DOI:10.1145/1516046.1516068

Two Hardware-Based Approaches for Deterministic Multiprocessor Replay

By Derek R. Hower, Pablo Montesinos, Luis Ceze, Mark D. Hill, and Josep Torrellas

Abstract

Many shared-memory multithreaded executions behave nondeterministically when run on multiprocessor hardware such as emerging multicore systems. Recording nondeterministic events in such executions can enable deterministic replay—e.g., for debugging. Most challenging to record are memory races that can potentially occur on almost all memory references. For this reason, researchers have previously proposed hardware to record key memory race interactions among threads.

The two research groups coauthoring this paper independently uncovered a dual approach: *focus on recording how long threads execute without interacting*. From this common insight, the groups developed two significantly different hardware proposals. *Wisconsin Rerun* makes few changes to standard multicore hardware, while *Illinois DeLorean* promises much smaller log sizes and higher replay speeds. By presenting both proposals in one paper, we seek to illuminate the promise of the joint insight and inspire future designs.

1. INTRODUCTION

Modern computer systems are inherently nondeterministic due to a variety of events that occur during an execution, including I/O, interrupts, and DMA fills. The lack of repeatability that arises from this nondeterminism can make it difficult to develop and maintain correct software. Furthermore, it is likely that the impact of nondeterminism will only increase in the coming years, as commodity systems are now shared-memory multiprocessors. Such systems are not only impacted by the sources of nondeterminism in uniprocessors, but also by the outcome of memory races among concurrent threads.

In an effort to help ease the pain of developing software in a nondeterministic environment, researchers have proposed adding *deterministic replay* capabilities to computer systems. A system with a deterministic replay capability can record sufficient information during an execution to enable a replayer to (later) create an equivalent execution despite the inherent sources of nondeterminism that exist. With the ability to replay an execution verbatim, many new applications may be possible:

Debugging: Deterministic replay could be used to provide the illusion of a *time-travel debugger* that has the ability to selectively execute both forward and backward in time.

Security: Deterministic replay could also be used to enhance the security of software by providing the means for an in-depth analysis of an attack, hopefully leading to rapid patch deployment and a reduction in the economic impact of new threats.

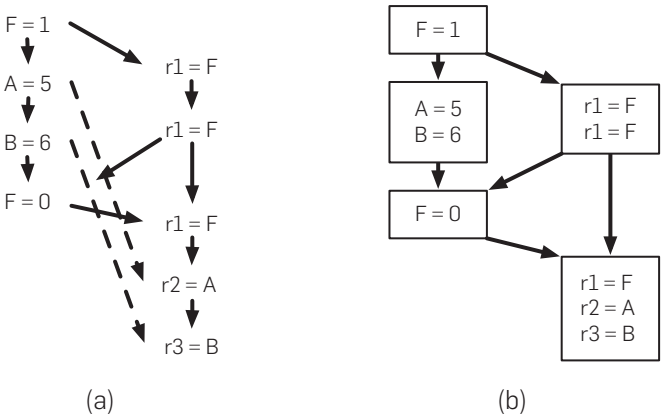
Fault Tolerance: With the ability to replay an execution, it may also be possible to develop hot-standby systems for critical service providers using commodity hardware. A virtual machine (VM) could, for example, be fed, in real time, the replay log of a primary server running on a physically separate machine. The standby VM could use the replay log to mimic the primary's execution, so that in the event that the primary fails, the backup can take over operation with almost zero downtime.

As existing commercial products have already shown, deterministic replay can be achieved with a software-only solution when executing in a uniprocessor environment.[18] This is due, in part, to the fact that sources of nondeterminism in a uniprocessor, such as interrupts or I/O, are relatively rare events that take a long time to complete. However, when executing in a shared-memory multiprocessor environment, memory races, which can potentially occur on every memory access, are another source of nondeterminism. All-software solutions exist,[4, 8] but results show that they do not perform well on workloads that interact frequently. Thus, it is likely that a general solution will require hardware support. To this end, Bacon and Goldstein[2] originally proposed recording all snooping coherence transactions, which, while fast, produced a serial and voluminous log (see Figure 1).

Xu et al.[16] modernized hardware support for multiprocessor deterministic replay in general and *memory race recording* in particular. A memory race recorder is responsible for logging enough information to reconstruct the order of all fine-grained memory interleavings that occur during an execution. To reduce the amount of information that needs to be logged (so that longer periods can be recorded for a fixed hardware cost), the system proposed by Xu et al. implemented in hardware an enhancement to Netzer's transitive reduction optimization.[13] The idea is to skip the logging of those races that can be implied through transitivity, i.e., those races

The original Wisconsin Rerun[6] paper as well as the original Illinois DeLorean[11] paper were published in the *Proceedings of the 35th Annual International Symposium on Computer Architecture* (June 2008).

Figure 1: An example of efficient race recording using (a) an explicit transitive reduction and (b) independent regions. In (a), solid lines between threads are races written to the log, while dashed lines are those races implied through transitivity.



implied through the combination of previously logged races and sequential program semantics. Figure 1a illustrates a transitive reduction. Inter-thread races between instructions accessing locations A and B, respectively, are not logged since they are implied by the recorded race for location F.

While both the original[16] and follow-on[17] work by Xu et al. were successful in achieving efficient log compression (~1B/1000 instructions executed), they required a large amount of hardware state, on the order of an additional L1 cache per core, in order to do so. Subsequent work by Narayanasamy et al.[12] on the Strata race recorder reduced this hardware requirement but, as results in Hower and Hill[6] show, may not scale well as the number of hardware contexts in a system increases. This is largely because Strata writes global information to its log entries that contains a component from each hardware thread context in the system.

A key observation, discovered independently by the authors of this paper at the Universities of Illinois and Wisconsin, is that by focusing on regions of *independence*, rather than on individual dependencies, an efficient and scalable memory race recorder can be made *without* sacrificing logging efficiency. Figure 1b illustrates this notion by breaking the execution of Figure 1a into an ordered series of independent execution regions. Because intra-thread dependencies are implicit and do not need to be recorded, the execution in Figure 1b can be completely described by the three inter-thread dependencies, which is the same amount of information required after a transitivity reduction shown in Figure 1a.

The authors of this paper have developed two different systems, called *Rerun*[6] and *DeLorean*,[11] that both exploit the same independence observation described above. These systems, presented in the same session of ISCA 2008, exemplify different trade-offs in terms of logging efficiency and implementation complexity. Rerun can be implemented with small modifications to existing memory system architectures but writes a larger log than DeLorean. DeLorean can achieve a greater log size reduction and a higher replay speed but requires novel hardware to do so.

2. RERUN

Wisconsin Rerun[6] exploits the concept of *episodic race recording* to achieve efficient logging with only small modifications to existing memory system architectures. The Rerun race recorder does not interfere with a running program in any way; it is an impartial observer of a running execution, and as such avoids artificially perturbing the execution under observation.

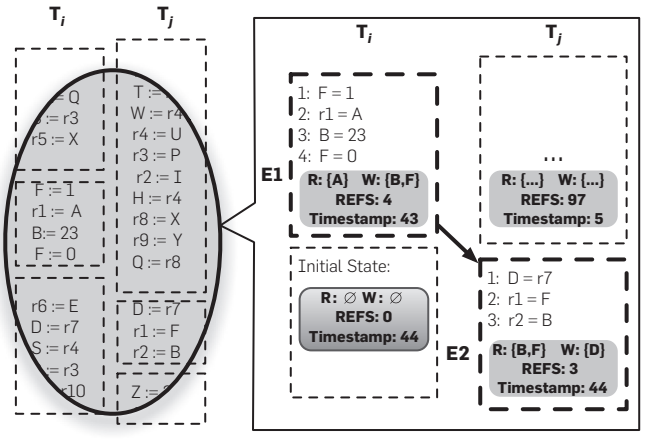
2.1. Episodic memory race recording

This section develops insights behind Rerun. It motivates Rerun with an example, gives key definitions, and explains how Rerun establishes and orders episodes.

Motivating Example and Key Ideas: Consider the execution in Figure 2 that highlights two threads *i* and *j* executing on a multicore system. Dynamic instructions 1–4 of thread *i* happen to execute without interacting with instructions running concurrently on thread *j*. We call these instructions, collectively labeled E_1, an episode in thread *i*'s execution. Similarly, instructions 1–3 of thread *j* execute without interaction and constitute an episode E_2 for thread *j*. As soon as a thread's episode ends, a new episode begins. Thus, every instruction execution is contained in an episode, and episodes cover the entire execution (right side of Figure 2).

Rerun must solve two subproblems in order to ensure that enough episodic information is recorded to enable deterministic replay of all memory races. First, it must determine when an episode ends, and, by extension, when the next one begins. To remain independent, an episode *E* must end when another thread issues a memory reference that *conflicts* with references made in episode *E*. Two memory accesses conflict if they reference the same memory block, are from different threads, and at least one is a write. For example, episode E_1 in Figure 2 ends because thread *j* accesses the variable *F* that was previously written (i.e., *F* is in the write set of E_1). Formally, for all combinations of episodes *E* and *F*

Figure 2: An example of episodic recording. Dashed lines indicate episode boundaries. In the blown up diagram of threads *i* and *j*, the shaded boxes show the state of the episode as it ends, including the read and write sets, memory reference counter, and the timestamp. The shaded box in the last episode of thread *i* shows the initial episode state.



in an execution, the *no-conflict* condition of Equation 1 must hold. Let $R_E(W_E)$ denote episode E's read (write) set:

$$[W_E \cap (R_F \cup W_F) = \varnothing] \wedge [R_E \cap W_F = \varnothing] \quad (1)$$

Importantly, while an episode *must* end to avoid conflicts, episodes *may* end early for any or no reason. In Section 2.2, we will ease implementation cost by ending some episodes early.

Second, an episodic recorder must establish an ordering of episodes among threads. Rerun does so using Lamport scalar clocks,[7] which is a technique that guarantees the timestamp of any episode E executing on thread i has a scalar value that is greater than the timestamp of any episode on which E is dependent and less than the timestamp of any episode dependent on E. In our example, since the episode E_1 ends with a timestamp of 43, the subsequent episode executing on thread j (E_2), which uses block F after thread i, must be assigned a timestamp of (at least) 44.

The specific Rerun mechanism meets three conditions sufficient for a Lamport scalar clock implementation:

1. When an episode E on $thread_E$ begins, its $timestamp_E$ begins with a value one greater than the timestamp of the previous episode executed by $thread_E$ (or 0 if episode E is $thread_E$'s first episode).
2. When an episode E adds a block to its read set R_E that was most-recently in the write set W_D of completed episode D, it sets its $timestamp_E$ to $maximum[timestamp_E, timestamp_{D+1}]$.
3. When an episode E adds a block to its write set W_E that was most-recently in the write set W_{D_0} of completed episode D_0 or in the read set of any episode $D_1 \ldots D_N$, it sets its $timestamp_E$ to $maximum[timestamp_E, timestamp_{D_0}+1, timestamp_{D_1}+1, \ldots, timestamp_{D_N}+1]$.

When each episode E ends, Rerun logs both $timestamp_E$ and $references_E$ in a per-thread log. $references_E$ is a count of memory references completed in E, and is used to record the episode length. The Lamport clock algorithm ensures that the execution order of all conflicting episodes corresponds to monotonically increasing timestamps. Two episodes can only be assigned the same timestamp if they do not conflict and, thus, can be replayed in any alternative order with affecting replay fidelity.

A replayer (not shown) uses information about episode duration and ordering to reconstruct an execution with the same behavior. If episodes are replayed in timestamp order, then the replayed execution will be logically equivalent to the recorded execution. Unfortunately, the use of Lamport scalar clocks make Rerun's replay (mostly) sequential.

2.2. Rerun implementation

Here we develop a Rerun implementation for a system based on a cache-coherent multicore chip, with key parameters shown in Table 1. Though we describe Rerun in terms of a specific base system, the mechanism can be extended to other systems, including those with a TSO memory consistency model, out-of-order cores, multithreaded cores, alternate cache designs, and snooping coherence. Details of the changes needed to accommodate these alternate architectures can be found in the original paper.[6]

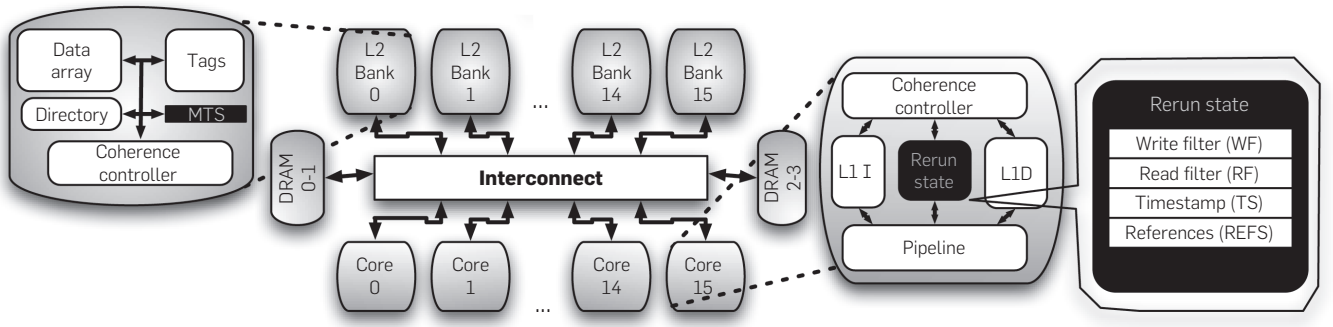
Rerun Hardware: As Figure 3 depicts, Rerun adds modest hardware state to the base system. To each core, Rerun adds:

- Read and Write Bloom filters, *WF* and *RF*, to track the current episode's write and read sets (e.g., 32B and 128B, respectively).
- A Timestamp Register, *TS*, to hold the Lamport Clock of the current episode executing on the core (e.g., 4B).
- A Memory Reference Counter, *REFS*, to record the current episode's references (e.g., 2B).

Table 1: Base system configuration.

Cores	16, in-order, 3 GHz
L1 Caches	Split I&D, private, 32K four-way set associative, write-back, 64B lines, LRU replacement, three cycle hit
L2 Caches	Unified, shared, inclusive, 8M 8-way set associative, write-back, 16 banks, LRU replacement, 37 cycle hit
Directory	Full bit vector in L2
Memory	4G DRAM, 300 cycle access
Coherence	MESI directory, silent replacements
Consistency Model	Sequential consistency (SC)

Figure 3: Rerun hardware.



To each L2 cache bank, Rerun also adds a "memory" timestamp register, *MTS* (e.g., 4B). This register holds the maximum of all timestamps for victimized blocks that map to its bank. A victimized block is one replaced from an L1 cache, and its timestamp is the timestamp of the core at the time of victimization.

Finally, coherence response messages—data, acknowledgements, and writebacks—carry logical timestamps. Book-keeping state, such as a per-core pointer to the end of its log, is not shown.

Rerun Operation: During execution, Rerun monitors the no-conflict equation by comparing the addresses of incoming coherence requests to those in *RF* and *WF*. When a conflict is detected, Rerun writes the tuple <*TS*, *REFS*> to a per-thread log, then begins a new episode by resetting *REFS*, *WF*, and *RF*, and by incrementing the local timestamp *TS* according to the algorithm in Section 2.1.

By gracefully handling virtualization events, Rerun allows programmers to view logs as *per thread*, rather than *per core*. At a context switch, the OS ends the core's current episode by writing *REFS* and *TS* state to the log. When the thread is rescheduled, it begins a new episode with reset *WF*, *RF*, and *REFS*, and a timestamp equal to the max of the last logged *TS* for that thread and the *TS* of the core on which the thread is rescheduled. Similarly, Rerun can handle paging by ensuring that TLB shootdowns end episodes.

Rerun also ends episodes when implementation resources are about to be exhausted. Ending episodes just before 64K memory references, for example, allows *REFS* to be logged in 2B.

2.3. Evaluation

Methods: We evaluate the Rerun recording system using the Wisconsin GEMS[10] full system simulation infrastructure. The simulator configuration matches the baseline shown in Table 1 with the addition of Rerun hardware support. Experiments were run using the Wisconsin Commercial Workload Suite.[1] We tested Rerun with these workloads and a microbenchmark, racey, that uses number theory to produce an execution whose outcome is highly sensitive to memory race ordering (available at www.cs.wisc.edu/~markhill/racey.html).

Rerun Performance: Figure 4 shows the performance of Rerun on all four commercial workloads. Rerun achieves an uncompressed log size of about 4B logged per 1000 instructions. Importantly, we notice modest variation among the log size of each workload, leading us to believe that Rerun can perform well under a variety of memory access patterns.

We show the relative performance of Rerun in comparison to the prior state of the art in memory race recording in Figure 5. Rerun achieves a log size comparable to the most efficient prior recorder (RTR[17]), but does so with a fraction of the hardware cost (~0.2KB per core vs. 24KB per core). Like RTR, and unlike Strata,[12] Rerun scales well as the number of cores in the system increases, due, in part, to the fact that Rerun and RTR both write thread-local log entries rather than a global entry with a component from each thread.

Figure 4: Rerun absolute log size.

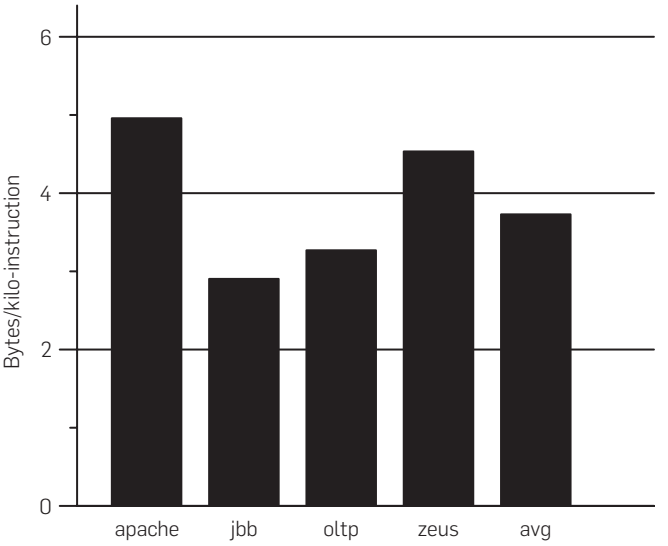
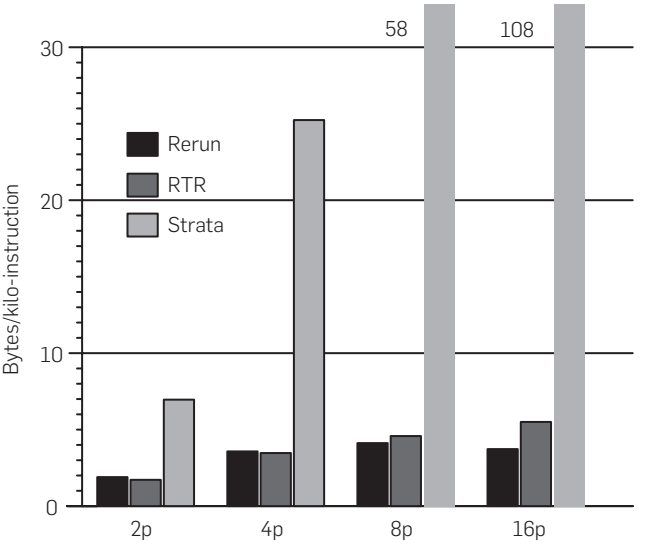


Figure 5: Hardware cost comparison to RTR and Strata.



3. DELOREAN

Illinois DeLorean[11] is a new approach to deterministic replay that exploits the opportunities afforded by a new execution substrate: one where processors continuously execute large blocks of instructions atomically, separated by register checkpoints.[3, 5, 9, 15] In this environment, to capture a multi-threaded execution for deterministic replay, DeLorean only needs to log the total *order* in which blocks from different processors commit.

This approach has several advantages. First, it results in a substantial reduction in log size compared to previous schemes—at least about one order of magnitude. Second, DeLorean can replay at a speed comparable to that of the initial execution. Finally, in an aggressive operation mode, where DeLorean predefines the commit order of the blocks

from different processors, DeLorean generates only a very tiny log—although there is a performance cost. While *DeLorean*'s execution substrate is not standard in today's hardware systems, the required changes are mostly concentrated in the memory system.

3.1. The DeLorean idea

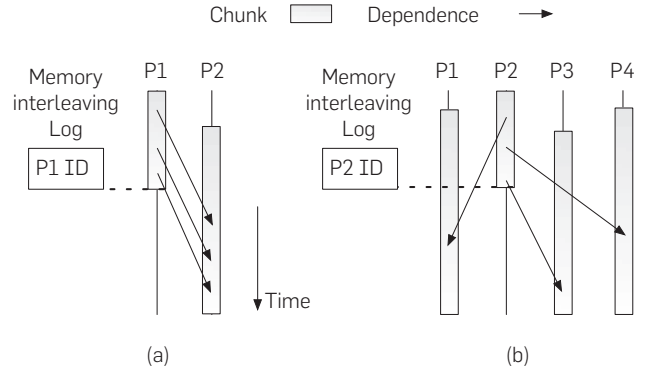
There have been several proposals for multiprocessors where processors continuously execute blocks of consecutive dynamic instructions atomically and in isolation.[3, 5, 9, 15] In this environment, the updates made by a block of instructions (or *Chunk*) only become visible when the chunk commits. When two chunks running concurrently on two different processors conflict—there is a data dependence across the two chunks—the hardware typically squashes and retries one the chunks. Moreover, after a chunk completes execution, there is an optimized global commit step in an arbiter module that informs the relevant processors that the chunk is committed. The net effect is that the interleaving between the memory accesses of different processors appears to occur *only* at chunk boundaries.

In such environment, recording the execution for replay simply involves logging the total sequence of chunk commits. This has two very important consequences for replay systems. The first one is that the memory ordering log is now very small. Indeed, rather than recording individual dependences or groups of them like in all past proposals, 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 (the ID of the committing processor, if all chunks have the same size), and that the log is updated infrequently (chunks are thousands of instructions long).

The second consequence is that, because the memory accesses issued by a processor inside a chunk are not visible to the rest of the processors until the chunk commits, such accesses can be fully reordered and overlapped. This means that both execution and replay under DeLorean proceed at a high speed.

DeLorean naturally combines multiple data dependences between two or more processors into a single entry in the log that records the memory interleaving—the Memory Interleaving Log. An example is shown in Figure 6a, where *all* the dependences between the accesses in the chunks executed by processors *P1* and *P2* (shown with arrows in the figure) are combined into a *single* entry in the log. The figure also shows that such log entry is *simply P1*'s ID. In a second example shown in Figure 6b, multiple dependences across several processors are summarized in a single log entry. Specifically, the *single* log entry inserted when the chunk from *P2* commits is enough to summarize the three dependences.

Figure 6: Combining multiple dependences into a single log entry.



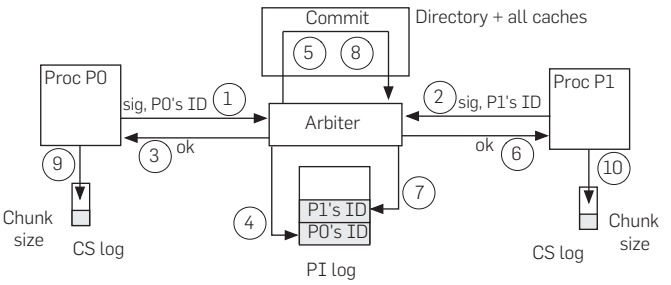
3.2. DeLorean execution modes

DeLorean provides two main execution modes, namely *OrderOnly* and *PicoLog*. To understand them, we start by describing a naive, third execution mode called *Order&Size*. In *Order&Size*, each log entry contains the ID of the processor committing the chunk and the chunk size—measured in number of retired instructions. During execution, an arbiter module (a simple state machine that enforces chunk commit order[3]) logs the sequence of committing processor IDs in a *Processor Interleaving* (PI) log. At the same time, processors record the size of the chunk they commit in a per-processor Chunk Size (CS) log. The combination of a single PI log and per-processor CS logs constitutes the Memory Interleaving Log.

Figure 7 shows DeLorean's operation in *Order&Size* mode. During the initial execution, when a processor such as *P0* or *P1* finishes a chunk, it sends a request-to-commit message to the arbiter (steps *1* and *2*). Such messages contain the processor IDs plus Bloom-filter signatures that summarize the memory footprint of the chunks[3] (*sig* in the figure). Suppose that the arbiter grants permission to *P0* first (step *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 message 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*).

Our first DeLorean execution mode, called *OrderOnly*, omits logging chunk sizes by making "chunking"—i.e., the decision of when to finish a chunk—deterministic. DeLorean accomplishes this by finishing chunks when a fixed number of instructions have been committed. In reality, certain events 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. For example, consider an uncached load to an I/O port. The chunk is truncated but its log entry does not

Figure 7: DeLorean's operation.



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. When one such event occurs, the CS log adds an entry with: (1) what chunk gets truncated (its position in the sequence of chunks committed by the processor) and (2) its size. With this information, the exact chunking can be reproduced during replay.

Consequently, *OrderOnly* generates a PI log with only processor IDs and very small per-processor CS logs. For the large majority of chunks, steps *9* and *10* in Figure 7 are skipped.

Our second DeLorean execution mode, called *PicoLog*, builds on *OrderOnly* and additionally eliminates the need for a PI log by "predefining" the chunk commit interleaving during both initial execution and replay. This is accomplished by enforcing a given commit policy—e.g., pick processors round-robin, allowing them to commit one chunk at a time. It needs only the tiny per-processor CS log discussed for *OrderOnly*. Thus, *the Memory Interleaving Log is largely eliminated*. The drawback is that, by delaying the commit of completed chunks until their turn, *PicoLog* may slow down execution and replay.

Looking at Figure 7, *PicoLog* skips steps *4*, *7* and, typically, *9* and *10*. The arbiter grants commit permission to processors according to a predefined order policy, irrespective of the order in which it receives their commit requests. Note, however, that a processor does not stall when requesting commit permission; it continues executing its next chunk(s).[3]

Table 2 shows the PI and CS logs in each of the two execution modes and *Order&Size*.

3.3. DeLorean implementation

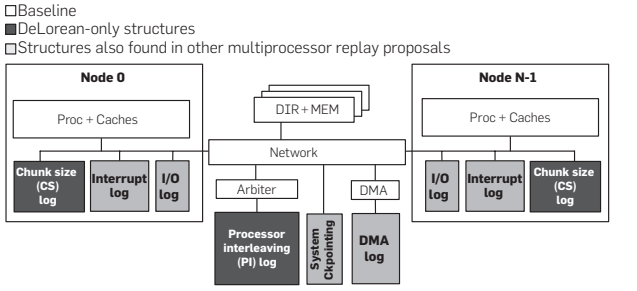
Our DeLorean implementation uses a machine that supports a chunk-based execution environment with a generic network and an arbiter. It augments it with the three typical mechanisms for replay: the Memory Interleaving Log (consisting of the PI and CS logs), the input logs, and system checkpointing (Figure 8).

The input logs are similar to those in previous replay schemes. As shown in Figure 8, they include one shared log (*DMA* log) and two per-processor logs (*Interrupt* and *I/O* logs). The DMA acts like another processor in that, before it updates memory, it needs to get commit permission from the arbiter. Once permission is granted, the DMA log logs the data that the DMA writes to memory. The per-processor

Table 2: PI and CS logs in each execution mode.

	PI Log		CS Log	
Execution Mode	**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

Figure 8: Overall DeLorean system implementation.



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. Like in previous replay schemes, DeLorean includes system checkpointing support.

3.4. DeLorean replay

During replay, processors must execute the same chunks and commit them in the same order. In *Order&Size*, each processor generates chunks that are sized according to its CS log, while in *OrderOnly* and *PicoLog*, processors use the CS log only to recreate the chunks that were truncated nondeterministically. In *Order&Size* and *OrderOnly*, the arbiter enforces the commit order present in the PI log.

As an example, consider the log generated during initial execution as shown in Figure 7. During replay, suppose that *P1* finishes its chunk before *P0*, 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, processors use their CS log to decide when to finish each chunk (*Order&Size*) or those chunks truncated nondeterministically during the initial execution (*OrderOnly* and *PicoLog*).

Thanks to our chunk-based substrate, during replay all processors execute concurrently. Moreover, each processor fully reorders and overlaps its memory accesses within a chunk. Chunk commit involves a fast check with the arbiter.[3] The processor overlaps such check with the computation of its next chunk.

3.5. Exceptional events

In DeLorean, the same instruction in the initial and the replayed execution must see exactly the same full-system architectural state. 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.

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

Table 3: Exceptional events that may affect chunk construction.

Do Not Truncate a Chunk	Truncate a Chunk	
	Deterministically	**Nondeterministically**
1. Interrupts 2. Traps	1. Reach limit number of instructions 2. Uncached accesses (e.g., I/O initiation) 3. Special system instructions	1. Cache overflow attempt 2. Repeated chunk collision

and induce chunk squashes. Consequently, we need to be careful that chunks are still replayed deterministically.

Table 3 lists the exceptional events that might affect chunk construction during the initial execution. A full description of these events and the actions taken when they occur is presented in Montesinos et al.[11] At a high level, there are events that do not truncate the chunk, events that truncate it deterministically, and events that truncate it nondeterministically. The latter are the only ones that induce the logging of an entry in the CS log. Such events are the attempt to overflow the cache and repeated chunk collision. Overall, as described in Montesinos et al.,[11] even in the presence of all these types of exceptional events, DeLorean's replay is deterministic.

3.6. Evaluation

We used the SESC simulator[14] to evaluate DeLorean. We simulated a chip multiprocessor with eight cores clocked at 5 GHz. We ran the SPLASH-2 applications as well as SPECjbb2000 and SPECweb2005. In our evaluation, we estimated DeLorean's log size and its performance during recording and replay. In this section, we show a summary of the evaluation presented in Montesinos et al.[11]

Figure 9 shows the size of the PI and CS logs in *OrderOnly* in bits per kilo-instruction. We evaluate DeLorean 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. The SP2-G.M. bars correspond to the geometric mean of SPLASH-2.

The figure shows that our preferred 2,000-inst. *OrderOnly* configuration uses on average only 2.1b (or 1.3b if compressed) per kilo-instruction to store both the PI and CS logs. For comparison purposes, the estimated average size of the compressed Memory Races Log in RTR under Sequential Consistency (SC) from Xu et al.[17] is 8b per kilo-instruction. We call this system Basic RTR and use it as a reference, although we note that the set of applications measured here and in Xu et al.[17] are different. 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 9: Size of the PI and CS logs in *OrderOnly*. The numbers under the bars are the standard chunk sizes in instructions.

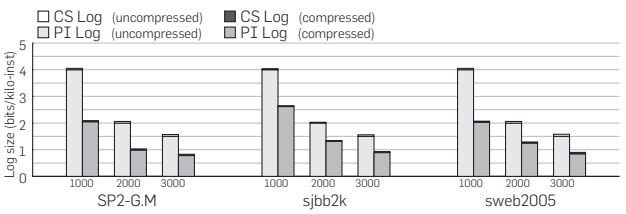


Figure 10: Size of the CS log in *PicoLog*. Recall that *PicoLog* has no PI log. The numbers under the bars are the standard chunk sizes in instructions.

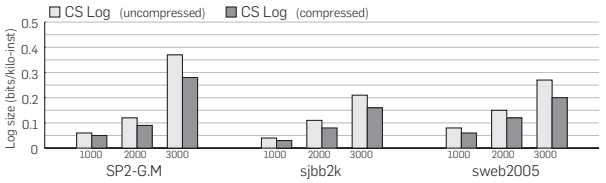


Figure 10 shows the size of the CS log in *PicoLog*. Recall that *PicoLog* has no PI log. We see that the CS log needs 0.37b or fewer per kilo-instruction in all cases—even without compression. Our preferred 1,000-instruction *PicoLog* configuration generates a compressed log with an average of only 0.05b per kilo-instruction. To put this in perspective, it implies that, if we assume an IPC of 1, the combined effect of all eight 5GHz processors is to produce a log of only about 20GB per day.

Finally, we consider the speed of DeLorean during recording and replay. It can be shown that *OrderOnly* introduces negligible overhead during recording, and that it enables replay, on average, at 82% of the recording speed. Under *PicoLog*, recording and replay speeds decrease, on average, to 86% and 72%, respectively, of the recording speed under *OrderOnly*.

4. CONCLUSION

This paper presented two novel hardware-based approaches for deterministic replay of multiprocessor executions, namely *Wisconsin Rerun* and *Illinois DeLorean*. Both approaches seek to enable deterministic replay by focusing on recording how long threads execute without interacting. Rerun makes few changes to standard multicore hardware, while DeLorean promises much smaller log sizes and higher replay speeds. Future work includes improving Rerun's replay speed, generalizing DeLorean's hardware design alternatives, and making the original multithreaded executions more deterministic.

Acknowledgments

We thank Norman Jouppi and David Patterson for suggesting this article and Norman Jouppi for writing the Perspective. Hower and Hill thank those acknowledged in the Rerun paper, including NSF grants CCR-0324878, CNS-0551401, and CNS-0720565. Hill has a significant financial interest in Sun Microsystems. Montesinos, Ceze, and Torrellas acknowledge the support provided by NSF under grants CCR-0325603 and CNS-0720593 and Intel and Microsoft for funding this work under the Universal Parallel Computing Research Center. C

References

1. Alameldeen, A.R., Mauer, C.J., Xu, M., Harper, P.J., Martin, M.M.K., Sorin, D.J., Hill, M.D., Wood, D.A. Evaluating non-deterministic multi-threaded commercial workloads. In *Proceedings of the 5th Workshop on Computer Architecture Evaluation Using Commercial Workloads* (February 2002), 30–38
2. Bacon, D.F., Goldstein, S.C. Hardware-assisted replay of multiprocessor programs. *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices* (1991), 194–206.
3. Ceze, L., Tuck, J.M., Montesinos, P., Torrellas, J. BulkSC: Bulk Enforcement of Sequential Consistency. In *Proceedings of the 34th International Symposium on Computer Architecture* (San Diego, CA, USA, June 2007).
4. Dunlap, G.W., Lucchetti, D., Chen, P.M., Fetterman, M. Execution replay on multiprocessor virtual machines. In *International Conference on Virtual Execution Environments (VEE)* (2008).
5. Hammond, L., Wong, V., Chen, M., Carlstrom, B.D., Davis, J.D., Hertzberg, B., Prabhu, M.K., Wijaya, H., Kozyrakis, C., Olukotun, K. Transactional memory coherence and consistency. In *Proceedings of the 34th International Symposium on Computer Architecture* (June 2004).
6. Hower, D.R., Hill, M.D. Rerun: Exploiting episodes for lightweight race recording. In *Proceedings of the 35th Annual International. Symposium on Computer Architecture* (June 2008).
7. Lamport, L. Time, clocks and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (July 1978), 558–565.
8. Leblanc, T.J., Mellor-Crummey, J.M. Debugging parallel programs with instant replay. *IEEE Trans. Comp. C-36*, 4 (April 1987), 471–482.
9. Lucia, B., Devietti, J., Strauss, K., Ceze, L. Atom-aid: Detecting and surviving atomicity violations. In *Proceedings of the 35th International Symposium on Computer Architecture* (June 2008).
10. Martin, M.M.K., Sorin, D.J., Beckmann, B.M., Marty, M.R., Xu, M., Alameldeen, A.R., Moore, K.E., Hill, M.D., Wood, D.A. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *Comp. Arch. News* (September 2005), 92–99.
11. Montesinos, P., Ceze, L., Torrellas, J. DeLorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In *Proceedings of the 35th International Symposium on Computer Architecture* (June 2008).
12. Narayanasamy, S., Pereira, C., Calder, B. Recording shared memory dependencies using strata. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, October 2006), 229–240.
13. Netzer, R.H.B. Optimal tracing and replay for debugging shared-memory parallel programs. In *Workshop on Parallel and Distributed Debugging* (San Diego, California, May 1993), 1–11.
14. Renau, J., Fraguela, B., Tuck, J., Liu, W., Prvulovic, M., Ceze, L., Sarangi, S., Sack, P., Strauss, K., Montesinos, P. SESC Simulator (January 2005), http://sesc.sourceforge.net.
15. Vallejo, E., Galluzzi, M., Cristal, A., Vallejo, F., Beivide, R., Stenstrom, P., Smith, J.E., Valero, M. Implementing kilo-instruction multiprocessors. In *Proceedings of the 2005 International Conference on Pervasive Systems* (July 2005).
16. Xu, M., Bodik, R., Hill, M.D. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Proceedings of the 30th Annual International Symposium on Computer Architecture* (June 2003), 122–133.
17. Xu, M., Bodik, R., Hill, M.D. A regulated transitive reduction (RTR) for longer memory race recording. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (October 2006), 49–60.
18. Xu, M., Malyugin, V., Sheldon, J., Venkitachalam, G., Weissman, B. Retrace: Collecting execution trace with virtual machine deterministic replay. In *Proceedings of the 3rd Annual Workshop on Modeling, Benchmarking and Simulation* (June 2007).

Derek R. Hower (drh5@cs.wisc.edu)
Computer Sciences Department
University of Wisconsin-Madison.

Mark D. Hill (markhill@cs.wisc.edu)
Computer Sciences Department
University of Wisconsin-Madison.

Pablo Montesinos (pmontesi@cs.uiuc.edu)
Computer Science Department
University of Illinois
Urbana-Champaign.

Josep Torrellas (torrellas@cs.uiuc.edu)
Computer Science Department
University of Illinois
at Urbana-Champaign.

Luis Ceze (luisceze@cs.washington.edu)
Department of Computer Science
and Engineering
University of Washington.