess the latency of that operation deterministically. Loads that miss the store buffer are often hits to a clean line, so we assign them a low weight. But stores that miss in the store buffer tend to be cache misses, and thus have high latency. Assigning higher weights to such stores results in better quantum balance since the weight assigned to each instruction better approximates its actual latency. Finally, we add to the sum of instruction weights as instructions retire, allowing access to hit/miss information, and also avoiding any issues with wrong-path instructions.

### 4.2 Buffering

RCDC’s **hardware** provides support for buffering data, while **software** (*i.e.*, the compiler) controls what data is buffered via the `BufferedStore` instruction. This section details how buffering support is implemented as an extension to the cache hardware.

Cache-based data buffering imposes a few system requirements: (1) buffered lines cannot be provided to remote requests; (2) the commit protocol, which makes buffered data available to all processors, needs to be deterministic; and (3) the system needs to support context switches. RCDC provides this functionality on top of a conventional directory-based MOESI cache coherence protocol and implements buffering in private caches, while still naturally



**Figure 3.** RCDC system overview, showing the division of responsibility between hardware and software. The shaded boxes show RCDC’s additions.

supporting higher-level shared caches. For simplicity of explanation, we consider only a single private L1 cache per processor in the discussion below.

**Cache Extensions for Store Buffering** Each L1 cache line is extended with a *write-mask*, which has as many bits as bytes in the cache line. When a `BufferedStore` is executed, the corresponding write-mask bits are set. Consequently, lines with non-null write-masks contain buffered data.

To ensure that store buffer capacity is exhausted deterministically, we modify the cache eviction policy to always preferentially evict unwritten cache lines from a set. This ensures determinism while maximizing the amount of progress a processor can make before running out of store buffer capacity. When all lines of a cache set are buffered and an eviction needs to happen, RCDC triggers a `BufferFull` trap and the runtime system ends the quantum.

Non-buffered stores to cache lines in the non-buffered state proceed normally, following the conventional MOESI protocol. Non-buffered stores to cache lines in the buffered state are treated like `BufferedStores`. Note that buffering data from a non-buffered store is valid with respect to the instruction semantics as it is always correct to buffer private data — it just will not bring any benefit. If a non-buffered store necessarily cannot be buffered because of program semantics, then software needs to guarantee that this does not happen (e.g., using careful memory layout).

**Coherence Operations** We augment the transitions in a conventional MOESI protocol to handle our new `BufferedStore` instruction. This requires three changes to a conventional MOESI protocol. First, if an L1 cache receives a request for a line whose write-mask is non-null, the request is *nacked*. The requester then goes to a shared higher-level cache (or memory) to fulfill its request. This is necessary to guarantee that buffered data is never provided to remote requests. Second, a line must be in the Shared state before it can be written by a `BufferedStore` instruction. Finally, our commit protocol (Section 4.3) moves lines to the Owned state after they have been published. As a consequence of these last two changes, moving a line to the Shared state to satisfy a `BufferedStore` may require a write-back operation (e.g., because that line may have been buffered in the previous round).

Interestingly, isolating each thread’s updates into separate store buffers also yields a solution to false sharing, by allowing threads to perform updates to the same cache line within a quantum round without any serialization via the coherence protocol. The line becomes temporarily incoherent, but the updates are merged deterministically at the end of the round. If threads’ updates are in conflict (i.e., two threads update the same bytes), there is a data race in the user program — data-race free programs can never observe this relaxation of coherence.

**Context Switches** The kernel can context switch away from and back to a thread at any time, even during parallel mode, as long as

it invokes the `SaveBufferedLines` and `RestoreBufferedLines` instructions to save and restore a thread’s current store buffer.

These instructions make use of a per-thread, in-memory data structure called the Buffered Data Table (BDT), which contains the saved store buffer contents for a given thread. A BDT has a row for each cache line in a processor’s store buffer, with one column for the line’s data, another for the line’s write-mask, and a third for a “next” pointer whose use is described below. A row in the BDT is considered valid if its write-mask is non-null. The `SaveBufferedLines` instruction simply flushes all buffered lines from the cache to the BDT. As it does so, it clears the write-masks of all buffered lines in the L1 cache and transitions them to the Invalid state, making them available for the next thread to be switched in. The `RestoreBufferedLines` instruction iterates over all buffered lines in a given BDT, restoring them into the L1 cache. After a line is restored from the BDT to the L1 cache, its write-mask is cleared in the BDT to signify that the line is no longer saved in-memory.

These two instructions additionally maintain a separate, per-process table called the Buffered Address Map (BAM). The BAM is a table of pointers mapping each line address to a linked list of BDT entries storing the in-memory versions of that line. The pointer in each BDT entry points to the next thread’s BDT entry in the list. Each BDT entry represents the saved state of one buffered version of the given cache line. By walking the list, the BAM table can be used to enumerate all in-memory versions of a buffered line. BAMs are used during the commit process as described in the following section.

We highlight that these instructions can be expensive, not only on their own, but also because of the extra work they impose on the commit process. In Section 5 we describe a few kernel scheduling optimizations that make these instructions infrequent.

### 4.3 Committing Buffered Data

In DMP-HB, the transition to commit mode is controlled by **software**, which uses the `Commit` instruction to initiate the actual commit process in **hardware**. The RCDC software runtime executes the following pseudocode for each thread when it reaches its quantum boundary, e.g., when its quantum budget has been exhausted:

```

1 end_quantum() {
2   global_barrier()
3   Commit
4   global_barrier()
5 }
```

The first barrier represents the transition from parallel mode to commit mode; the `Commit` instruction represents commit mode; and the second barrier represents the transition back to parallel mode to start the next quantum (see Figure 1b). The first barrier ensures that all threads are ready to commit, while the second

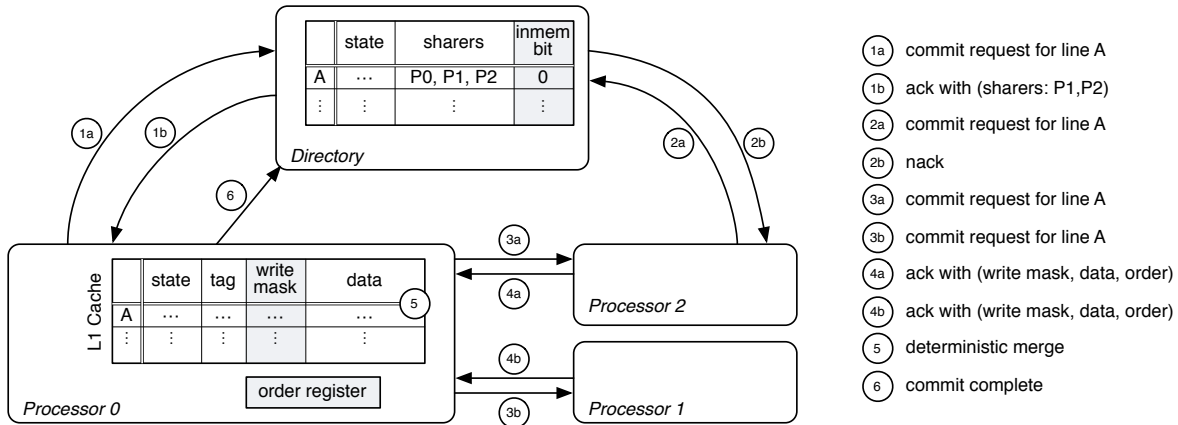


Figure 4. RCDC commit process when all application threads are scheduled. Shaded areas are RCDC additions.

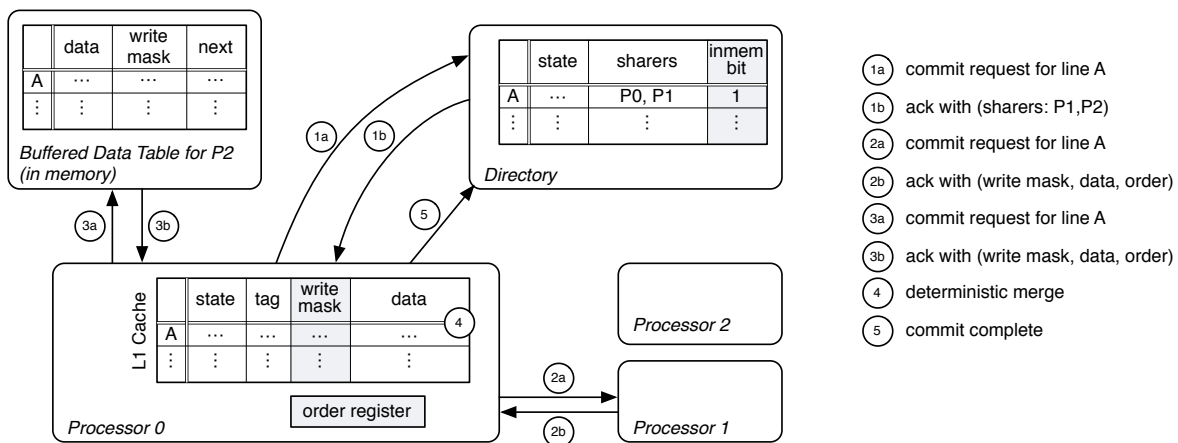


Figure 5. RCDC commit process when an application thread is switched out.

barrier ensures that all threads have finished publishing the contents of their store buffers.

The goal of the commit process is to merge buffered data deterministically and publish it globally. If a line has no buffered data in any cache, commit has no effect on that line. When a line has buffered data in one or more caches, the commit process deterministically merges all buffered data and then publishes this data to the rest of the system by moving the merged line to the Owned state.

A processor executes the commit instruction by iterating over all lines in its cache that have buffered data, *i.e.*, those lines with non-null write-masks. For each of those lines, the processor executes the commit protocol. The commit protocol coordinates with the directory and with other processors, collects all buffered versions of a line, and then deterministically merges them. Once the commit protocol has been executed for all of the processor's buffered lines, the `Commit` instruction retires. At this point, the processor's entire store buffer has been globally published.

The commit protocol needs to handle two cases: committing when all buffered versions of a line are in-cache; and committing when at least one buffered version of a line is out-of-cache (*i.e.*, because the thread was context-switched out). We describe both cases in detail below.

**In-cache Commit (all threads running)** The processor issues a commit message for the given line to the directory; the directory replies with an acknowledgment for commit and a list of sharers for that line. The processor sends a commit message to each sharer. Upon receipt of a commit message, each sharer returns a reply

indicating if it has the line, and if so, it includes the write-mask and the data for the requested line, as well as its deterministic *order id*; it then clears its write-mask and moves the line to Invalid state. When the committing processor receives a reply, it merges the other processor's data into its own line. Once the processor has collected replies from all sharers, it clears the line's write-mask and moves the line to the Owned state, making the line visible to all processors.

The merge algorithm takes lines from two processors,  $P_0$  and  $P_1$ , and computes the result of  $P_0$ 's writes happening before  $P_1$ , where  $P_0$  has the smaller order id. This algorithm is straightforward and has been described by prior work [4].

Note that if the directory nacks the request for commit of a given line, this implies that some other processor has already started the commit for that line; the requester then waits until it receives a commit request for the line from another processor. Also note that while the commit process can actually happen in any order, the final state is guaranteed to be deterministic because the merge process is deterministic. Moreover, when the sharers list includes only the committing processor, no merge is necessary; the processor simply clears the line's write-mask and moves the line to the Owned state.

**Example.** Figure 4 illustrates how RCDC deals with multiple caches trying to commit to the same line A. Processors  $P_0$ ,  $P_1$ , and  $P_2$  all have buffered copies of line A. First,  $P_0$  and  $P_2$  send concurrent commit requests for line A to the directory (1a, 2a).  $P_0$ 's message arrives first, and the directory responds to  $P_0$  with an acknowledgment message (1b), including the list of sharers, allowing  $P_0$  to proceed with committing line A. Since commit has

```

1  sync_acquire(o: SyncObject) {
2      if (o.quantum < curr_quantum ||
3          o.releaser == self) {
4          return // fence not necessary
5      }
6      end_quantum()
7  }
8
9  deterministic_lock(l: Lock) {
10     StopInsnCount
11     while (true) {
12         wait_for_turn()
13         if (CAS(l.locked, 0, 1)) {
14             sync_acquire(l)
15             StartInsnCount
16             return
17         }
18         end_quantum()
19     }
20 }

21 sync_release(o: SyncObject) {
22     o.quantum = curr_quantum
23     o.releaser = self
24 }
25
26
27
28
29 deterministic_unlock(l: Lock) {
30     StopInsnCount
31     wait_for_turn()
32     sync_release(l)
33     l.locked = 0
34     StartInsnCount
35 }

```

Figure 6. Deterministic Locking for DMP-HB

started for line *A*, the directory responds to *P2* with a negative acknowledgment (2b) and *P2* waits for a commit request (which it is bound to get since it is guaranteed that another processor is committing *A*). *P0* continues by sending commit messages to all sharers of *A* (3a, 3b). *P1* and *P2* respond with a message containing their data, write mask, and deterministic order (4a, 4b) and then invalidate their copy of the line. Upon receiving the acks, *P0* deterministically merges the data with its own (5) and notifies the directory that line *A* has been committed (6).

**Out-of-cache Commit (at least one thread is switched out)** We now describe the more general case where at least one sharer has been switched out. RCDC supports this case with a simple extension to the directory: each directory entry, in addition to the sharers, also includes a single bit called the *in-memory bit*, which indicates if the line has data in an in-memory Buffered Data Table. This in-memory bit is set by the `SaveBufferedLines` instruction.

When a committing processor issues a commit message for a line to the directory, the directory replies with an acknowledgment and a list of sharers as before, and the processor communicates with the sharers as before. However, the directory also replies with the state of the in-memory bit. If the in-memory bit is set, the committing processor walks rows in Buffered Data Tables via the Buffered Address Map table to enumerate all in-memory versions of the line being committed. The processor merges these versions into its own line using the same algorithm as before, and then sends a commit-complete message to the directory. At this point the directory can clear the in-memory bit.

Note that commit still proceeds correctly even if the only thread that has a given line buffered is switched out, since that thread will invoke the `Commit` instruction when it is eventually switched in. (The barrier on line 4 of `end_quantum` ensures this.) Also, note that the directory serves as a serialization point for the cache line operations performed by the `Commit`, `SaveBufferedLines`, and `RestoreBufferedLines` instructions; this prevents races between the commit process and context switches, making it safe for the kernel to switch out a thread at any time without sacrificing determinism (note especially that a thread can be safely switched out between lines 2 and 3 of `end_quantum`).

*Example.* Figure 5 illustrates how RCDC commits line *A* when a thread that has buffered line *A* has been switched out. *P0*, *P1*,

and *P2* all had buffered copies of line *A*. The thread on *P2* is ready for commit but was switched out just before the commit process starts. The buffered data is saved in *P2*'s Buffered Data Table in memory, and the in-memory bit set in the directory. *P0* sends a commit message (1a) to the directory for line *A*. The directory replies with an acknowledgment (1b), including the list of sharers and the in-memory bit. *P0* then sends a commit message for line *A* to *P1* (2a), which replies with an acknowledgment (2b) before invalidating the copy of the line. At the same time, *P0* accesses the Buffered Address Map to enumerate the list of in-memory lines, and notices that *P2*'s BDT contains a copy of line *A* (3a). *P2*'s saved line is found and returned to *P0* (3b), and then removed from the Buffered Data Table. Next, *P0* merges both received versions of line *A* with its own version (4) and clears the write mask. Finally, *P0* notifies the directory that the commit for line *A* is complete (5), and the directory resets line *A*'s in-memory bit.

#### 4.4 Synchronization Library

Our synchronization library is implemented using two basic building blocks: *conditional memory fences* and *deterministic serialization*. Conditional memory fences enforce DMP-HB's memory model. A mechanism for deterministic serialization based on the Kendo algorithm [26] is used to execute synchronization during parallel mode of DMP-HB.

Figure 6 shows our implementation of deterministic mutex locks. The `sync_acquire` and `sync_release` functions represent conditional memory fences, while `wait_for_turn` represents deterministic serialization. Other synchronization objects such as barriers, condition variables, and even lock-free data structures can be built from these same building blocks. For brevity we describe only a lock implementation in this paper.

**Conditional Memory Fences** We use the functions `sync_acquire` and `sync_release` to implement deterministic `lock` and `unlock` just as traditional nondeterministic implementations of `lock` and `unlock` use `acquire` and `release` fences [17]. The key difference is the conditional on lines 2-3 of `sync_acquire`: when this conditional is true, the release-to-acquire happens-before edge is redundant and a fence can be elided. When this conditional is false, a fence is necessary: `end_quantum` is invoked, which executes the `Commit` instruction. This conditional implements the observation

noted earlier in Section 2.3.1: a fence is not necessary when the happens-before edge is local to a thread (line 3) or crosses a quantum boundary (line 2). Note also that when lines 2-5 are removed, `sync_acquire` is a full fence, and so the remaining algorithm implements a consistency model equivalent to TSO.

**Deterministic Serialization** We use the Kendo algorithm [26] to serialize synchronization deterministically. The basic idea is as follows: before performing synchronization, thread  $T$  must wait for its *turn*, meaning it must wait until it has the global minimum instruction count (where ties are broken by thread ID). While waiting for its turn,  $T$  must disable instruction counting by invoking `StopInsnCount`; this ensures deterministic instruction counting since  $T$  may have to wait a nondeterministic amount of time before its turn arrives. After synchronization is complete,  $T$  invokes `StartInsnCount`. The `wait_for_turn` function can be implemented by polling other threads' instruction counts via the `ReadInsnCount` instruction.<sup>2</sup>

Note that lines 13-14 and lines 32-33 execute atomically: `wait_for_turn` designates the beginning of an atomic region that is ended by `StartInsnCount`.<sup>3</sup> It is within these regions that the lock object is updated. For these updates to appear atomically, they must apply directly to the global memory space; *i.e.*, all reads and updates of lock objects must bypass the store buffer. To ensure that lock objects are never buffered, lock objects can never exist on the same cache line as ordinary data; this introduces a *partition* of shared memory into lock objects and ordinary data.

## 5. System Issues

**Support for nondeterministic execution** The use of store buffers is a software choice. Therefore, programs can choose to execute nondeterministically. Kernel code, for example, would not need to be executed deterministically.

One caveat is that our eviction policy risks monopolizing the cache. Recall that buffered lines are pinned in the cache; if a cache set fills with buffered lines, it cannot be reused until the store buffer has committed. This can accidentally prevent important systems code (*e.g.*, context switch code) from running. We have two solutions. The first is to reserve a small victim buffer for non-buffered cache lines; and the second is to reserve just  $N - 1$  lines of a set for buffered data, where the cache uses  $N$ -way sets.

**Processes** In RCDC, each process is by default its own determinism domain; in other words, threads within a process behave deterministically with respect to each other. Deterministic processes can run alongside nondeterministic processes. Moreover, if multiple processes share memory pages, the processes can be aggregated into a single determinism domain, much like the deterministic process group abstraction in dOS [5]. As long as different determinism domains do not share memory pages, the boundary of determinism domains can be defined completely by software without any extra hardware support.

**Context Switches** To maintain determinism, RCDC requires that a thread's current instruction count and the contents of its store buffer be saved and restored across context switches. To reduce the amount of state that must be saved and restored, the OS kernel can be modified in two ways, described below.

First, the kernel can be modified to context switch away from a deterministic thread only at a quantum boundary, *i.e.*, just after

<sup>2</sup>Alternatively, we could implement `wait_for_turn` via interprocessor interrupts rather than polling.

<sup>3</sup>Atomicity for the lock release is necessary to guarantee that concurrent releases of the same lock (*e.g.*, due to programmer error) still result in a deterministic outcome.

line 4 of `end_quantum` (Section 4.3). This eliminates the need to save and restore the contents of store buffers, since store buffers are always empty at a quantum boundary.

Additionally, if there are  $N$  CPUs but more than  $N$  threads in a given determinism domain, the kernel can schedule threads in groups of  $N$  per quanta, much like gang scheduling [27]. This considerably reduces the need to save and restore the contents of store buffers. It also can improve quantum balance, by eliminating the underutilization that occurs when  $N + 1$  threads must be scheduled per round, yet there are only  $N$  processors available.

**Paging** It is important to make sure that none of the pages that have buffered data are paged out. The simplest way to provide this guarantee is to restrict paging so it happens only at the end of commit mode. In addition, the runtime system can provide the kernel with a list of pages that are provably unshared; these can be paged out at any time.

**Memory Errors** As discussed in Section 4.4, `librcdc`'s lock objects must be partitioned from ordinary data. If this partition is broken by some memory operation, *e.g.*, due to a memory error in a type unsafe language like C++, then that memory operation is a potential source of nondeterminism. For example, an errant read that happens to address a lock object will return a nondeterministic value, since that read can race with some other thread performing a lock acquire.

**Store Buffer Parameters and Determinism** The parameters of the store buffer (*i.e.*, the cache geometry) can affect quantum boundaries because buffer overflows cause a quantum to end. Thus, RCDC cannot guarantee the same deterministic execution will arise on two machines with different cache/store buffer configurations. One can address this potential issue by restricting store buffer usage such that its effective size is the same across different machines. The number of threads a program uses, and the parameters used to build quanta (*e.g.*, size) are also implicit inputs that must be replicated to ensure repeatability.

## 6. Results

The goals of our evaluation are to understand the effects of memory ordering relaxation on deterministic execution and to understand how RCDC's mechanisms behave dynamically. To these ends, we evaluate RCDC in two basic ways: (1) a hardware simulator of the actual mechanisms described in the paper; and (2) a software-only implementation of DMP-HB using a compiler and runtime system.

We built a hardware simulation infrastructure using the Intel Pin [21] binary instrumentation tool. The model focuses on the first order effects and includes RCDC's major components, including store buffering in private caches, quantum formation, committing, and consistency models for both DMP-TSO and DMP-HB. For the memory system, private 8-way 32KB L1 and private 8-way 256KB L2 caches for each core, with a 16-way 8MB shared L3. All caches have 64B lines. Instructions take 1 cycle to execute, and it takes an additional 1, 10, 35 and 120 cycles to access the L1, L2, L3 and main memory, respectively. We modeled 2, 4, 8 and 16 processor systems. With the exception of Figure 10, all workloads are run with a target quantum size of 50,000 instructions, except for `ferret` (25k), `fluidanimate` (1k) and `streamcluster` (1k). We determined these parameters by finding the best performance of our workloads, at 16 processors, for each quantum size in the range shown in Figure 10. Quantum commit costs 100 cycles. Error bars indicate the 95% confidence interval for the mean of 10 runs.

Our hardware simulations use version 2.1 of the PARSEC [7] benchmark suite. We used the `simsmall` input set for each workload. Due to excessive memory usage, we were not able to run the `freqmine`, `raytrace` and `facesim` workloads. Due to a lack of support for reader-writer locks and lock-free synchronization in our run-

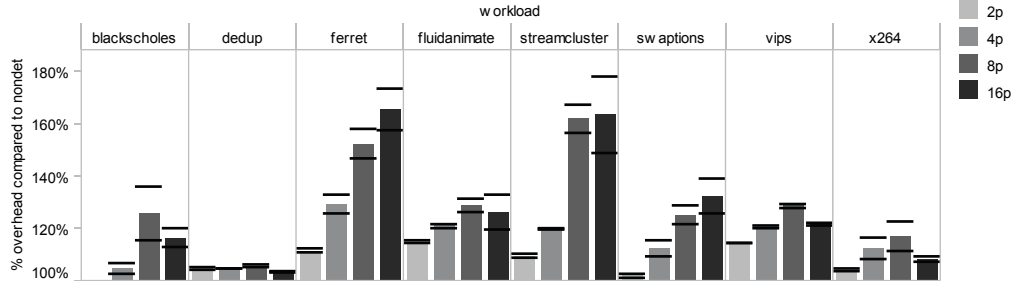


Figure 7. Performance of RCDC normalized to NONDET for 2, 4, 8 and 16 processors.

time system, we were not able to run the bodytrack and canneal workloads, respectively.

Our software-only implementation was built on top of the publicly-available CoreDet [4] compiler and runtime system infrastructure. The source code for our simulator, modifications to CoreDet, and experimental data are available from <http://sampa.cs.washington.edu>.

### 6.1 Performance and Scalability

We start with a performance comparison of RCDC and the non-deterministic baseline (NONDET), as measured using our hardware simulator. Figure 7 plots performance of DMP-HB for 2, 4, 8, and 16 processors normalized to NONDET with the same number of processors. Most applications suffer little performance degradation, but the overheads are still just over 60% in the worst case with 16 threads. Broadly, the performance costs in RCDC come from imbalance (periodic barriers at the end of parallel mode), extra stalls due to costly fences in synchronization operations, and the cost of committing buffered data. We characterize these costs more precisely below. Overall, RCDC provides fully deterministic execution for a modest runtime cost for many of our workloads.

From Figure 7 we can also see how RCDC’s performance scales with additional cores. In a minority of cases (*e.g.*, ferret), RCDC does not scale as well as NONDET. Most of the time, however, RCDC scales just as well as NONDET does, as evidenced by a consistent slowdown despite increasing core counts. Sometimes (*e.g.*, vips) RCDC even closes the performance gap at higher core counts because the underlying benchmark does not scale well even with NONDET. Some of RCDC’s overheads, like reduced cache capacity due to store buffering, can take advantage of additional parallel resources even when the underlying application cannot.

We also implemented a version of DMP-TSO on top of RCDC to assess the benefit of the extra memory reordering relaxation offered by DMP-HB. Figure 8 compares the performance of RCDC-DMP-HB and RCDC-DMP-TSO, normalized to NONDET with the same number of processors. We include only the benchmarks ferret, fluidanimate, and vips; other benchmarks have less frequent synchronization, so the performance of DMP-HB and DMP-TSO is essentially identical. For these three benchmarks, DMP-HB yields markedly better performance compared to DMP-TSO, which comes from the fact that DMP-HB is able to elide many costly fences (*i.e.*, quantum boundaries) that DMP-TSO cannot elide. Figure 9 further justifies these results.

### 6.2 Characterization

To better understand RCDC’s behavior, Figure 9 breaks down the reasons for quantum boundaries. The three reasons a quantum can end are: *instruction count*, which is simply when a quantum has reached its maximum size; *store buffer overflows*, when the store buffer overflows and the thread cannot continue until its buffered data is committed; and *fences*, when a synchronization operation needs a memory fence to ensure the consistency model is upheld.

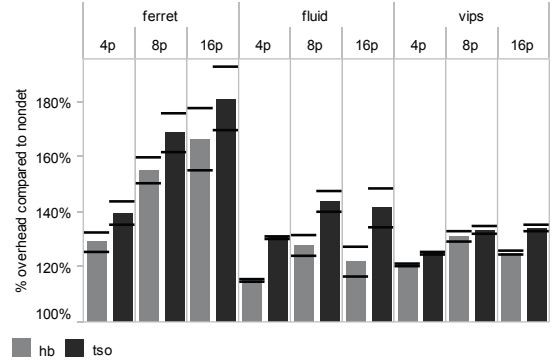


Figure 8. Performance of RCDC-DMP-HB and RCDC-DMP-TSO normalized to NONDET for 4, 8 and 16 processors.

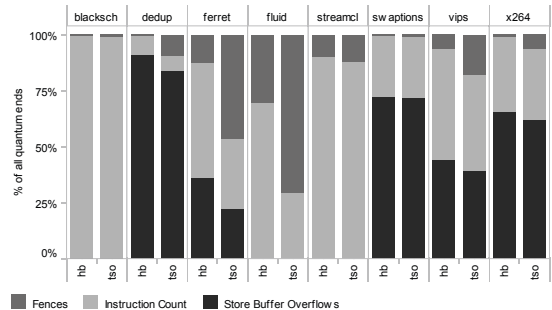


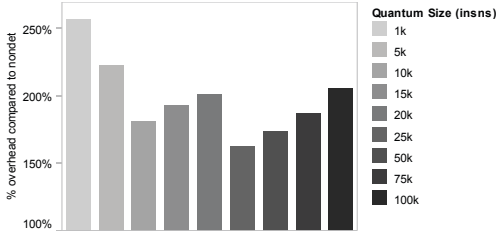
Figure 9. Reasons why quanta end for DMP-HB and DMP-TSO implemented in RCDC, for 16 processors.

Note that DMP-HB has many fewer commits due to fences (the top segment of each bar) than DMP-TSO. This quantifies the effect discussed in Section 2.3 (Figure 2), which is the essence of why DMP-HB offers significantly better performance than DMP-TSO.

Store buffer overflows are a frequent source of quantum imbalance for several workloads. While DMP-HB is effective at reducing the number of fences, some of the premature quantum ends that would have been a fence with DMP-TSO are then replaced with store buffer overflows, which still result in quantum imbalance.

### 6.3 Sensitivity to Quantum Size

We end our RCDC evaluation with a characterization of how maximum quantum size affects performance. Figure 10 shows performance of ferret on a 16-processor RCDC system. The relationship between performance and quantum size can be highly non-linear: for ferret, larger quanta help smooth the effects of frequent quantum rounds, but beyond 25k instructions the extra imbalance of large quanta hurts performance. This effect was noticeable with both DMP-HB and DMP-TSO.

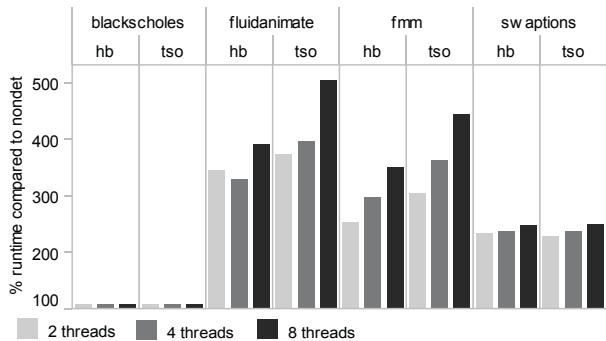


**Figure 10.** Performance of ferret with 16 processors using different quantum sizes.

#### 6.4 Compiler-Runtime Implementation

In addition to the above hardware simulation, we implemented DMP-HB in CoreDet [4]. This implementation required changes only to CoreDet’s synchronization library, not to the compiler nor other parts of the runtime. We evaluated our CoreDet implementation using the PARSEC and SPLASH2 benchmark suites, and include a comparison of the performance of DMP-HB with the performance of DMP-TSO. For this evaluation, we enabled all of CoreDet’s compiler optimizations.

Figure 11 summarizes this evaluation. The performance of DMP-HB is largely the same as that of DMP-TSO, with two exceptions: fluidanimate and fmm. Both these benchmarks have a relatively high frequency of synchronization. DMP-HB’s improved handling of synchronization allows it to increase performance by about 20%: from 5x to 4x overhead for fluidanimate and from 4.5x to 3.5x overhead for fmm. This shows that the benefits of relaxed consistency determinism are not limited to hardware.



**Figure 11.** Performance of CoreDet implementations of DMP-HB and DMP-TSO normalized to NONDET for 2, 4, and 8 threads.

## 7. Related Work

We discussed the most related recent work on deterministic execution [4, 14, 26] in Section 1. DMP [14] is a mostly-hardware approach that yields only sequentially consistent executions. CoreDet [4] follows with a software-only implementation that leverages relaxed memory ordering in a TSO fashion. Calvin [19], done concurrently with our work, is a pure-hardware implementation of DMP-TSO. The dOS [5] operating system provides deterministic execution for unmodified binaries by tracking memory ownership via page tables. Kendo [26] is a synchronization library that provides deterministic guarantees for data-race-free programs written using pthreads. Note that while RCDC also leverages data-race-free models for performance, it does *not* sacrifice determinism in the presence of races, which we believe is instrumental in reaping the full benefits of deterministic execution. Grace [6] is a runtime system that provides deterministic execution for C/C++ fork-join parallel programs, executing each thread in a fork region atomi-

cally and committing them deterministically. Grace’s implementation uses page-based software transactional memory techniques.

There is a significant body of work on deterministic parallel programming languages. Stream-based programming languages, such as StreamIt [31], offer deterministic behavior because communication happens only via explicitly defined channels (streams). Similarly, SHIM [16] is a parallel language that supports explicit communication only via deterministic message passing. NESL [8] and Data Parallel Haskell [12] are examples of data-parallel functional languages that expose sequential semantics to programmers and yield deterministic programs. Another notable class of deterministic languages are implicitly parallel languages, such as Jade [29]. In this model, programmers write programs in a sequential, imperative language and then augment the code with information about how data is accessed. The system then extracts parallelism without violating the original sequential program semantics. More recently, Bocchitto et al., developed Deterministic Parallel Java (DPJ) [9], which is a set of extensions to Java that enable programmers to write deterministic programs via a type and effect system. Determinator [3] proposes a deterministic consistency model that is defined only for programs that are written according to a specialized programming model. Determinator’s implementation is a microkernel that provides deterministic execution using page-based isolation. While deterministic languages are a long-term solution to the problem, the existing options are typically domain-specific and are not widely used: the vast majority of parallel and concurrent programs written today still use mainstream languages like C, C++, and Java.

A widely explored approach to dealing with nondeterminism is recording and replaying interprocessor interaction via shared memory. A notable example is FDR [35, 36], which records shared accesses that lead to communication between processors and performs transitive reduction [25] to reduce log size. More recent work [20, 23, 24] exploited coarse regions of execution to achieve shorter log sizes; other recent work also explored recording partial information during execution and performing search during replay [2, 28]. Tern [13] extends this idea by memoizing partial schedules during testing and steering execution toward these tested paths on future, untested inputs. In contrast to record and replay systems, deterministic execution provides repeatable behavior by default without having to log the shared-memory interactions between processors.

Relaxing memory ordering is a recurring theme in research on shared memory multiprocessors (*e.g.*, weak-ordering [1] and release-consistency [17]). The key motivation for relaxing memory ordering is to allow the system (*i.e.*, the hardware and compiler) to reorder memory operations and consequently enable important performance optimizations. To allow the programmer to control when ordering is required for correctness, relaxed memory models provide fence operations which are typically high-latency operations. For that reason, prior work has explored ways to reduce the cost of fences. A notable example is conditional memory ordering (CMO) [33]. CMO delays the effect of fences until a synchronization operation actually leads to a happens-before relationship between processors. This is an effective optimization because locks typically exhibit locality. CMO’s implementation uses conditional memory fences that are equivalent to RCDC’s `sync_acquire` and `sync_release`.

One of the indirect benefits of RCDC is decreasing the cost of false sharing; when multiple quanta within a round write to the same cache line, they do not need to repeatedly re-acquire exclusive ownership of the line since writes are buffered. Delayed consistency [15] leads to a similar effect but does not provide any deterministic guarantees.

## 8. Conclusions

We have presented RCDC, a new deterministic multiprocessing architecture that leverages memory ordering relaxation to improve performance. We propose a new deterministic execution algorithm that combines deterministic synchronization with weak memory ordering to improve performance by reducing unnecessary stalls when enforcing determinism for arbitrary multithreaded programs. We also propose a hybrid hardware/software design that requires the hardware to provide only software-controlled store buffering and precise instruction counting, thereby reducing hardware complexity. Our results show that RCDC is competitive with non-deterministic multiprocessors, in terms of both absolute performance and scalability, without employing speculation. Moreover, our HW/SW approach allows precise control of when determinism should be enforced, providing flexibility to system software.

We believe this work is an important step toward realistic systems for the deterministic execution of arbitrary programs. Relaxed memory ordering aids performance by avoiding global barriers for synchronization operations while our HW/SW approach provides simplicity and flexibility.

## 9. Acknowledgments

We would like to thank the anonymous reviewers for their valuable comments. We also thank the members of the Sampa group for their feedback on the manuscript. This work was supported in part by NSF CAREER grant CCF-0846004 and a Microsoft Research New Faculty Fellowship.

## References

- [1] S. Adve and M. Hill. Weak Ordering – A New Definition. In *ISCA*, 1990.
- [2] G. Altekar and I. Stoica. ODR: Output-Deterministic Replay for Multicore Debugging. In *SOSP*, 2009.
- [3] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient System-Enforced Deterministic Parallelism. In *OSDI*, 2010.
- [4] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. CoreDet: a Compiler and Runtime System for Deterministic Multithreaded Execution. In *ASPLOS*, 2010.
- [5] T. Bergan, N. Hunt, L. Ceze, and S. Gribble. Deterministic Process Groups in dOS. In *OSDI*, 2010.
- [6] E. Berger, T. Yang, T. Liu, , and G. Novark. Grace: Safe and Efficient Concurrent Programming. In *OOPSLA*, 2009.
- [7] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *PACT*, 2008.
- [8] G. Blelloch. NESL: A Nested Data-Parallel Language (Version 3.1). Technical report, Carnegie Mellon University, Pittsburgh, PA, 2007.
- [9] R. Bocchino, V. Adve, D. Dig, S. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A Type and Effect System for Deterministic Parallel Java. In *OOPSLA*, 2009.
- [10] H.-J. Boehm and S. Adve. Foundations of the C++ Concurrency Memory Model. In *PLDI*, 2008.
- [11] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. BulkSC: Bulk Enforcement of Sequential Consistency. In *ISCA*, 2007.
- [12] M. M. T. Chakravarty, R. Leshchinskiy, S. P. Jones, G. Keller, and S. Marlow. Data Parallel Haskell: A Status Report. In *Workshop on Declarative Aspects of Multicore Programming (DAMP)*, 2007.
- [13] H. Cui, J. Wu, C. che Tsai, and J. Yang. Stable Deterministic Multithreading through Schedule Memoization. In *OSDI*, 2010.
- [14] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: Deterministic Shared Memory Multiprocessing. In *ASPLOS*, 2009.
- [15] M. Dubois, J. Wang, L. Barroso, K. Lee, and Y. Chen. Delayed Consistency and Its Effects on the Miss Rate of Parallel Programs. In *Supercomputing*, 1991.
- [16] S. A. Edwards and O. Tardieu. SHIM: A Deterministic Model for Heterogeneous Embedded Systems. In *EMSOFT*, 2005.
- [17] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *ISCA*, 1990.
- [18] C. Gniady, B. Falsafi, and T. N. Vijaykumar. Is SC + ILP = RC? In *ISCA*, 1999.
- [19] D. Hower, P. Dudnik, D. Wood, and M. Hill. Calvin: Deterministic or Not? Free Will to Choose. In *HPCA*, 2011.
- [20] D. Hower and M. Hill. ReRun: Exploiting Episodes for Lightweight Memory Race Recording. In *ISCA*, 2008.
- [21] C. K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. PIN: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *PLDI*, 2005.
- [22] J. Manson, W. Pugh, and S. Adve. The Java Memory Model. In *POPL*, 2005.
- [23] P. Montesinos, L. Ceze, and J. Torrellas. DeLorean: Recording and Deterministically Replaying Shared-Memory Multiprocessor Execution Efficiently. In *ISCA*, 2008.
- [24] S. Narayanasamy, C. Pereira, and B. Calder. Recording Shared Memory Dependencies Using Strata. In *ASPLOS*, 2006.
- [25] R. H. B. Netzer. Optimal Tracing and Replay for Debugging Shared-Memory Parallel Programs. In *PADD*, 1993.
- [26] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: Efficient Deterministic Multithreading in Software. In *ASPLOS*, 2009.
- [27] J. Ousterhout. Scheduling Techniques for Concurrent Systems. In *ICDCS*, 1982.
- [28] S. Parka, W. Xiong, Z. Yin, R. Kaushik, K. Lee, S. Lu, and Y. Zhou. Do You Have to Reproduce the Bug at the First Replay Attempt? – PRES: Probabilistic Replay with Execution Sketching on Multiprocessors. In *SOSP*, 2009.
- [29] M. Rinard and M. Lam. The Design, Implementation, and Evaluation of Jade. *ACM TOPLAS*, 20(3), 1988.
- [30] K. Russell and D. Detlefs. Eliminating synchronization-related atomic operations with biased locking and bulk rebiasing. In *OOPSLA*, 2006.
- [31] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A Language for Streaming Applications. In *CC*, 2002.
- [32] E. Vallejo, M. Galluzzi, A. Cristal, F. Vallejo, R. Beivide, P. Stenstrom, J. E. Smith, and M. Valero. Implementing Kilo-Instruction Multiprocessors. In *ICPS*, July 2005.
- [33] C. von Praun, H. Cain, J. Choi, and K. Ryu. Conditional Memory Ordering. In *ISCA*, 2006.
- [34] T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Mechanisms for Store-wait-free Multiprocessors. In *ISCA*, 2007.
- [35] M. Xu, R. Bodik, and M. Hill. A “Flight Data Recorder” for Enabling Full-System Multiprocessor Deterministic Replay. In *ISCA*, 2003.
- [36] M. Xu, M. Hill, and R. Bodik. A Regulated Transitive Reduction for Longer Memory Race Recording. In *ASPLOS*, 2006.