DeLorean: Recording and Deterministically Replaying Shared-Memory Multiprocessor Execution Efficiently[*]

Pablo Montesinos, Luis Ceze[†] and Josep Torrellas

Department of Computer Science
University of Illinois at Urbana-Champaign
{pmontesi, torrellas}@cs.uiuc.edu

[†]Department of Computer Science and Engineering
University of Washingon
luisceze@cs.washington.edu

Abstract

Support for deterministic replay of multithreaded execution can greatly help in finding concurrency bugs. For highest effectiveness, replay schemes should (i) record at production-run speed, (ii) keep their logging requirements minute, and (iii) replay at a speed similar to that of the initial execution. In this paper, we propose a new substrate for deterministic replay that provides substantial advances along these axes. In our proposal, processors execute blocks of instructions atomically, as in transactional memory or speculative multithreading, and the system only needs to record the commit *order* of these blocks. We call our scheme *DeLorean*. Our results show that DeLorean records execution at a speed similar to that of Release Consistency (RC) execution and replays at about 82% of its speed. In contrast, most current schemes only record at the speed of Sequential Consistency (SC) execution. Moreover, DeLorean only needs 7.5% of the log size needed by a state-of-the-art scheme. Finally, DeLorean can be configured to need only 0.6% of the log size of the state-of-the-art scheme at the cost of recording at 86% of RC's execution speed — still faster than SC. In this configuration, the log of an 8-processor 5-GHz machine is estimated to be only about 20GB per day.

1. Introduction

Debugging multithreaded codes is challenging because concurrency bugs are typically exercised only under certain timing conditions, and their effects often manifest only after many instructions. With the growing popularity of multicores, it is crucial to find effective debugging techniques for multithreaded codes.

One such technique is hardware-assisted deterministic replay of multithreaded programs. The idea is to record in a log how memory accesses interleave during an initial multithreaded execution. Later, the log is used to replay the execution, recreating the same memory access interleaving — hopefully illuminating what brought the execution to a buggy state. Recent schemes for hardware-assisted deterministic replay include FDR [15], BugNet [7], RTR [16] and Strata [6].

We argue that schemes for deterministic replay have four desirable traits. First, to capture the timing of production-run bugs accurately, they should record at production-run speeds. Second, to support long recording periods, their logging requirements should be minimal. Third, to provide an effective debugging environment, their replay speed should be similar to the initial execution speed. Finally, they should require only modest hardware support.

While existing schemes for hardware-assisted deterministic replay have made major strides in these directions, they still fall short of our goals in some axes. First, they require Sequential Consistency (SC) [6, 7, 15] — a strict consistency model whose typical implementations have relatively low performance and, therefore, can distort the timing of bugs relative to production-run execution. The exception is RTR [16], which introduces an algorithm to record under Total Store Order (TSO). However, the impact of this algorithm on execution speed or log size is not evaluated. Secondly, existing schemes capture shared-memory dependences by logging them individually [15] or in groups [6, 16]. For this, they need to log about one byte per processor per kilo-instruction after compression, which limits the duration of the recorded interval. Finally, it is unclear how fast these schemes replay.

In this paper, we present *DeLorean*, a new approach to deterministic replay that provides substantial advances in some of these axes. DeLorean uses a new execution substrate: one where processors execute large blocks of instructions atomically, separated by processor checkpoints, like in transactional memory or thread-level speculation. To capture a multithreaded execution, DeLorean only needs to record the total *order* in which blocks from different processors commit — not individual shared-memory dependences. This results in a substantial reduction in log size compared to previous schemes. Moreover, since the memory accesses of a processor can overlap and reorder within and across the same-processor blocks, DeLorean can record execution at the speed of the most aggressive consistency models used today — and replay at a comparable speed. While the hardware used is not standard in today's current systems, the required changes are mostly concentrated in the memory system and are arguably simple.

DeLorean offers different execution modes that provide different trade-offs between performance and log size. One mode, called *OrderOnly*, records execution at the speed of Release Consistency (RC) execution and replays at about 82% of RC speed. This is in contrast to most other schemes, which only record at SC execution speeds and provide no details on replay speeds. *OrderOnly* only needs 1.3 bits of memory-ordering log per processor per kilo-instruction, which is 16% of the log size needed by the state-of-the-art RTR design [16]. Moreover, by reorganizing *OrderOnly*'s log

according to the Strata [6] design, we further reduce the log size to 7.5% of RTR's.

A second execution mode called *PicoLog* reduces the memory-ordering log to 0.05 bits per processor per kilo-instruction, which is 0.6% of the log size of RTR [16]. In this mode, the log of an 8-processor 5-GHz machine is estimated to be only about 20GB per day. This mode has a lower execution speed — 86% of RC's execution speed, which is still higher than typical SC speed. Overall, a block-based replay scheme such as DeLorean has great potential to enhance the debugging of production-run multithreaded codes.

This paper is organized as follows. Section 2 gives a background; Sections 3 and 4 introduce DeLorean and its implementation; and Sections 5 and 6 evaluate DeLorean.

2. Background

2.1. Hardware-Assisted Deterministic Replay

Several hardware-based schemes have been proposed for deterministic multiprocessor replay. Bacon and Goldstein [1] captured dependences between concurrent threads by logging the coherence messages in the bus of a multiprocessor using an attached board. The Flight Data Recorder (FDR) [15] is a full-system recorder for directory-based multiprocessors under SC. Like Bacon and Goldstein's scheme [1], FDR observes coherence messages between processors. It improves on the former scheme notably by implementing a hardware version of Netzer's Transitive Reduction (TR) optimization [8]. TR eliminates the need to record dependences that are transitively implied by others.

Figure 1(a) illustrates TR. Processor *P1* writes locations *a* and *b*, and later *P2* accesses *b* and *a*. The dependence *1:Wa→2:Ra* does not need to be recorded because it is transitively implied by *1:Wa→1:Wb*, *1:Wb→2:Wb*, and *2:Wb→2:Ra*. Consequently, FDR only records *1:Wb→2:Wb*. FDR saves the processor ID and instruction count of the two instructions in a Memory Races Log buffer.
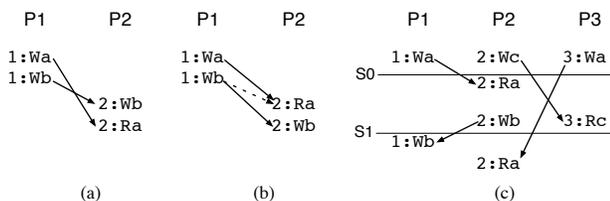

Figure 1. Key insights of previous work on deterministic replay: FDR (a), RTR (b), and Strata (c).

FDR augments each cache block with the count of the last instruction that accessed it. FDR increases the area of the caches by 6.25% and generates a compressed log size of 2MB per 1GHz processor per second [15].

BugNet [7] reuses FDR's hardware to replay user code and shared libraries. It efficiently records the output of all load instructions by compressing them with a hardware-based dictionary scheme.

Xu *et al.* [16] extend FDR in several ways. For the purpose of our paper, we focus on two extensions. The first one is the Regulated Transitive Reduction (RTR). The idea is to judiciously introduce artificial dependences so that Netzer's TR can eliminate other dependences. Figure 1(b) illustrates RTR. The code has dependences *1:Wa→2:Ra* and *1:Wb→2:Wb*. RTR introduces artificial dependence *1:Wb→2:Ra*, which is recorded. Now, RT eliminates the need to record the other two dependences. RTR also saves space by representing recurring dependences with a vector notation.

The second contribution of Xu *et al.*'s work is a recording algorithm for a TSO machine [16]. It extends FDR's algorithm as follows. There is hardware in the processor that detects when a load has violated SC. In this case, the dependence that FDR would log (assuming SC) is not logged. Instead, the hardware logs the value read by the load, which is later fed to the replayer. Supporting TSO is significant because TSO is used in real machines. However, the authors do not evaluate the impact of the new algorithm on execution speed or on log size [16].

We refer to Xu *et al.*'s work [16] as the RTR system, and distinguish between the Base (no TSO) and Advanced (TSO) support. The Base RTR support logs about 1B per processor per kilo-instruction (compressed).

The Strata [6] replay scheme records dependences differently than FDR/RTR. Rather than logging individual dependences with a pair of instruction counts, each Strata log entry (a "Stratum") is a vector of as many counters as processors. Each counter is the number of memory operations issued by the corresponding processor since the last stratum was logged. A stratum is logged before a processor issues the second access of an inter-processor dependence. Figure 1(c) shows a reference trace with the points (*S0* and *S1*) where strata are logged. Right before the second reference of the dependence *1:Wa→2:Ra* is issued, Strata logs the memory reference counts of all 3 processors. The same process is repeated right before the second reference of the dependence *2:Wb→1:Wb*. The other two dependences in the figure do not require the creation of a new stratum: each of them already has its two references in different strata regions.

Strata works with directory- and snoop-based systems — both under SC. Strata can choose to ignore WAR dependences when building the log. In this case, WAR dependences are uncovered at replay at the cost of slowing down the replay with multiple re-executions [6]. The compressed log for 4 processors is 2.2KB per 1M memory references.

2.2. Comparison of HW-Assisted Full-System Replay Schemes

Columns 2-5 of Table 1 show our estimation of how FDR [15], RTR [16], and Strata [6] measure along the four axes that Section 1 argued are key for replay schemes: initial execution speed, log size, replay speed, and hardware needed. Note that, under log size, we consider only the memory-ordering log. This is because the other logs, such as input and DMA logs, are less critical [16] and are handled similarly by the schemes.

FDR, Strata, and Base RTR have been shown to affect execution speed negligibly. Consequently, we list as their execution speed that of the memory consistency model supported, namely SC. Advanced RTR supports TSO but its execution speed has not been measured.

The log size for Base RTR and Strata is smaller than for FDR; there is no information for Advanced RTR. There is also no information on the replay speed of these schemes, but we estimate that, in their current shape, replay is significantly slower than the initial

Property	FDR	RTR		Strata	DeLorean in *OrderOnly* mode: Non-Predefined Chunk Commit Interleaving	DeLorean in *PicoLog* mode: Predefined Chunk Commit Interleaving
		Base	Advanced			
Initial Execution Speed	SC	SC	TSO?	SC	RC	0.86× RC
Memory-Ordering Log Size	Medium	Small	Not reported	Small	Very small	Tiny
Replay Speed	Not reported	Not reported	Not reported	Not reported	0.82×RC	0.72×RC
Hardware Needed	Cache hier	Cache hier	Cache hier + proc	Very little	BulkSC/IT/TCC (Mem hier)	BulkSC/IT/TCC (Mem hier)

Table 1. Comparing the main issues in hardware-assisted, full-system replay schemes.

execution. In the case of Strata, there are three reasons: (i) the log strata likely act as synchronization barriers for replaying processors, (ii) the presence of WARs (if not recorded) requires multiple replays of the same stratum region, and (iii) the replay under directory schemes needs a prepass to combine the multiple logs. In the case of FDR and RTR, every dependence requires a communication between two replaying processors. Moreover, the conservative dependences introduced by RTR may potentially cause processor stalls. Finally, all the schemes require changes in the cache hierarchy, with Advanced RTR requiring changes in the processor as well. Strata has very few hardware requirements.

3. Chunk-Based Execution & Replay

3.1. Motivation

Recent proposals on systems with all-the-time software-annotated transactions such as TCC [5], checkpointed multiprocessors with all-the-time hardware-based transactions such as Implicit Transactions (IT) [13], or high-speed SC implementations such as BulkSC [2] have described an environment where processors continuously execute blocks of consecutive dynamic instructions atomically and in isolation. Such an environment can also be supported in systems with thread-level speculation or with coarse-grain memory ordering support such as ASO [14]. In this environment, the updates made by a block of instructions (which we will call a *Chunk*) only become visible when the chunk commits. Moreover, when two concurrently-executing chunks conflict — there is a data dependence across the two chunks — one of the chunks is typically squashed and retried. The net effect is that the interleaving between the memory accesses of different processors appears to occur *only* at chunk boundaries.

In this environment, recording the execution for replay involves logging the sequence of chunk commits. This provides two fundamental advantages over conventional recorders. The first one is that we can record and replay executions where the memory accesses issued by a processor within a chunk (and in fact across chunks [2]) are fully reordered and overlapped. Recording under such conditions has been a major stumbling block for this area's research, and is recognized by Xu *et al.* [16] as an open problem. The significance of this is that both execution and replay can now proceed at a speed similar to that of a highly-relaxed memory consistency model such as RC. This enables the recording of access interleavings in *true production runs*. It also enables high-speed deterministic replay. In our view, this has major implications on improved code debugging techniques.

The second fundamental advantage is that the memory-ordering log is now very small. Indeed, rather than recording individual dependences, or even groups of them such as in Strata [6] and RTR [16], the log in a chunk-based system only needs to record the *total order* in which chunks from different processors commit. This means that each log entry is short (naively, the ID of the committing

processor and the chunk size) and the log is updated infrequently (chunks can be thousands of instructions long). Furthermore, in an aggressive design, we can predefine when to finish a chunk and start a new one, and even the chunk commit order. In this case, we practically eliminate the need to log at all.

In the rest of this section, we examine the design space in chunk-based systems, present our proposed chunk-based architecture called DeLorean, and then put it in the context of conventional replayers.

3.2. Design Space in Chunk-Based Systems

In a chunk-based system, the memory-ordering log does not store individual or groups of dependences; it only needs to store the total order of chunk commits. In the simplest design, each log entry contains the ID of the processor committing the chunk and the chunk size — measured in number of retired instructions.

We can reduce the log size by either reducing the number of entries or reducing the size of each entry. To reduce the number of entries, we can increase the chunk size — i.e., include more instructions in each chunk. However, increasing the chunk size beyond a certain point is counter-productive. First, we may hurt performance because long chunks increase the chances of inter-chunk conflicts and resulting squashes. Second, we may be unable to increase the effective chunk size. Indeed, a long chunk may access more lines mapping to a cache set than ways the cache has — risking the cache overflow of speculatively updated lines. Before this happens, the chunk has to be forcefully finished and committed.

To reduce the size of each log entry, we can omit the chunk size or the ID of the committing processor from the entry. To be able to omit the chunk size, we need to make "chunking" — i.e., the decision of when to finish a chunk — deterministic. We can accomplish this in different ways. One could be to finish chunks at software annotation points — perhaps similar to what is done in transactional memory systems. Another is to finish chunks when a certain number of memory operations or instructions have been committed — like it is done in BulkSC or IT.

In reality, there may be events that truncate a currently-running chunk and force it to commit before it has reached its "expected" size. This is fine as long as the event reappears deterministically in the replay. An example is an uncached load to an I/O port. The chunk is truncated but its log entry does not need to record its actual size because the uncached load will reappear in the replay and truncate the chunk at the same place.

There are, however, a few events that truncate a currently-running chunk and are not deterministic — e.g., cache overflow as discussed above. We will examine these rare events in Section 4.2. When one such event occurs, the log is augmented with information on: (i) what chunk gets truncated and (ii) its size. With this information, the exact chunking can be reproduced during replay.

To be able to omit the ID of the committing processor from the log entry, we need to "predefine" the chunk commit interleaving.

	Non-Deterministic Chunking	Deterministic Chunking
Non-Predefined Chunk Commit Interleaving	Name: *Order&Size* Execution: Arbiter logs committing processors Processors log chunk sizes Replay: Arbiter consumes Proc-Interleaving log Arbiter enforces order in Proc-Interleaving log Processors consume private Chunk-Size log Processors chunk according to private Chunk-Size log Log size $\approx (\log (\text{\# of procs}) + \log (\text{max\_chunksize})) \times \dfrac{\text{\# dyn. insts}}{\text{chunksize}}$ bits	Name: *OrderOnly* Execution: Arbiter logs committing processors Replay: Arbiter consumes Proc-Interleaving log Arbiter enforces order in Proc-Interleaving log Processors execute chunks normally Log size $\approx \log (\text{\# of procs}) \times \dfrac{\text{\# dyn. insts}}{\text{chunksize}}$ bits
Predefined Chunk Commit Interleaving	—	Name: *PicoLog* Execution: Arbiter enforces predefined commit order Replay: Arbiter enforces predefined commit order Processors execute chunks normally Log size ≈ 0 bits

Table 2. Execution modes in chunk-based systems.

This can be accomplished by enforcing a given commit policy — e.g., pick processors round-robin, allowing them to commit one chunk at a time. The drawback is that, by delaying the commit of completed chunks until their turn, we may slow down execution and replay.

Based on all these ways to reduce the log size, we have three *Execution Modes* in chunk-based systems (Table 2). In the following discussion, we assume that the machine has an *Arbiter* module that observes the order of chunk commits. The arbiter can be associated with the bus controller in a bus-based machine or be an independent module in a machine with a generic network as in BulkSC [2] or Scalable TCC [3].

In the *Order&Size* Mode (top left), chunking is not deterministic and the chunk commit interleaving is not predefined. During execution, the arbiter logs the sequence of committing processor IDs in a *Processor Interleaving* (PI) log. In addition, processors log the size of the chunks they commit in the per-processor *Chunk Size* (CS) log. During replay, each processor generates chunks that are sized according to its CS log, and the arbiter enforces the commit order present in the PI log. The combination of a single PI log and per-processor CS logs constitutes the memory-ordering log. The table shows the estimated size of the memory-ordering log, where *max_chunksize* and *chunksize* are the maximum and average chunk size, respectively.

In the *OrderOnly* Mode (top right), the commit interleaving is not predefined, but chunking is deterministic. Therefore, there is no need to log the chunk size. During execution, the arbiter logs the committing processor IDs in the PI log; during replay, it uses the PI log to enforce the same commit interleaving. The log size is smaller because we only have the PI log. In reality, each processor also keeps a very small CS log where, for each of its few chunks that were truncated non-deterministically, it records both the position in the sequence of chunks committed by the processor and the size.

In the *PicoLog* Mode (bottom right), chunking is deterministic and the commit interleaving is predefined. During both execution and replay, the arbiter enforces a given commit order. There is no PI log. Each processor keeps the very small CS log discussed for *OrderOnly*. The memory-ordering log is largely eliminated.

Note that a mode where the chunking is not deterministic but the chunk commit interleaving is predefined (bottom left) is unattractive. We save log space in the arbiter only to use more in the processors.

3.3. DeLorean: A Chunk-Based Execution-Replay Architecture

DeLorean is our architecture for chunk-based execution-replay (Figure 2). It takes a machine that supports a chunk-based execution environment with a generic network and an arbiter for chunk commit as in BulkSC [2] or Scalable TCC [3], and augments it with the three typical mechanisms for replay: the memory-ordering log, the input logs, and system checkpointing.
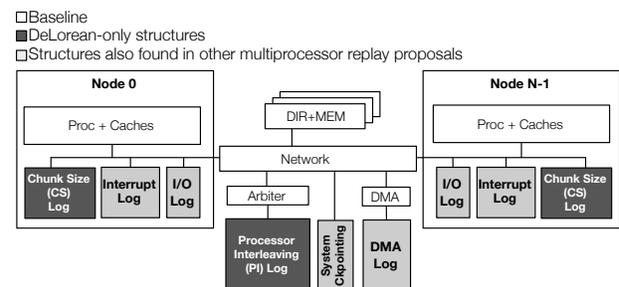

Figure 2. DeLorean architecture.

The memory-ordering log consists of the PI and CS logs. They replace the Memory Races Log Buffer in FDR [15] and RTR [16], and the Strata Log in Strata [6]. They are configured differently depending on which of the three execution modes of Table 2 is desired — allowing for different trade-offs between speed and log size. For each execution mode, Table 3 lists the log entry formats and the time when the logs are updated.

Execution Mode	PI Log		CS Log	
	Log Entry Format	When Updated	Log Entry Format	When Updated
Order&Size	procID	Chunk Commit	size	Chunk Commit
OrderOnly	procID	Chunk Commit	chunkID, size	Chunk Truncation
PicoLog	-	-	chunkID, size	Chunk Truncation

Table 3. PI and CS logs in each mode. The PI log can be reorganized based on Strata to reduce its size (Section 4.3).

In the *Order&Size* and *OrderOnly* modes, when the arbiter gives commit permission to a processor during execution, it also saves the processor's procID in the PI log. During replay, the arbiter uses the

sequence of `procIDs` in the PI log to give commit permissions to processors in the correct order.

We can reorganize the PI log according to the Strata [6] design and reduce its size by half (Section 4.3).

In the *Order&Size* mode, when a processor gets permission to commit a chunk during execution, it records the number of dynamic instructions in the chunk (`size`) in its CS log. In the *OrderOnly* and *PicoLog* modes, the processor only updates its CS log when the chunk to be committed has been truncated non-deterministically. In this case, it stores the processor-local sequence order of the chunk (`chunkID`) and its `size`. During replay, each processor uses its CS log to determine when it needs to terminate each chunk (in *Order&Size*), or only those that were truncated non-deterministically in the initial execution (in *OrderOnly* and *PicoLog*).

The input logs are similar to those in previous replay schemes. As shown in Figure 2, they include one shared log (*DMA* log) and two per-processor logs (*Interrupt* and *I/O* logs). The DMA log logs the data that the DMA writes to memory. During the initial execution, the DMA acts like another processor in that, before it updates memory, it needs to get commit permission from the arbiter. When the arbiter grants permission, the DMA writes to memory and a copy of the data is saved in the DMA log. Moreover, the arbiter creates an entry in the PI log with the DMA's `procID`. Note that, in the *PicoLog* mode, there is no PI log. In this case, the arbiter records the "commit slot" of the DMA operation, namely the current value of a counter that counts the total number of chunk commits since recording started. Later, during replay, when the arbiter finds the DMA's `procID` in its PI log — or, in the *PicoLog* mode, when the arbiter's count of chunk commits matches a saved DMA commit slot — the data in the next entry of the DMA log is consumed.

The per-processor Interrupt log stores, for each interrupt, the time it is received, its type, and its data. Time is recorded as the processor-local `chunkID` of the chunk that initiates execution of the interrupt handler. The per-processor I/O log records the values obtained by I/O loads. Section 4.2 provides more details.

Like previous replay schemes, DeLorean includes system checkpointing support, possibly with schemes such as ReVive [9] or SafetyNet [12]. We do not focus on this issue.

As a summary, the last two columns of Table 1 compare DeLorean in *OrderOnly* and *PicoLog* modes to existing schemes. The memory-ordering log is either very small or practically nil. We will see that DeLorean executes at a speed similar to that of RC execution in *OrderOnly* mode and only modestly slower in *PicoLog* mode. Replay speed will also be shown to be high. The hardware needed is that of a chunk-based system like BulkSC, IT, or TCC, which modifies the memory hierarchy more than the conventional schemes. Processor modifications are largely confined to supporting checkpointing.

3.4. DeLorean in the Context of Other Replayers

3.4.1. Initial Execution

Recall that the advantages of DeLorean in the initial execution are that: (i) it records an environment where memory accesses reorder and overlap substantially, delivering a performance similar to that of a relaxed-consistency machine, and (ii) its log is minuscule. The first advantage comes at the cost of potential squash and reexecution of code sections. In most cases, the squash frequency is

very small and the execution time is largely unaffected. In theory, however, squashing can noticeably slow down an application. This issue is not present in conventional replay schemes.

The minuscule log is the combination of two facts: DeLorean needs few log entries and each entry is small. For this discussion, consider the *OrderOnly* mode. DeLorean naturally combines multiple dependences between two processors into a single dependence — something that RTR does at a smaller scale by creating stricter dependences artificially. This is shown in Figure 3(a), where *all* the dependences between the instructions in the chunks executed by *P4* and *P5* (shown with arrows in the figure) are combined into a *single* PI log entry. Moreover, as shown in the figure, such log entry is *simply P4*'s ID.
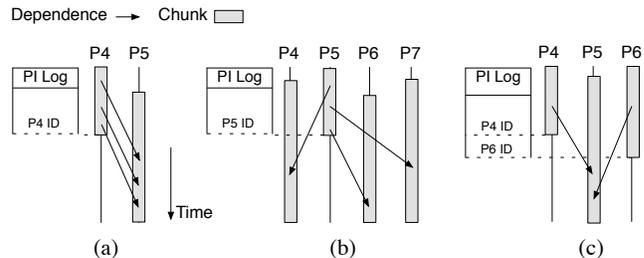

Figure 3. Comparing DeLorean to RTR and Strata.

Similarly, like Strata, DeLorean naturally combines multiple dependences across several processors into a single one. Indeed, as shown in Figure 3(b), when a chunk finishes, it is like a *single-processor* stratum: in the figure, the three dependences are summarized into a *single* log entry, which is *simply P5*'s ID. Unlike Strata, though, DeLorean is unable to combine executions from multiple processors into a single stratum. This is shown in Figure 3(c), where the chunk-commit log entries for *P4* and *P6* are not combined as they would in Strata. However, while a Strata log entry is very wide — it is a vector of as many reference counters as processors the machine has — a DeLorean log entry is only a processor ID.

Interestingly, we can reorganize DeLorean's log according to Strata's design and save space. This is shown in Section 4.3.

3.4.2. Replay

An advantage of DeLorean's replay over previously-proposed schemes is its high speed: all processors execute concurrently, with each processor fully reordering and overlapping its memory accesses. Chunk commit involves a fast check with the arbiter, which is overlapped with the computation of the next chunk, as in BulkSC. Intuitively, therefore, replay speed is likely to be high.

In comparison, the processors in the other schemes replay at most at SC speed (or TSO in Advanced RTR). Moreover, they require more communication between the replaying processors: FDR and RTR require a cross-processor communication for each dependence in the log, while Strata requires the replaying processors to synchronize in a barrier at every log stratum. Finally, as discussed in Section 2.2, Strata has other potential sources of replay overhead.

In practice, replay is likely to proceed on top of a hypervisor layer. A detailed analysis of replay requires considering virtualization issues that are beyond our scope.

4. DeLorean Implementation

We now consider three implementation aspects of DeLorean: implementation choices, exceptional events, and an optimization to further reduce the log size.

4.1. Implementation Choices and Operation

Fundamentally, a chunk-based execution-replay system needs support for speculative tasking and cross-task address disambiguation — the support needed for transactional memory and thread-level speculation. Such support can be implemented in software, hardware, or in a hybrid way. Moreover, there are multiple degrees of freedom, including whether conflict detection and version management are done lazily or eagerly. In addition, the network can be a bus or generic. If generic, we need an arbiter module — which can be designed in a distributed manner to avoid bottlenecks [2].

DeLorean can be implemented in any of these ways. In this paper, we choose to implement DeLorean using the signature-based BulkSC [2]. A reason is that signatures enable fast and efficient memory disambiguation, and an additional log optimization (Section 4.3). Moreover, chunks are automatically created by the hardware, eliminating any need to add software annotations to the application to indicate when the current chunk should finish. Specifically, a chunk in *OrderOnly* and *PicoLog* modes finishes when the processor has committed a certain fixed number of instructions since the chunk started. We call such chunk size the standard chunk size. Finally, we use a generic network with a directory and a single arbiter module. Appendix A overviews BulkSC.

With this support, Figure 4 summarizes DeLorean's operation. During the initial execution in *Order&Size* mode, when a processor such as *P0* or *P1* finishes a chunk, it sends its ID and signature to the arbiter (messages *1* and *2*). Suppose that the arbiter grants permission to *P0* first (message *3*). In this case, the arbiter logs *P0*'s ID (*4*) and propagates the commit operation to the rest of the machine (*5*). While this is in progress, if the arbiter determines that both chunks can commit in parallel, it sends a commit grant to *P1* (*6*), logs *P1*'s ID (*7*), and propagates the commit (*8*). As each processor receives commit permission, it logs the chunk size (*9* and *10*). In *OrderOnly*, steps *9* and *10* are skipped. In *PicoLog*, steps *4*, *7*, *9* and *10* are skipped, and the arbiter grants commit permission to processors according to a predefined order policy, irrespective of the order in which it receives their commit requests. In all cases, a processor does not stall when requesting commit permission; it continues executing its next chunk(s).
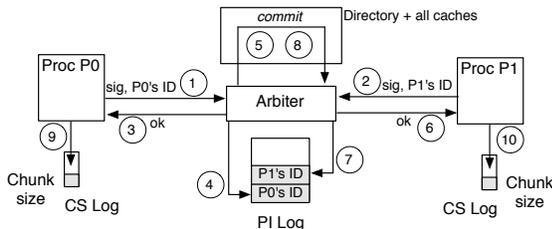

Figure 4. DeLorean's operation.

During replay, suppose that *P1* finishes its chunk first, and the arbiter receives message *2* before *1*. The arbiter checks its PI log (or its predefined order policy in *PicoLog*) and does not grant permission to commit to *P1*. Instead, it waits until it receives the request

from *P0* (message *1*). At that point, it grants permission to commit to *P0* (*3*) and propagates its commit (*5*). The rest of the operation is as in the initial execution but without logging. In addition, in *Order&Size*, processors use the information in their CS log to decide when to finish each chunk.

4.2. Exceptional Events

In DeLorean, the same instruction in the initial and the replayed execution must see exactly the same full-system architectural state. Only then can the stream of committed instructions be guaranteed to be the same in both runs. This means, for example, that the two runs perform the same number of spins on a spinlock, and the same number of system calls and I/O operations.

On the other hand, it is likely that structures that are not visible to the software such as the cache and branch predictor will contain different state in the two runs. This is because, in the two runs, the relative timing of some events may be different, the number of chunk squashes may be different, and structures like the cache and branch predictor may diverge.

Unfortunately, chunk construction is affected by the cache state — through cache overflow that requires finishing the chunk — and by the branch predictor — through wrong-path speculative loads that may cause spurious dependences and induce chunk squashes. Consequently, we need to be careful that chunks are still replayed deterministically.

This section addresses this problem. Table 4 lists the exceptional events that might affect chunk construction during the initial execution. In the following, we consider each one in turn.

Do Not Truncate a Chunk	Truncate a Chunk	
	Deterministically	Non-deterministically
1) Interrupts 2) Traps	1) Reach limit number of instructions 2) Uncached accesses (e.g., I/O initiation) 3) Special system instructions	1) Attempt to overflow the cache 2) Repeated chunk collision (Not for *PicoLog* and not during replay)

Table 4. Exceptional events that may affect chunk construction.

4.2.1. Interrupts and Traps

An interrupt during the initial execution does not truncate the current chunk (Table 4). If the interrupt has low priority, the processor waits until the current chunk completes; if the interrupt has high priority or the current chunk has recently started, the processor squashes the current chunk. In either case, after this, the processor starts a new chunk while initiating execution of the interrupt handler. Moreover, an entry in the Interrupt log is created with: (i) the ID of the new chunk — namely, the number of chunks committed by this processor up to now plus one — and (ii) the interrupt's type and data[1].

During replay, interrupts are replayed in the same way in all execution modes. Specifically, each processor keeps a count of the chunks it has committed so far. When such count is one lower than

---

[1]In *PicoLog* mode, if the interrupt has high priority, the processor can request that the arbiter commit the chunk that handles the interrupt immediately — rather than for the processor to wait until it is its turn to commit. If so, the arbiter records the "commit slot" of the interrupt chunk like it does for DMA requests (Section 3.3), to know when to consume it during replay.

the chunk ID in the next entry of its Interrupt log, the processor starts a new chunk by consuming the Interrupt log entry.

A trap does not truncate the current chunk (Table 4). Instead, the current chunk simply continues to grow, now executing instructions from the trap handler. In addition, the trap is not logged, since it will deterministically reappear during replay. Consider, for example, a page fault trap. The instruction that caused it will cause it again in the replay because the memory state is the same — since wrong-path speculative loads cannot trigger the fault, and squashed chunks cannot modify memory state. The TLB state may be different during replay, but we assume TLBs managed in hardware.

4.2.2. Deterministic Truncation of a Chunk

There are certain events that truncate the chunk that is currently-executing in the processor and that will reappear deterministically during replay (Table 4). They include the trivial case when the number of instructions committed by the chunk reaches the size limit. More importantly, they include instructions that are hard to undo in a speculative environment, like uncached accesses (such as those that initiate I/O operations) and special system instructions (such as those that change the processor frequency or mask/unmask interrupts).

Following BulkSC [2], when one these hard-to-undo instructions is encountered, the currently-running chunk is truncated, the instruction is executed, and a new chunk starts. Typically, the execution of the instruction is not initiated until the previous chunk commits, and the subsequent chunk does not start until the instruction commits. There is no need to log the size of the truncated chunk in the *OrderOnly* and *PicoLog* modes because the event will reoccur in the replay and truncate the chunk at exactly the same instruction. The event itself is not logged either. The only exception is that we must log in the I/O log the value loaded by I/O loads. Such values will be provided to the I/O loads when they are encountered again in the replay.

4.2.3. Non-Deterministic Truncation of a Chunk

Finally, there are two events that truncate the currently-executing chunk and are not deterministic (Table 4). They are the attempt to overflow the cache and repeated chunk collision.

When a chunk accesses more memory lines mapping to a cache set than ways the cache has, there is the danger that speculatively written data may overflow. Before this happens, execution has to stop. Squashing the chunk and re-executing it does not help because the problem will typically reoccur. Instead, we need to truncate the chunk, initiate its commit process, and start a new chunk. Note that the actual point in the chunk where overflow is detected is not deterministic. It depends on the actual reordering of loads and stores; e.g., a wrong-path speculative load may trigger the attempt to displace dirty speculative data. Moreover, multiple speculative chunks of a thread concurrently running may interfere and cause the overflow. Consequently, when a chunk is truncated due to attempted overflow during initial execution, the processor records in its CS log the truncated size of the chunk and, in *OrderOnly* and *PicoLog*, the chunkID as well.

During replay, processors use their CS log to identify what chunks need to be truncated and at what instruction. It is possible that, due to timing differences between initial execution and replay, one such chunk would not have caused overflow during replay —

still, it will be truncated to preserve determinism. It is also possible that, during replay, a chunk unexpectedly attempts to cause overflow and has to be committed sooner than in the initial execution. In this case, the processor, as it commits the shorter chunk, requests the arbiter to let it commit a second chunk immediately after. This second chunk has the rest of the original chunk.

The second non-deterministic event, repeated chunk collision, occurs when, during the initial execution, a chunk is repeatedly squashed by other chunks. The simplest solution proposed in [2] is to progressively reduce the size of the chunk until it can commit. This final size is not deterministic. Consequently, the processor records in its CS log the truncated size of the chunk and, in *OrderOnly*, the chunkID as well. Note that repeated chunk collision cannot occur in *PicoLog*. This is because a chunk can only be squashed by a committing chunk and, in *PicoLog*, there is a predefined chunk commit order.

During replay, processors use their CS log to truncate chunks that were truncated due to collisions in the initial execution. Note that, during replay, chunks may suffer a different set of collisions. However, the problem of repeated chunk collisions cannot occur because chunks have now a predefined commit order.

Overall, even in the presence of all these types of exceptional events, DeLorean's replay is deterministic. Appendix B outlines a proof for why DeLorean's replay is deterministic.

4.3. Optimization: Reducing the PI Log Size by Stratifying It

We can reduce the size of the PI log in *Order&Size* and *OrderOnly* by applying Strata's [6] approach to log construction. Specifically, we design the PI log to record chunk strata. Each stratum is a vector of counters that tell the number of chunks committed per processor since the previous stratum. These chunks have no cross-processor conflicts — although they may have within-processor cross-chunk conflicts. Consequently, we need not record their exact commit sequence because, during replay, those chunks among them that belong to different processors can be executed and committed in any order; those chunks that belong to the same processor will serialize their commit by construction.

DeLorean creates a new stratum when the chunk to log next has these properties: (i) it conflicts with chunks committed by other processors since the last stratum or (ii) it would overflow the counter of chunks committed by this processor since the last stratum. We call this log optimization *stratifying* the PI log.

Figure 5(a) shows an example of this technique. Assume that there is a conflict between the chunks from processors *3* and *0* whose PI log entries are connected with an arrow. The other chunks do not have cross-processor conflicts. Also, assume that each counter in the vector can at most reach *2*. The figure shows that stratum *S1* is created when the chunk from processor *0* is next to be logged, while *S2* is created when the last chunk from processor *1* is next to be logged.

We implement this optimization without affecting DeLorean's recording speed as follows. Chunks commit as usual. However, after a chunk commits, rather than dumping its procID into the PI log, we pass its signature *S* to a *Stratifier Module* (Figure 5(b)). The Stratifier contains: (i) the vector of chunk counters and (ii) one Signature Register (SR) per processor. The latter contain the

Baseline Architecture Configuration			Preferred DeLorean Configurations		
Processor	Memory	BulkSC	*Order&Size*	*OrderOnly*	*PicoLog*

Processors: 8 in a CMP
Frequency: 5.0 GHz
Fetch/issue/comm width: 6/4/5
I-window/ROB size: 80/176
LdSt/Int/FP units: 3/3/2
Ld/St queue entries: 56/56
Int/FP registers: 176/90
Branch penalty: 17 cyc (min)
	Private wback D-L1:
	32KB/4-way/32B-lines
	Round trip: 2 cycles
	MSHRs: 8 entries
	Shared L2:
	8MB/8-way/32B-lines
	Round trip: 13 cyc min
	MSHRs: 32 entries
	Mem round trip: 300 cyc
		Signature: 2 Kbits
		Commit arbitration
		  latency: 30 cyc
		Max. concurrent
		  commits: 4
		# Simultaneous chunks
		  per processor: 2
		# of arbiters: 1
		# of directories: 1
			Chunk Size:
			  2000 inst. maximum
			  25% chunks<2000inst.
			CS log entry:
			  Variable-sized
			  1bit if max size
			  else 12bit
			PI log entry:
			  4bit procID
				Chunk Size:
				  2000 inst.
				CS log entry:
				  21bit distance
				  11bit size
				PI log entry:
				  4bit procID
					Chunk Size:
					  1000 inst.
					CS log entry:
					  22bit distance
					  10bit size
					Commit ordering:
					  round robin

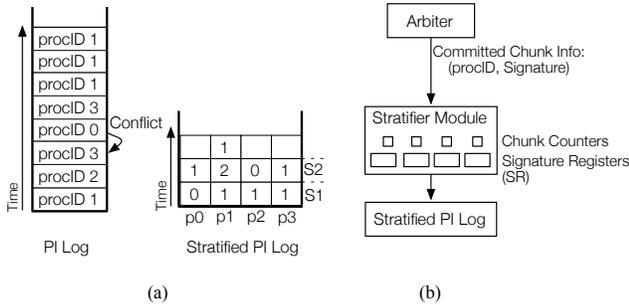**Table 5.** Evaluated architecture configurations.

Figure 5. PI log stratification: example (a) and design (b).

logical-OR of the signatures of all the chunks from a given processor committed since the last stratum. When a new chunk arrives at the Stratifier, if the corresponding counter is at its maximum value, the system creates a new stratum by dumping the counters into the PI log. Then, it sets the corresponding SR and counter to S and 1, respectively, and clears the other SRs and counters. Otherwise, S is logically ANDed with the other processors' SRs — without updating the latter. If there is a conflict, the system creates a new stratum as above. Otherwise, S is logically ORed into the corresponding SR and the corresponding counter is incremented.

5. Evaluation Setup

We use the SESC [11] cycle-accurate execution-driven simulator to evaluate an 8-processor DeLorean Chip Multiprocessor (CMP). The architectural parameters are shown in Table 5. As shown in the table, the BulkSC parameters are largely like those in [2]. The table also shows the preferred parameters for DeLorean's *Order&Size*, *OrderOnly* and *PicoLog* modes. Specifically, *Order&Size* uses chunks of at most 2,000 instructions, variable-sized CS log entries (1 bit if the chunk has maximum size or 12 bits otherwise), and 4-bit PI log entries. The latter encode the IDs of the 8 processors and the DMA. To model an environment with variable-sized chunks, we artificially truncate the chunk probabilistically: we truncate 25% of the chunks in *Order&Size*, giving them a size between 1 and the maximum size using a uniform probability distribution.

OrderOnly uses 2,000-instruction chunks, 32-bit CS log entries (which include 11 bits for the truncated chunk size and, in lieu of chunkID, 21 "distance" bits for the number of chunks committed by the processor since its most-recent truncated chunk), and 4-bit PI log entries. *PicoLog* uses 1,000-instruction chunks, 32-bit CS log entries, and round-robin processor commit order. In our experiments with different chunk sizes in *OrderOnly* and *PicoLog*, we keep the CS log entry size constant, thus changing the distance bits. Our simulator models both initial execution and replay.

All log buffers are enhanced with compression hardware that uses the LZ77 algorithm.

We compare the speed of DeLorean's execution and replay to two other systems that do not support BulkSC, speculative tasking, or logs. The first one is the CMP of Table 5 under RC with speculative execution across fences and hardware exclusive prefetching for stores. We call it *RC*. The second is the CMP of Table 5 under an aggressive SC implementation that includes speculative execution of loads and hardware exclusive prefetching for stores. We call it *SC*. We assume that the performance of the initial execution in FDR [15], Strata [6], and Basic RTR [16] is similar to that of *SC*. Finally, we estimate the performance of Advanced RTR using data on Processor Consistency (PC) performance.

As applications, we use SPLASH-2, SPECjbb2000 and SPECweb2005. The SPLASH-2 codes are evaluated without system references. They run to completion, and include all applications but Volrend (which fails in our infrastructure). SPECjbb2000 and SPECweb2005 are evaluated by interfacing the Simics full-system simulator as a front-end to our SESC simulator. Therefore, we capture system references as well. SPECjbb2000 is configured to use 8 warehouses, while SPECweb2005 runs the e-commerce workload. Each runs for over 1 billion instructions after warm-up.

6. Evaluation

In our evaluation, we assess DeLorean's log size (Section 6.1) and performance (Section 6.2), and characterize *PicoLog* (Section 6.3).

6.1. DeLorean's Log Size

Figure 6 shows the size of the PI and CS logs in *OrderOnly* in bits per processor per kilo-instruction. We evaluate configurations with standard chunk sizes of 1,000, 2,000 and 3,000 instructions. For each of them, we report the size of both logs with and without compression. In the figure, the CS log contribution is stacked atop the PI log's, but it is too small to be seen. The SP2-G.M. bars correspond to the geometric mean of SPLASH-2. For comparison, the figure shows a line with the average size of the compressed Memory Races Log in Basic RTR from [16]. We will use this line as a reference, although we note that the set of applications measured here and in RTR [16] are different.

The figure shows that our preferred 2,000-inst. *OrderOnly* configuration uses on average only 2.1 bits (or 1.3 bits if compressed) per processor per kilo-instruction to store both the PI and CS logs. This means that these compressed logs use only 16% of the space that we estimate is needed by the compressed Memory Races Log in Basic RTR.

Figure 6 also shows that the size of the CS log is negligible. Moreover, as we increase the standard chunk size, the size of the PI
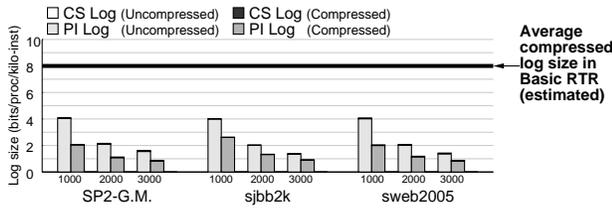
Figure 6. Size of the PI and CS logs in *OrderOnly*. The numbers under the bars are the standard chunk sizes in instructions.

log decreases. This is because there are fewer chunks to commit. However, chunks are also more likely to conflict, and the potentially higher number of squashes may affect performance.

Figure 7 shows the space required by the CS log in *PicoLog*. Recall that *PicoLog* has no PI log. We see that the CS log needs 0.37 bits or fewer per processor per kilo-instruction in all cases — even without compression. Our preferred 1,000-instr. *PicoLog* configuration needs a compressed log with an average of only 0.05 bits per processor per kilo-instruction. To put this in perspective, it implies that, if we assume an IPC of 1, the combined effect of all eight 5-GHz processors is to produce a log of only about 20GB per day. This is a very small log. It is 0.6% of the estimated size needed by the compressed Memory Races Log in Basic RTR. Since the CS log entries are due to chunk truncation caused by attempted cache overflow, we see that such an event is rare.
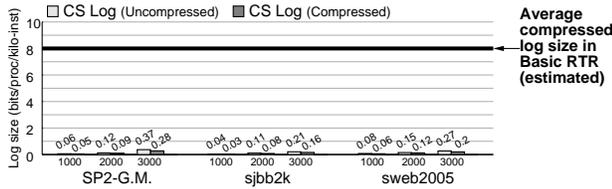

Figure 7. Size of the CS log in *PicoLog*. Recall that *PicoLog* has no PI log. The numbers under the bars are the standard chunk sizes in instructions.

Figure 8 shows *Order&Size*'s PI and CS log sizes. We can see that this execution mode requires larger PI and CS logs, sometimes comparable to Basic RTR log sizes. Our preferred 2,000-instr. compressed configuration uses, on average, 3.7 bits per processor per kilo-instruction. This is 46% of the estimated space needed by the compressed Memory Races Log in Basic RTR.
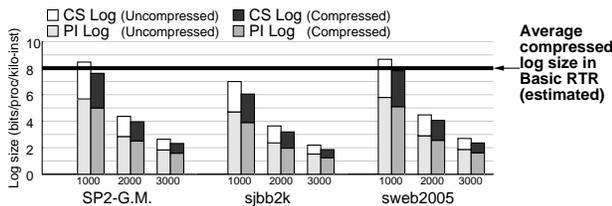

Figure 8. Size of the PI and CS logs in *Order&Size*. The numbers under the bars are the maximum chunk sizes in instructions.

So far, we have roughly compared the per-processor log size of two schemes: DeLorean's 8-processor runs and Basic RTR's 4-processor runs. To compare to Strata, we can use the fact that both the Strata [6] and RTR [16] papers quantitatively compare their

schemes' log sizes to FDR's. Alternatively, we can use the numbers in the Strata paper — which measure different applications than we do and are again for only 4-processor runs. In this case, the Strata paper claims a compressed log size of 2.2KB per million memory operations for the 4 processors combined. DeLorean needs 364B and 13.7B per processor per million memory operations in *OrderOnly* and *PicoLog*, respectively. This is 64% and 2%, respectively, of the space Strata claims to need. However, if, to speed-up Strata's replay, we also add WAR dependences in Strata's log, Strata's log size increases by 25% [6]. In addition, since the size of a Strata log entry is proportional to the number of processors, Strata's log size may increase substantially for 8-processor runs. Consequently, this comparison is likely not very accurate.

6.1.1. Stratifying the PI Log

Figure 9 compares the size of the PI log in 2000-instr. *OrderOnly* without and with stratification. We consider three Stratified PI log designs, which differ in the maximum number of committed chunks allowed per processor per stratum, namely 1, 3, or 7. The bars are normalized to the non-stratified design. We can see that stratifying the PI log while allowing 1 or 3 committed chunks per processor per stratum saves log space. In the case of 1 chunk per processor per stratum, the PI log size decreases by an average of 54%. This results in an average total *OrderOnly* log size of about 0.6 bits per processor per kilo-instruction, or 7.5% of the estimated Basic RTR log size. Allowing 7 chunks per processor per stratum results in wasted space and larger logs in SPECweb2005.
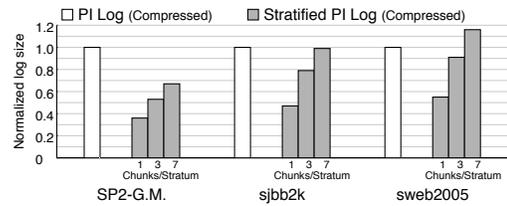

Figure 9. Size of the PI log in *OrderOnly* without and with stratification. The numbers under the bars are the maximum number of chunks per processor per stratum.

6.2. DeLorean's Performance

Figure 10 compares the performance of *RC* and *SC* to that of the initial execution under each of the three DeLorean modes plus the Stratified *OrderOnly* with one chunk. For comparison purposes, we also show the performance of a BulkSC environment. All bars are normalized to the performance of *RC*.

The figure shows that the average performance of *Order&Size* and *OrderOnly* is only 2-3% lower than that of *RC*. Moreover, some of this reduction is the result of running under BulkSC (which causes some chunk squashes), as can be seen by comparing to the *BulkSC* bar. Consequently, we conclude that DeLorean's logging support causes negligible slowdown. The figure also shows that Stratified *OrderOnly* delivers a performance similar to *OrderOnly*. Stratification, therefore, has negligible performance impact.

The figure also shows that *PicoLog* has a lower performance — on average, execution proceeds at 86% of *RC*'s speed. This is still faster than *SC*, which averages 79% of *RC*. As we will see in Section 6.3, *PicoLog*'s lower performance is less caused by load imbal-
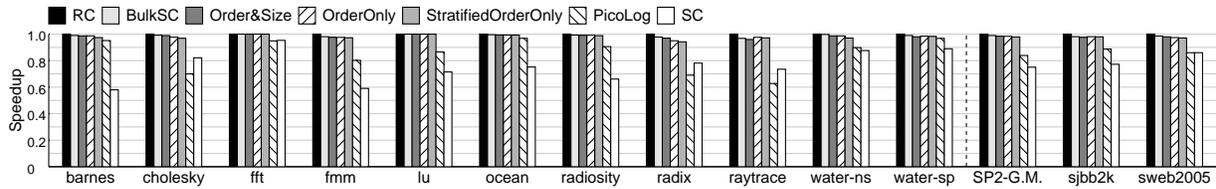
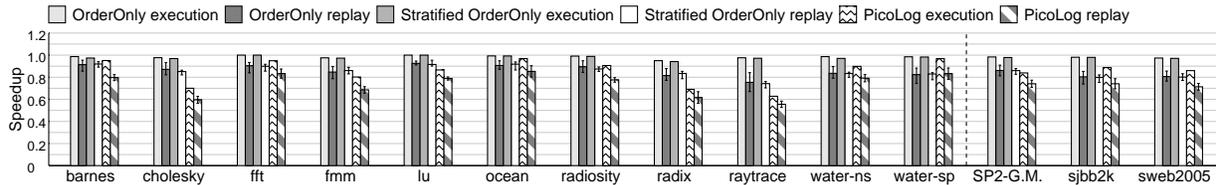**Figure 10.** Performance during initial execution normalized to *RC*.

Figure 11. Performance of several environments during initial execution and replay. All bars are normalized to *RC*.

ance due to round-robin commit ordering than to chunk squashes. This problem especially affects `raytrace`. Overall, it can be argued that a performance 14% lower than *RC* is a modest price to pay for a deterministic replay system that only logs 0.05 bits per processor per kilo-instruction — or 20GB per day for the combined eight 5-GHz processors (Section 6.1).

Given that FDR, Strata, and Base RTR have also been shown to have negligible recording overhead, we estimate their performance with the *SC* bar — which is a fairly aggressive implementation of sequential consistency. It is seen in the figure that all DeLorean execution modes on average outperform *SC*, typically substantially. This is because, through chunk-based execution, DeLorean allows for very aggressive reordering and overlapping of accesses.

If we estimate the performance of Advanced RTR to be that of the machine supporting TSO, we can compare Advanced RTR to DeLorean. TSO's performance is similar to that of Processor Consistency (PC). Since our infrastructure does not model TSO or PC, we simply note that previous work showed that PC's performance is significantly lower than RC's [4, 10] — hence significantly lower than at least that of *OrderOnly* and *Order&Size*. Quantitative comparisons are not possible due to the use of different applications.

6.2.1. Performance During Replay

We use our replay simulator to estimate the performance of DeLorean's replay. Since replay will likely occur under a virtualized environment, we penalize the replay speed by disabling parallel commit and increasing the commit arbitration latency in the arbiter from 30 to 50 cycles. Moreover, in our simulator, we add random delays to the replay execution to ensure that the timings are different from the initial execution. Specifically, we take the PI log from the initial execution and use it in 5 different replay runs. In each run, we add from 10 to 300-cycle stalls before a randomly-selected 30% of the commit operations. We also change the delay of 1.5% randomly-selected cache hits to that of cache misses and the same number of cache misses to cache hits. Finally, we report the average performance of the 5 runs.

Figure 11 compares the performance of *OrderOnly*, Stratified *OrderOnly* with one chunk, and *PicoLog* during initial execution and replay. All bars are normalized to *RC*. From the figure, we see that, on average, both *OrderOnly* and Stratified *OrderOnly* replay at 82% of *RC*'s speed, while *PicoLog* replays at 72% of *RC*. Several factors contribute to the lower performance of replay, namely the

penalties added, the stall of processors waiting to commit, and two effects due to keeping several completed but uncommitted chunks: additional squashes and cache overflows. However, we believe that, at these speeds, deterministic replay opens up new possibilities in concurrency debugging.

Stratifying *OrderOnly* with one chunk does not appear to hurt replay performance. Overall, Stratifying *OrderOnly* has reduced the log size by half, at some hardware cost, without noticeably impacting the speed of execution recording or replay.

6.3. Characterizing Picolog

We perform a sensitivity analysis to determine how *PicoLog*'s performance changes with (i) the number of processors in the CMP, (ii) the standard chunk size in committed instructions, and (iii) the maximum number of chunks that a processor may be executing and are not yet committed. We called the latter the number of simultaneous chunks per processor in Table 5. Figure 12 shows the resulting performance of *PicoLog* relative to *RC* for the same number of processors. Because our infrastructure does not support the commercial applications on 16 processors, the data in the figure corresponds to SPLASH-2 only.
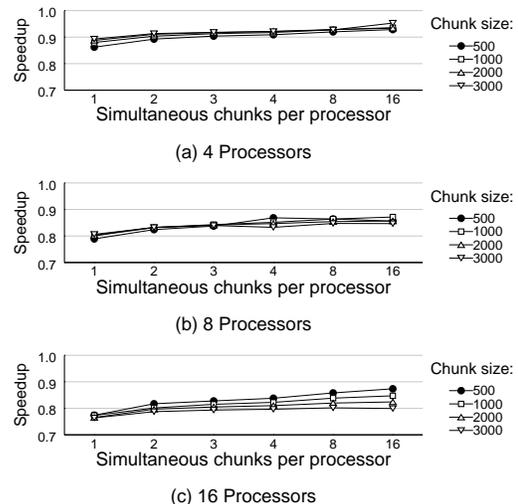

(a) 4 Processors

(b) 8 Processors

(c) 16 Processors

Figure 12. *PicoLog* performance relative to *RC*.

Increasing the number of processors reduces *PicoLog*'s relative performance. For example, with one chunk per processor and 1000-instr. chunks, the performance drops from 87% for 4 processors to 77% for 16. This is because there are more squashes and because it takes longer for a given processor to get its turn to commit.

The latter can be partly mitigated by increasing the number of simultaneous chunks per processor. These additional chunks keep the processor busy while the chunk waits for its turn to commit. However, the figure shows that we quickly get diminishing returns. The more chunks we add, the higher the chance for chunk collisions and attempted cache overflows. In our baseline design (Table 5), we used two simultaneous chunks per processor.

Finally, larger chunk sizes have little effect for 4- or 8-processor systems, but hurt performance for 16-processor systems. Large chunks with many processors tend to induce more conflicts.

Table 6 characterizes *PicoLog* for 8 processors. The *Parallel Commit* columns show data on the commit process. The *Ready Procs* column shows the average number of processors with fully-executed, ready-to-commit chunks at a given time. On average, there are 4.2-5.2 such processors. However, not all of them can commit. Indeed, while chunk commits may overlap if there are no conflicts, they are initiated in a round-robin manner. Consequently, if processor i is not ready to commit, $i+1$ cannot commit. The *Actual Commit* column shows the average number of chunks that end up committing at the same time. The average number is 2.6-3.0.

Appl.	Parallel Commit		Commit Token Passing				
	Ready Procs (Avg)	Actual Commit (Avg)	Proc Ready (%)	Wait Token (Cyc)	Wait Cplete (Cyc)	Token Rndtrip (Cyc)	Stall Cycles (%)
barnes	4.0	2.4	80.4	499	230	661	4.9
cholesky	5.3	3.0	84.7	750	431	793	29.4
fft	3.5	2.3	77.4	411	478	889	2.5
fmm	5.1	3.0	84.0	739	386	788	20.4
lu	3.9	2.3	79.5	487	207	757	5.4
ocean	3.9	2.5	78.4	1067	760	1601	4.2
radiosity	4.9	2.9	82.7	670	403	758	9.3
radix	2.5	2.3	65.6	524	1119	3262	0.3
raytrace	4.6	2.5	78.4	1290	691	1462	34.0
water-ns	4.4	2.6	80.9	541	249	659	9.4
water-sp	4.6	2.6	82.1	489	203	575	2.3
SP2-G.M.	4.2	2.6	79.3	638	403	956	6.0
sjbb2k	5.1	3.0	77.5	1634	694	1841	7.2
sweb2005	5.2	2.9	83.7	1002	612	1346	8.7

Table 6. Characterizing *PicoLog*.

The *Commit Token Passing* columns characterize how the "commit token" is passed around processors. *Proc Ready* is the percentage of time that a processor is ready to commit when it acquires the commit token. On average, it is 77-84%. For those processors that are ready, the *Wait for Token* column is the number of cycles elapsed from when they completed the chunk until they acquire the token; for those processors that are not ready, the *Wait for Complete* column is the number of cycles elapsed from when they receive the token until they complete the chunk. Both of these two numbers must be smaller than the *Token Roundtrip*, which is the number of cycles it takes for the token to circulate through all processors once. Such number is about 600-3,300 cycles. Finally, *Stall Cycles* shows the fraction of cycles that processors stall because they have completed two simultaneous chunks and not received the token. On average, this number is 6-9%.

Table 6 explains the low performance of some codes in Figure 10. For example, consider `raytrace` and `radix`. In `ray-`

`trace`, it can be shown that the squashes are concentrated on a few processors, which slow down the passing of the token for everyone. As a result, processors complete the chunk before receiving the token (the *Wait for Token* cycles are 1,290) and stall often (the fraction of stall cycles is 34.0%). In `radix`, it can be shown that squashes are spread over many processors. As a result, processors receive the token before chunk completion (the *Wait for Complete* cycles are 1,119) and stall little (the fraction of stall cycles is 0.3%).

Finally, DeLorean induces more network traffic than *RC* because of signature traffic chunk squashes. It can be shown that the traffic in *Order&Size* and *OrderOnly* is practically the same as in a plain BulkSC system which, in turn, is on average 9% higher in bytes than in *RC* [2]. In *PicoLog*, due to the higher squash frequency, the total network traffic is on average 17% higher than in *OrderOnly*.

7. Conclusions and Future Work

This paper has proposed DeLorean, a novel scheme for deterministic replay where processors execute groups of instructions atomically. DeLorean has two fundamental advantages over current schemes. First, it records at the speed of the most aggressive memory consistency models used today — and also replays at high speed. This makes it useful for production-run debugging. Second, it summarizes the execution interleaving into a truly small log.

DeLorean's execution modes offer a trade-off between performance and log size. In *OrderOnly*, DeLorean records at the speed of RC execution and replays at 82% of RC speed. In contrast, most other schemes record only at the speed of SC execution and provide no details on replay speed. RTR presents an algorithm for recording TSO executions but does not evaluate its impact on execution speed or log size. Moreover, *OrderOnly* only needs 1.3 bits of compressed memory-ordering log per processor per kilo-instruction and, with stratification, only 0.6 bits. We estimate the latter to be 7.5% of the log size needed by Basic RTR.

In *PicoLog* mode, DeLorean reduces the memory-ordering log to 0.05 bits per processor and kilo-instruction, which we estimate is 0.6% of the log size in Basic RTR. In this mode, we estimate that the total memory-ordering log of an 8-processor 5-GHz machine is only about 20GB per day. Recording speed decreases to 86% of RC execution speed — still higher than typical SC speed. Overall, we conclude that DeLorean greatly enhances the potential of deterministic replay to help debug multithreaded codes.

There are many directions for future work. Three of them stand out. The first one is to adapt DeLorean to work in more conventional multiprocessor environments that do not require hardware for chunk-based execution. The second one relates to the fact that there are a variety of implementations of SC. Specifically, the work involves taking current replay schemes, use very aggressive implementations of SC, and find out how the schemes compare to DeLorean. The third direction involves combining the best aspects of the different recording approaches (DeLorean, RTR, and Strata) into a better approach, along the lines described in Section 3.4 and in PI log stratification. We are working on these areas.

References

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

[2] L. Ceze, J. M. Tuck, P. Montesinos, and J. Torrellas, "BulkSC: Bulk Enforcement of Sequential Consistency," in *Inter. Symp. on Computer Architecture*, June 2007.

[3] H. Chafi, J. Casper, B. D. Carlstrom, A. McDonald, C. C. Minh, W. Baek, C. Kozyrakis, and K. Olukotun, "A Scalable, Non-Blocking Approach to Transactional Memory," in *Inter. Symp. on High Performance Computer Architecture*, Feb. 2007.

[4] K. Gharachorloo, A. Gupta, and J. Hennessy, "Hiding Memory Latency Using Dynamic Scheduling in Shared-Memory Multiprocessors," in *Inter. Symp. on Computer Architecture*, May 1992.

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

[6] S. Narayanasamy, C. Pereira, and B. Calder, "Recording Shared Memory Dependencies Using Strata," in *Inter. Conf. on Arch. Support for Prog. Lang. and Oper. Systems*, Oct. 2006.

[7] S. Narayanasamy, G. Pokam, and B. Calder, "BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging," *Inter. Symp. on Computer Architecture*, June 2005.

[8] R. H. B. Netzer, "Optimal Tracing and Replay for Debugging Shared-Memory Parallel Programs," in *Workshop on Parallel and Distributed Debugging*, May 1993.

[9] M. Prvulovic, Z. Zhang, and J. Torrellas, "ReVive: Cost-effective Architectural Support for Rollback Recovery in Shared-Memory Multiprocessors," in *Inter. Symp. on Computer Architecture*, May 2002.

[10] P. Ranganathan, V. S. Pai, and S. V. Adve, "Using Speculative Retirement and Larger Instruction Windows to Narrow the Performance Gap Between Memory Consistency Models," in *Symp. on Parallel Algorithms and Architectures*, June 1997.

[11] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos, "SESC Simulator," January 2005. http://sesc.sourceforge.net.

[12] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood, "SafetyNet: Improving the Availability of Shared-Memory Multiprocessors with Global Checkpoint/Recovery," in *Inter. Symp. on Computer Architecture*, May 2002.

[13] E. Vallejo, M. Galluzzi, A. Cristal, F. Vallejo, R. Beivide, P. Stenstrom, J. E. Smith, and M. Valero, "Implementing Kilo-Instruction Multiprocessors," in *Inter. Conf. on Pervasive Systems*, July 2005.

[14] T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Mechanisms for Store-Wait-Free Multiprocessors," in *Inter. Symp. on Computer Architecture*, June 2007.

[15] M. Xu, R. Bodik, and M. D. Hill, "A "Flight Data Recorder" for Enabling Full-System Multiprocessor Deterministic Replay," in *Inter. Symp. on Computer Architecture*, June 2003.

[16] M. Xu, M. D. Hill, and R. Bodik, "A Regulated Transitive Reduction (RTR) for Longer Memory Race Recording," in *Inter. Conf. on Arch. Support for Prog. Lang. and Oper. Systems*, Oct. 2006.

Appendix A: BulkSC Overview

In BulkSC [2], a processor takes a checkpoint every *N* committed instructions. The instructions between two checkpoints, called a *Chunk*, are executed speculatively until either they get squashed due to a data dependence with another chunk, or they all commit at once, after the chunk has completed. BulkSC relies on speculative tasking support such as that needed by transactional memory or thread-level speculation to execute dynamic chunks of instructions atomically and in isolation.

A processor can have more than one speculative chunk of a thread executing at a time. Memory accesses by a processor are allowed to fully reorder and overlap both within chunks and across chunks. However, chunks from one or multiple processors must appear to commit in a total order. In practice, for high performance, multiple chunks are allowed to commit concurrently as long as the addresses they have accessed do not overlap. Overall, BulkSC execution supports SC, although its performance is practically the same as that of RC execution [2].

BulkSC requires little modifications to the processor — beyond the ability to take regular checkpoints, which is already feasible today — or L1 cache arrays. Task speculation and address disambiguation are supported by a Bulk Disambiguation Module (BDM) connected to the L1 controller. The addresses read and written by a chunk are hash-encoded in hardware into a Read (R) and Write (W) signature in the BDM. Address disambiguation, chunk commit and chunk squash are implemented with signature operations supported in the BDM.

When a processor wants to commit a chunk, it sends its signatures to an arbiter. The arbiter intersects the signatures with those of the chunks that are currently committing. If the intersection is empty, the arbiter keeps the W signature, forwards it to the directory to make the commit visible to all processors, and grants permission to commit to the processor. While a processor is requesting permission to commit a chunk, it continues executing subsequent chunks — each has its own signatures.

Appendix B: Why DeLorean's Replay is Deterministic

We outline a proof for why DeLorean's replay is deterministic assuming that we use a BulkSC implementation. For brevity, we focus only on the *OrderOnly* mode, but a similar reasoning can be followed for the other two.

In DeLorean, a chunk executes atomically and in isolation. It cannot see any state change while it executes — otherwise it is squashed. The only state it observes is the state of the system when it is about to commit. Thus, we make the following observations.

Observation 1 *The execution path taken inside a chunk only depends on the state of the system when the chunk is about to commit.*

Observation 2 *Non-deterministic events that can modify a processor's instruction stream happen at chunk boundaries, and they are logged. They include both external events (e.g., I/O and interrupts) and internal ones (stores executed by other processors' chunks, which are made visible when those chunks commit).*

Observation 3 *Chunk sizes in the initial execution and in the replay are the same because the decision of when to truncate a chunk is either deterministic (it depends on the instruction stream itself) or reproducible (it is based on information found in the CS log).*

We now define deterministic replay and show that DeLorean's replay is deterministic.

Definition The *Global Commit Count* (GCC) is the number of chunks committed by all processors since execution began.

Definition An *Interval I(n,m)* of execution is the period between GCC=n and GCC=n+m, where m$\geq$1.

Definition The *deterministic replay* of interval *I(n,m)* is a new interval *I'(n,m)* such that the initial and final system states, the number of chunks executed by each processor, the instructions in each chunk and the chunk interleaving are the same in *I* and *I'*.

Theorem *Assuming that a system checkpoint was taken at GCC=n, DeLorean can deterministically replay an execution for the interval I(n,m).*

Proof We use induction on *m*. Start with *m=1*. The PI log has a single entry (say, processor P_i) and the system state has been restored to that at GCC=n. As replay starts, all processors execute, but the arbiter only allows P_i to commit. Because no other processor can commit, the system state that P_i observes is the one at the checkpoint. As P_i replays its chunk, Observation 1 tells us that the path taken by the execution inside the chunk will be the same as in the initial execution. Moreover, as per Observation 3, the number of instructions in the chunk will also be the same as in the initial execution. Finally, if the chunk was affected by an external event in the initial execution, Observation 2 tells us that the event was logged. In the replay of the chunk, we simply reproduce the logged event. Overall, the system has replayed deterministically.

We now assume that the system replayed deterministically for the first *k* committed chunks (*k<m*) and show that it will also do so for the *k+1* commit. At GCC=k, we know that: (i) the next processor in the PI log (say, P_i) is the one that executed next in the initial execution; (ii) P_i is at the same instruction as it was in the initial execution; and (iii) the system state that P_i's chunk observes now in the replay is the same as it observed in the initial execution. We can use observations 1, 2 and 3 like in *m=1* to show that chunk *k+1* is replayed deterministically. Therefore, the system replays deterministically.