Introduction for joint CACM paper.

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

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

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

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

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

ABSTRACT

Something abstract.

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 due to the fact that commodity systems are now shared memory multiprocessors, which, in addition to the sources of nondeterminism in uniprocessors, are also impacted by the outcome of memory races among threads.

In an effort to help ease the pain of developing software in a nondeterministic environment, researchers have recently proposed adding *deterministic replay* capability 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 classes of 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 could, for example, be fed, in real time, the replay log of a primary server running on a physically seperate 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 [8]. 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, commodity systems are no longer uniprocessor machines, leading to the need for an efficient multiprocessor deterministic replay solution. When executing in a shared memory multiprocessor environment, memory races, which can potentially occur on every memory access, are another source of nondeterminism. Because of the frequent, low-latency nature of memory races, it is likely that hardware support will be needed.

In their pioneering work, Xu, et al. [7] proposed the first system with 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. logged only those races that cannot be implied through transitivity, i.e. those races not implied through the combination of a previously logged ordering an sequential program semantics. Figure 1a illustrates a transitive reduction. Inter-thread races between instructions accessing locations `A` and `B`, respectively, are implied by the recorded race for location `F`.

While both the original [5] and follow-on [6] work by Xu, et al. were successful in achieving efficient log compression (`~1byte/1000` instructions executed), they required a large amount of hardware state, on the order of an additional

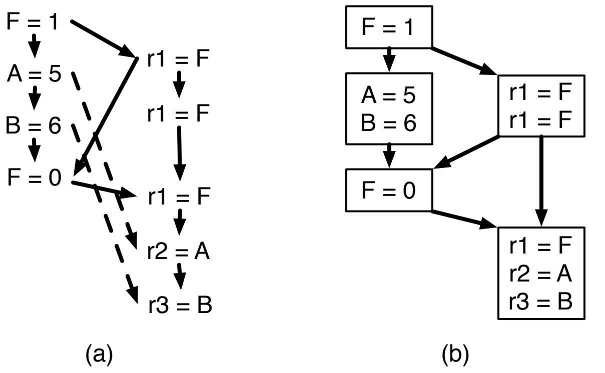

Figure 1: An example of efficient race recording using (a) an explicit transitive reduction and (b) indpendent regions. In (a), solid lines between threads are races written to the log, while dashed lines are those races implied through transitivity.

L1 cache per core, in order to do so. Subsequent work by Narayanasamy, et al. on the Strata race recorder reduced this hardware requirement but, as results in [2] 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 a 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 interthread 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 *DeLorean* and *Rerun*, that both exploit the same independence observation described above. These systems present different tradeoffs 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 but requires novel hardware in order to do so.

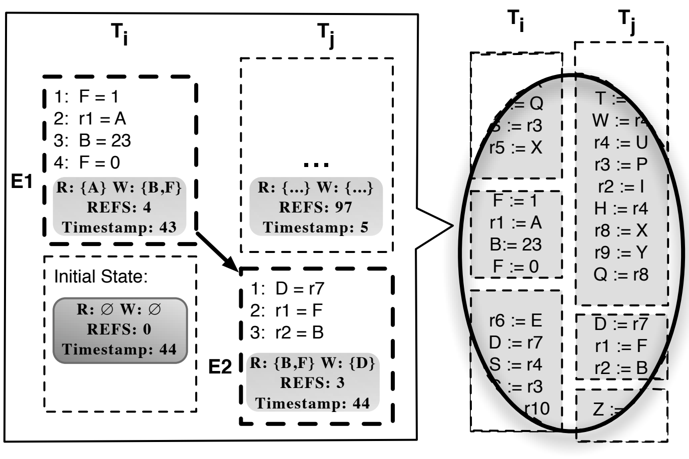

Figure 2: A example of episodic recording. Dashed lines indicate episode boundaries. In the blown up diagram of threads i and j, the shaded boxes show the state of the episode as it ends, including the read and write sets, memory reference counter, and the timestamp. The shaded box in the last episode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

2. RERUN

Rerun exploits the concept of *episodic race recording* to acheive 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 avoids artifically 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.

2.1.1 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 (or thread k). 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 appear atomic, 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 the 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 threads j accesses the variable F that was previously written (i.e. F is in the write set of E_1).

Second, Rerun must order episodes across threads such that episode *causal dependencies* are respected. An active episode E is causally dependent on an episode F from another thread if E either (a) reads a variable written in F or (b) writes a variable read or written in F. Rerun records episode dependencies using Lamport scalar clocks [3]. This technique guarantees that 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.

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.

2.1.2 Key Definitions

Let an *episode E* be a contiguous sequence of dynamic instructions executed by one thread without conflicting with other threads via memory. Let $thread_E$ denote the thread executing E, $references_E$ denote the number of dynamic memory references in E, and $timestamp_E$ denote the Lamport scalar clock associated with E. Finally, let R_E denote the memory blocks read in E (its *read set*) and W_E denote the memory blocks written in E (its *write set*).

2.1.3 Establishing Epsiodes

A new episode begins whenever a thread begins execution. When a thread starts a new episode E, it resets $references_E$ to zero and empties the read and write sets, R_E and W_E. As the thread executes dynamic memory references, it increments $references_E$ and adds memory block addresses to R_E and W_E as appropriate. Whenever a thread terminates an episode, it writes $references_E$ into a per-thread log, immediately begins a new episode, and repeats. Thus, a thead's execution will be logged as a series of episodes *without affecting the execution.*

$$[W_E \cap (R_E \cup W_F) = \emptyset] \wedge [R_E \cap W_F = \emptyset] \qquad (1)$$

Importantly, while a thread *must* end an episode for conflicts, Rerun *may* end an episode early for any or no reason, since any subset of an atomic region in an execution is itself atomic (and, unlike transactional memory, programmers do not specify what should be atomic). In Section 2.2, we will ease implementation cost by ending some episodes early.

2.1.4 Ordering Epsiodes

Episodes must be correctly ordered to enable a faithful deterministic replay. Rerun's ordering mechanism meets three conditions sufficient for a Lamport Scalar Clock [3] implementation:

1. When an episode 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 $D_1 ... D_N$, it sets its $timestamp_E$ to $maximum[timestamp_E, timestamp_{D_0} + 1, timestamp_{D_1} + 1, ..., timestamp_{D_N} + 1]$.

Table 1: Base System Configuration

Cores	16, in-order, 3GHz
L1 Caches	Split I&D, Private, 32K 4-way set associative, write-back, 64B lines, LRU replacement, 3 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)

Finally, when each episode E ends, Rerun logs its $timestamp_E$, along with $references_E$, in a per-thread log. 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.

2.2 Rerun Implementation

Here we develop a Rerun implementation for a system based on a cache-coherent multicore chip.

2.2.1 Base System

Our base system is a multicore chip together with DRAM and I/O. Figure 3 displays the multicore chip, while Table 1 gives key parameters. These include private L1 write-back caches, a shared L2 cache, an MESI directory protocol, and a sequentially consistent memory model. We discuss varying these assumptions in Section 2.2.4.

2.2.2 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., 32 and 128 bytes, respectively),
- A Timestamp Register, TS, to hold the Lamport Clock of the current episode executing on the core (e.g., four bytes), and
- A Memory Reference Counter, $REFS$, to record the current episode's references (e.g., two bytes).

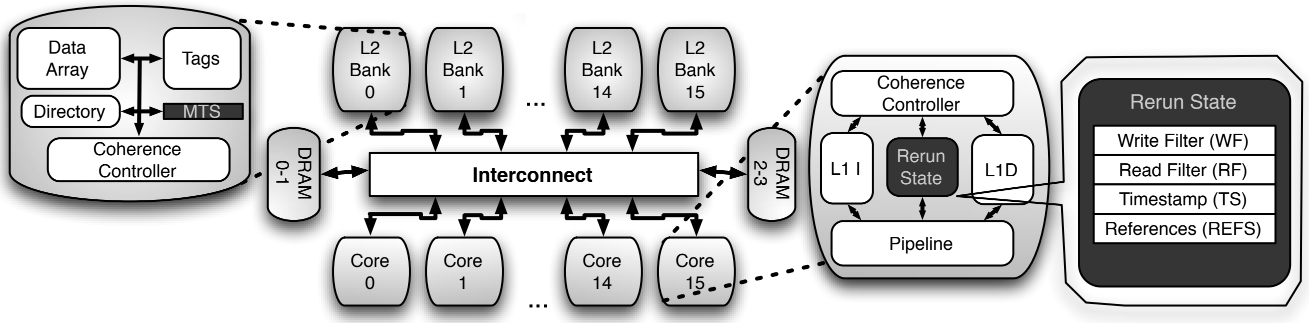

Figure 3: Rerun Hardware

To each L2 cache bank, Rerun also adds a "memory" timestamp register, MTS (e.g., four bytes). 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 vicitimization.

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

2.2.3 *Rerun Operation*

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 and writes its $REFS$ 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 two bytes.

2.2.4 *Extensibility*

Though we have described an implementation of Rerun in terms of a specific base system, Rerun can be applied to other systems, including systems 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 [2].

2.3 Evaluation

2.3.1 *Methods*

We evaluate the Rerun recording system using the Wisconsin GEMS [4] 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].

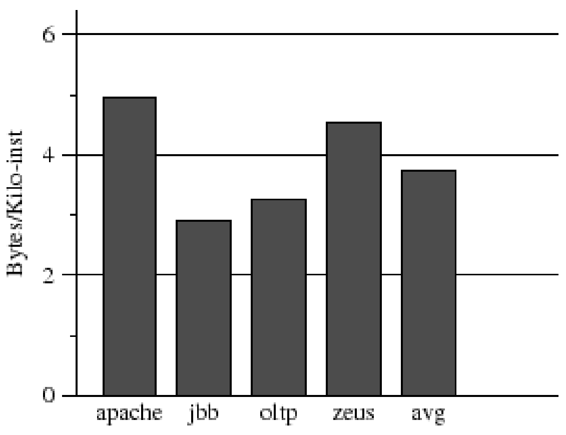

Figure 4: Rerun Absolute Performance

2.3.2 *Rerun Performance*

Figure 4 shows the performance of Rerun on all four commercial workloads. Rerun achieves an uncompressed log size of about 4 bytes logged per 1000 instructions. Importantly, we notice little 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 [6]), but does so with a fraction of the hardware cost (`~0.2`KB per core vs. 24KB per core). Like RTR, and unlike Strata, 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.

3. CONCLUSION

Rerun uses episodes to efficiently record the outcome of memory races, acheiving a log size comparable to the best prior state of the art while needing a small amount of hardware state (166 bytes/core). Rerun can be implemented with only minimal changes to a conventional memory system, and can generate a race log without perturbing the

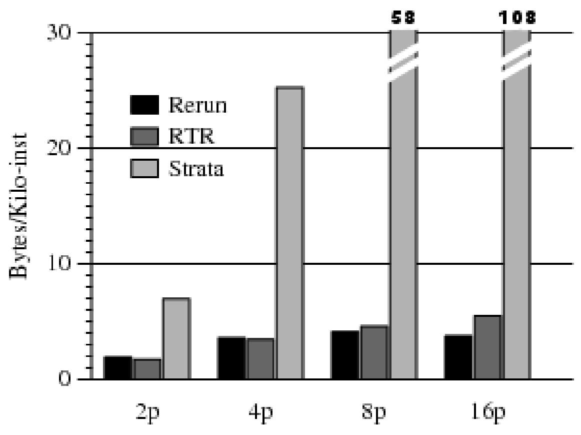

Figure 5: Comparison to RTR and Strata

execution under observation.

4. REFERENCES

[1] A. R. Alameldeen, C. J. Mauer, M. Xu, P. J. Harper, M. M. K. Martin, D. J. Sorin, M. D. Hill, and D. A. Wood. Evaluating non-deterministic multi-threaded commercial workloads. In *Proc. of the 5th Workshop on Computer Architecture Evaluation Using Commercial Workloads.*

[2] D. R. Hower and M. D. Hill. Rerun: Exploiting episodes for lightweight race recording. In *Proc. of the 35th Annual Intnl. Symp. on Computer Architecture.*

[3] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[4] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *Computer Architecture News*, pages 92–99, Sept. 2005.

[5] M. Xu, R. Bodik, and M. D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Proc. of the 30th Annual Intnl. Symp. on Computer Architecture.*

[6] M. Xu, R. Bodik, and M. D. Hill. A regulated transitive reduction (rtr) for longer memory race recording. In *Proc. of the 12th Intnl. Conf. on Architectural Support for Programming Languages and Operating Systems.*

[7] M. Xu, R. Bodik, and M. D. Hill. A hardware memory race recorder for deterministic replay. *IEEE Micro*, 27(1), Jan/Feb 2007.

[8] M. Xu, V. Malyugin, J. Sheldon, G. Venkitachalam, and B. Weissman. Retrace: Collecting execution trace with virtual machine deterministic replay. In *Proceedings of the 3rd Annual Workshop on Modeling, Benchmarking and Simulation*, June 2007.