Appears in the International Symposium on Computer Architecture (ISCA), June 2008

Rerun: Exploiting Episodes for Lightweight Memory Race Recording

Derek R. Hower Mark D. Hill

Computer Sciences Department
University of Wisconsin-Madison
http://www.cs.wisc.edu/multifacet
{drh5,markhill}@cs.wisc.edu

Abstract

Multiprocessor deterministic replay has many potential uses in the era of multicore computing, including enhanced debugging, fault tolerance, and intrusion detection. While sources of nondeterminism in a uniprocessor can be recorded efficiently in software, it seems likely that hardware support will be needed in a multiprocessor environment where the outcome of memory races must also be recorded.

We develop a memory race recording mechanism, called ***Rerun****, that uses small hardware state (~166 bytes/core), writes a small race log (~4 bytes/kilo-instruction), and operates well as the number of cores per system scales (e.g., to 16 cores). Rerun exploits the dual of conventional wisdom in race recording:* ***Rather than record information about individual memory accesses that conflict, we record how long a thread executes without conflicting with other threads.*** *In particular, Rerun passively creates atomic* episodes*. Each episode is a dynamic instruction sequence that a thread happens to execute without interacting with other threads. Rerun uses Lamport Clocks to order episodes and enable replay of an equivalent execution.*

1. Introduction

A system exhibiting *deterministic replay* capability can record sufficient information during an execution to enable a replayer to (later) create an equivalent execution despite inherent sources of nondeterminism that exist in modern computer systems. The information required includes initial state (e.g., a checkpoint) and nondeterministic events [39]. Recording a uniprocessor execution is viable in software (e.g., hypervisor), because the sources of nondeterminism, such as I/O, DMA fills, and interrupts, are relatively rare events. Deterministic replay of a uniprocessor machine has already proven useful for debugging [41] and intrusion detection [7] applications.

Most future systems, however, will use multicore chips that provide software with a shared memory model. This model adds *memory races*—conflicting accesses to both synchronization and data variables—as an additional source of nondeterminism to be recorded. Unfortunately, memory races have the potential to occur on almost every memory reference, making efficient analysis difficult for software.

Fortunately, recent work has proposed hardware support for multithreaded deterministic replay, in general, and, memory race recording, in particular [25, 26, 39, 40]. These systems log the outcome of memory races as they occur. To keep the storage and bandwidth needs reasonable, these systems only record a subset of all races that cannot be implied transitively, i.e. races that are not implied through the combination of a previously recorded dependence and program order semantics. The *Flight Data Recorder (FDR)*, both original [39] and enhanced [40], however, uses substantial hardware state to perform this reduction (e.g., 4-24KB per core). Hardware vendors would like to see this state reduced, in part, because it is cost paid even when recording is disabled. *Strata* [25] reduces this state, performs well for four-core systems, but, suffers a substantial increase in per-core log sizes as the number of cores per system grows (Section 5).

We advance the state of the art by proposing a new memory race recorder, called *Rerun*, that achieves scalable race log sizes on par with prior work at only a fraction of the hardware state. While races are typically described in terms of conflicts that occur between individual memory accesses, Rerun records the dual of this information: *how long a thread executes without conflicting with any other thread in the system.*

Rerun uses atomic *episodes* as the fundamental unit of ordering. An episode is a series of dynamic instructions from a single thread that happen to execute without conflicting with any other thread in the system. Episodes are created passively by observing system behavior without altering the normal execution flow. When recording is enabled, the entire system execution

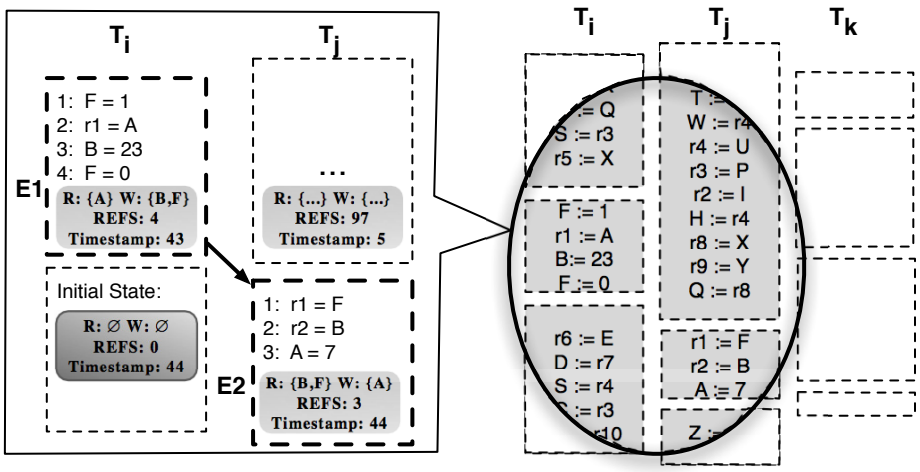

Figure 1: Example of episodic recording. Dotted lines indicate episode boundaries created during execution. In the blown up diagram of threads *i* and *j*, the shaded boxes show the state of the episode as it ends, including the read and write sets, memory reference counter, and the timestamp. The darker shaded box in the last episode of thread *i* shows what the episode state is initialized to when an episode begins.

is viewed as a collection of ordered episodes so that every dynamic instruction logically resides within the boundaries of an episode.

Rerun records the outcome of an execution by simply logging the length and order of episodes. A causal ordering among episodes from different threads is established using Lamport Scalar Clocks [15], which is a standard mechanism from distributed systems used to create a global notion of logical time in systems where no single point of ordering exists. Using this mechanism, Rerun associates each episode with a timestamp that correctly places the episode in a partial order of execution and preserves inter-thread dependencies.

We implement Rerun by augmenting cores in the system with a small amount of state (~166 bytes/thread) and by piggybacking on an existing coherence protocol. The coherence protocol allows Rerun to ensure that no two concurrently active episodes conflict with one another and provides a substrate for keeping logical time consistent.

With respect to prior memory race recorders, we show that Rerun is simultaneously comparable to (a) the recorder with the smallest hardware state (like Strata with snooping) and (b) the recorder with the slowest log growth rate (like enhanced FDR [40]).

The rest of the paper is organized as follows. Section 2 presents the key ideas for Rerun. Section 3 discusses our implementation in a base system and then elaborates on how Rerun can be extended to alternate architectures. In Section 4 we explain our evaluation methods and in Section 5 we present empirical results. We discuss related work and conclude in Sections 6 and 7, respectively.

2. Episodic Memory Race Recording

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

2.1. Motivating Example and Key Ideas

Consider the execution in Figure 1 that highlights two threads i and j executing on a multiprocessor 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 1).

Rerun must solve two sub-problems in order to ensure that enough episodic information is recorded to enable a deterministic replay of all memory races. First, it must determine when an episode ends and, by extension, when the next one begins (Section 2.3). 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 they reference the same memory block, are

from different threads, and at least one is a write. For example, episode E_1 in Figure 1 ends because thread 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 (Section 2.4). 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. 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 that is 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 (Section 3.5). If episodes are replayed in timestamp order, then the replayed execution will be logically equivalent to the recorded execution.

2.2. 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*).

Two episodes E and F from different threads avoid conflicts when the write set of one has no blocks in common with the union of the read and write sets of the other and vice versa. Formally, the *full no-conflict condition* is:

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

2.3. Establishing Episodes

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 blocks 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 thread's execution will be logged as a series of episodes *without affecting the execution*.

Episode E must end if, for any episode F from another thread, the following *half-no-conflict condition* may become false at $thread_E$:

$$[W_E \cap (R_F \cup W_F) = \varnothing] \wedge [R_E \cap W_F = \varnothing] \quad \text{(EQ 2)}$$

By symmetry, $thread_F$'s logic will check:

$$[W_F \cap (R_E \cup W_E) = \varnothing] \wedge [R_F \cap W_E = \varnothing] \quad \text{(EQ 3)}$$

Together, these checks ensure that one thread or the other will always end its episode before the full non-conflict condition (EQ 1) becomes false. This ensures that concurrently executing episodes never conflict.

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 3, we will ease implementation cost by ending some episodes early.

2.4. Ordering Episodes

Episodes must be correctly ordered to enable a faithful deterministic replay. Rerun's ordering mechanism meets three conditions sufficient for a Lamport Scalar Clock [15] 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_{D0} of completed episode D_0 or in the read-set of any episode D_1 ... D_N, it sets its $timestamp_E$ to maximum[$timestamp_E$, $timestamp_{D0}$+1, $timestamp_{D1}$+1, ..., $timestamp_{DN}$+1].

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 without effecting replay fidelity.

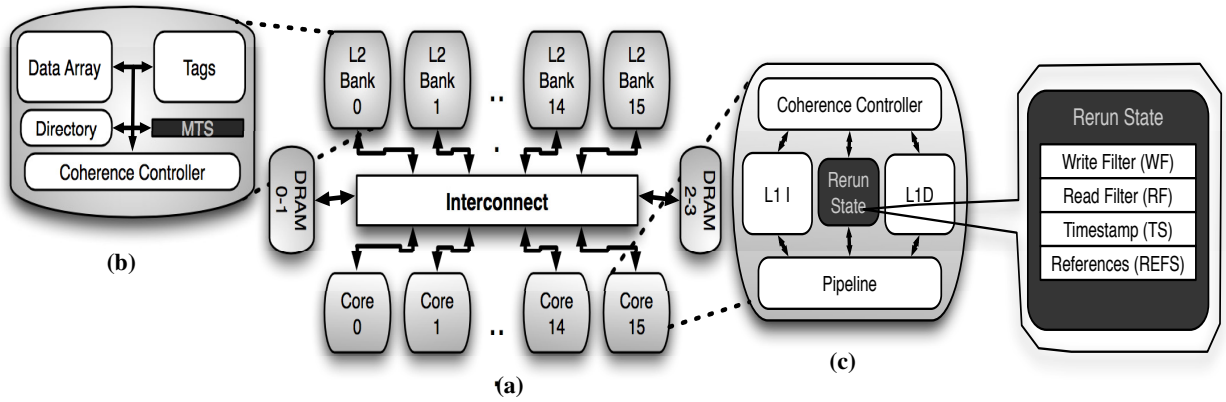

Figure 2: (a) Base system configuration (b) Rerun hardware added to each L2 bank (c) Rerun hardware added to each core

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 Cache	Unified, Shared, Inclusive, 8M 8-way set associative, write-back, 16 banks, LRU replacement, 37 cycle hit
Directory	Full bit vector in the L2
Memory	4G DRAM, 300 cycle access
Coherence	MESI Directory, silent replacements
Consistency Model	Sequential Consistency (SC)

3. Rerun Implementation

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

3.1. Base System

Our base system is a multicore chip together with DRAM and I/O. Figure 2 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 3.6.

3.2. Rerun Hardware

As Figure 2 depicts, Rerun adds modest hardware state to the base system. To each core, Rerun adds:

- Write and Read Bloom filters, `WF` and `RF`, to track the current episode's write and read sets (e.g., 32 and 128 bytes, respectively (see Section 5.1)),
- 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).

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 victimization.

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

3.3. Rerun Operation

Rerun implements episodic race recording (Section 2) via the algorithm of Figure 3. When a core writes (reads) a block *b*, it increments `REFS` and logically adds the address of *b* to its `WF` (`RF`), perhaps redundantly. When a core receives data on a miss, the core also sets its `TS` to the maximum of `TS` and one more than the value of the incoming timestamp (Section 2.4 Rules 2 and 3) Neither of the above actions cause Rerun to end the current episode.

Rerun detects conflicts among episodes when the base system's coherence protocol either directs a request to this core for a block in its `WF` or an exclusive request in its `RF` (half-no-conflict condition (EQ 2) violated). These Bloom filters may detect a true conflict (episode must end) or false conflict (episode may end early). In either case, Rerun ends the episode by logging the episode's `REFS` and `TS`. By recording the end of every episode with its associated timestamp, a complete partial order of episode execution is preserved in the log. The system then prepares for a new episode by clearing `WF`, `RF`, and `REFS`, and incrementing `TS` (Section 2.4 Rule 1). Regardless of whether a coherence request ends an episode, Rerun always appends its current `TS` on the coherence response message so that a causal ordering can be established among interacting episodes.

Rerun might miss a conflict if a block in a thread's read or write set was victimized from its private cache because, in the case of a directory protocol, coherence requests may no longer be forwarded to the core. To guard against this, Rerun ends an episode whenever a block *b* in WF or RF is victimized. When notifying the L2 of the vicitimization, a core also appends its current TS to the coherence message sent to block *b*'s bank. The bank then sets its MTS to the maximum of its current value and the incoming TS. Future responses for any block from this bank will carry a timestamp greater than or equal to the TS of the thread that victimized *b*.

Rerun allows programmers to view logs as *per thread*, rather than *per core*, by gracefully handling virtualization events. 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.

3.4. Rerun Discussion

Some comments on Rerun operation are in order. First, Rerun's passing of timestamps in coherence messages may seem complex. Nevertheless, it is a classic message-passing implementation of Lamport Clocks that uses coherence messages in a manner similar to FDR. Rerun, however, carries a globally-meaningful Lamport Clock in messages, while FDR's messages carry a per-core instruction count.

Second, one might expect that Rerun's log size could be much larger than FDR, for example, because Rerun does not perform an explicit transitive reduction. To the contrary, we will show Reruns logs are comparable to FDR. To see why this might be the case, let us return to the example in Figure 1 of Section 2. Here, FDR would note conflicts on memory blocks A, B, and F, but only log one 8-byte entry for block F, since this entry implies the others by transitivity. In this case, Rerun also makes a single log entry (when episode E_1 ends) but does so with two fewer bytes, leading to a potentially slower log growth.

Third, one might be concerned that Rerun's conservative policy of ending an episode when a block in WF or RF is victimized might inordinately shorten episode duration. We have found this fear unjustified, perhaps because recently-referenced blocks are less likely to be replaced by most cache replacement algorithms. Similarly, ending all episodes on a paging or other virtualization event has a small impact on log size due to the rarity of these events.

State per Core
REFS: Dynamic Memory Reference Count
TS: Timestamp
WF: Write Filter
RF: Read Filter
LOG: Race Log

State per Shared Cache Bank
MTS: Timestamp

Core Actions

```
On commit of a memory operation memop
        REFS++
        if( memop is a store )
                WF.insert(memop.address)
        if( memop is a load )
                RF.insert(memop.address)

On receiving data with timestamp t
        TS = MAX( t+1, TS )

On sending data/ack for block b
        SEND( b, TS )
        if( WF.find( b.address ) ||
            (RF.find( b.address ) &&
             request == GETX) )
                LOG.append( REFS, TS )
                WF.clear()
                RF.clear()
                REFS = 0
                TS++

Before replacing block b
        SEND( b, TS )
        if( WF.find( b.address ) ||
            RF.find( b.address ) )
                LOG.append( REFS, TS )
                WF.clear()
                RF.clear()
                REFS = 0
                TS++
```

Shared Cache Bank Actions

```
On sending a response for block b
        SEND( b, MTS )
On receiving data with timestamp t
        MTS = MAX( t, MTS )
```

Figure 3: Rerun Algorithm

3.5. Rerun Replay

For testing Rerun, we developed a stand-alone single-threaded software replayer. This replayer replays episodes in Lamport scalar clock order. It scans the logs of all threads, picks an episode with the smallest timestamp, executes it, and repeats until no episodes remain. While the replay process is mostly sequential, there is a limited opportunity for parallelism when replaying episodes from different threads that have the same recorded timestamp.

Many applications of deterministic replay may not need a fast replayer (e.g., debugging). However, replay speed may be critical in other proposed applications

such as replay-based fault tolerance [3, 19, 28, 29]. Future work will examine techniques for faster replay, possibly including an alternative timebase during recording, preprocessing logs to sort episodes, or use of hardware acceleration.

3.6. Extensibility

Though we have described an implementation of Rerun in terms of a specific base system, Rerun can be applied to other systems. We briefly discuss some variants here and note any changes that may be needed over the base implementation.

Memory Consistency Model. Until this point, the Rerun system we have described requires sequential consistency (SC). Thus, Rerun works with both simple and aggressive implementations of SC [6, 9, 10, 13, 34, 37], including SC implementations of relaxed models.

We have designed, but not implemented, an add-on to Rerun that extends episodic recording to the TSO consistency model [36]. The extended Rerun design separately logs load operations that could result in a violation of sequential consistency. We add logic to the system that detects potentially SC-violating loads using techniques developed by Xu et al. [40]. When troubling loads are detected, the extended Rerun will either log the value that the load returned (as was done by Xu, et al.), or log a timestamp that corresponds to the correct value the load received, depending on whichever is easiest to implement. In the latter case, the recorded timestamp will be equal to the timestamp of the thread when the load returned a value. A replayer can use the extra timestamp to reconstruct the value of the load by monitoring that location in the replayed memory image.

Future work may include extending Rerun to the x86 memory consistency model [12] (not much weaker than TSO), as well as more relaxed models.

Out-of-Order and/or Multithreaded Cores. It is straightforward to adapt Rerun for out-of-order cores by associating Rerun logic with commit logic, in part, because Rerun activities never cause mis-speculation.

Rerun also adapts well for systems with hardware multithreading. In such a system, each hardware thread context would require its own `WF`, `RF`, and `REFS`. Replicating this state costs less than for other recorders because Rerun's state is small (166 bytes). The major algorithmic difference over our base system is that each read or write by a thread context has to check filters for other threads contexts on same core to detect conflicts that might not cause coherence events.

Cache Design. Our base system uses a write-back private L1 cache at each processing node. If instead a write-through L1 cache were used, Rerun could still use a private write-back L2 to maintain logging efficiency. More work is needed to make Rerun, as well as FDR and Strata, efficient in the absence of private write-back caches.

There is a great deal of flexibility in the design of other levels of the memory hierarchy. Rerun only requires that the level of shared memory closest to the core (e.g. the shared L2 in our base system) has an MTS register to maintain a consistent ordering of evicted blocks. Multiple banks of shared memory, so long as the addressable content of each bank is disjoint, work well under Rerun without additional modification.

Snooping Coherence. Replacing directory-based coherence with bus-based snooping coherence alters Rerun in two ways. First, bus-based snooping does not provide invalidation acknowledgement messages that can carry timestamps, as is required in the base Rerun design. Xu [38] sketches a solution for FDR with snooping that can be adapted for Rerun. The key idea is to piggyback timestamps on a subset of request messages rather than acknowledgements at a cost of bandwidth overhead. Systems that implement snooping protocols over a point-to-point network would likely have the ability to send data on acknowledgements, and as such would work with an unmodified Rerun algorithm. Second, snooping coherence protocols broadcast all requests. This eliminates the need to end episodes on block evictions.

Hardware Transactional Memory. Rerun has the potential to interact favorably with hardware transactional memory [16]. For example, many flat and closed-nested transactions can be recorded as a single episode corresponding to the outermost transaction, as done for Atlas [14]. Future work can examine how Rerun interacts with transactional memory subtleties, such as open-nested transactions and compensating actions [22, 24].

4. Evaluation Methods

We evaluate the Rerun recording system using the Wisconsin GEMS [20] full system simulation infrastructure, which models an enterprise-level SPARC server running an unmodified Solaris 10 operating system. The simulator configuration matches that of the baseline shown in Table 1 with the addition of Rerun hardware support. All experiments were run using the Wisconsin Commercial Workload suite [1], which consists of a task-parallel web server (apache), a pipelined web server (zeus), a java middleware application (jbb), and a TPC-C-like online transaction processing (oltp) workload on DB2. Since results from alternative workloads are similar, we present most data

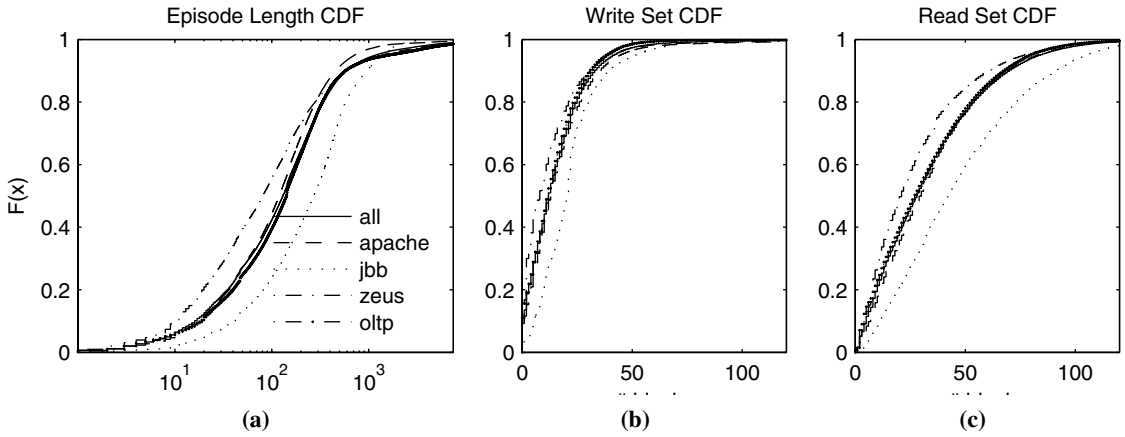

Figure 4: Distribution of episode lengths for all workloads.

for an average workload computed as if each benchmark ran for an equal number of instructions.

For comparison purposes, we also model two hardware race recording systems previously proposed, namely the Flight Data Recorder and Strata, on the same system configuration. When it is important to distinguish, we name the original FDR of 2003 [39] *FDR-1* and the enhanced version of 2006 [40] *FDR-2*. Our performance studies use FDR-2, which artificially creates stricter dependencies to reduce the log size. We use Xu et al.'s FDR-2 code and so have a high confidence in our comparisons.

We reimplemented Strata with directories (Strata-Dir) [25] so that it can be used in our simulation infrastructure. Because we do not have the same workloads used in the original evaluation of Strata, we can not validate the correctness of our implementation by a direct comparison to previously published results. However, we can indirectly validate our implementation using results published for Strata-Dir, FDR-1, and FDR-2.

With four cores, Strata asserts a log size ~6x smaller than FDR-1, while FDR-2 claims to be ~25x smaller than FDR-1. Thus, to a first order, we would expect Strata-Dir's log to be ~4x (25x/6x) larger than FDR-2's. Results (not shown) confirm the ratio approximately holds. Results presented in Section 5 show a ~6x increase due, in part, to the smaller cache sizes we assume. Finally, we use the Strata paper's observation that Strata with snooping (Strata-Snoop) and directories (Strata-Dir) create logs of similar size.

5. Experimental Results

We next refine Rerun by selecting good implementation parameters. Then we analyze Rerun's sensitivity to private cache size. Finally, Section 5.3 we compare Rerun's overall performance with the state of the art: the enhanced FDR-2 [40] and Strata-Dir [25].

5.1. Refining Rerun

An effective Rerun implementation must determine appropriate sizes of `REFS`, `WF`, and `RF`.

Counting References (`REFS`). When a episode ends, Rerun logs the final value of `REFS` to record the number of memory references in the episode. A tradeoff exists in the selection of `REFS`'s size: a large size (e.g., four bytes) lets episodes grow to their natural length resulting in less frequent logging, while a small size (e.g., two bytes) reduces the size of an individual log entry but requires extra logging when `REFS` reaches its maximum value (e.g., 64K). An optimal balance that achieves an efficient log size is dictated by the distribution of episode lengths.

To determine how large episodes naturally grow for our workloads, we ran experiments using infinite caches, perfect (i.e. no false positives) filters, and an unbounded `REFS` counter. Figure 4a shows the cumulative distribution function (CDF) of episode lengths for all four workloads, measured in the number of dynamic memory references. Over 99.9% of all episodes are less than 45,000 memory references long. Thus, assuming that the `REFS` structure is limited to byte boundaries, a 16-bit counter will suffice to capture nearly all episodes without adding unnecessary bits to the log.

We validated that the 16-bit `REFS` strikes the best balance between log entry size and log frequency for our workloads by running experiments with perfect filters and 8-, 16-, and 32-bit counters. Results (not shown) revealed that the 16-bit `REFS` counter did indeed produce the smallest log.

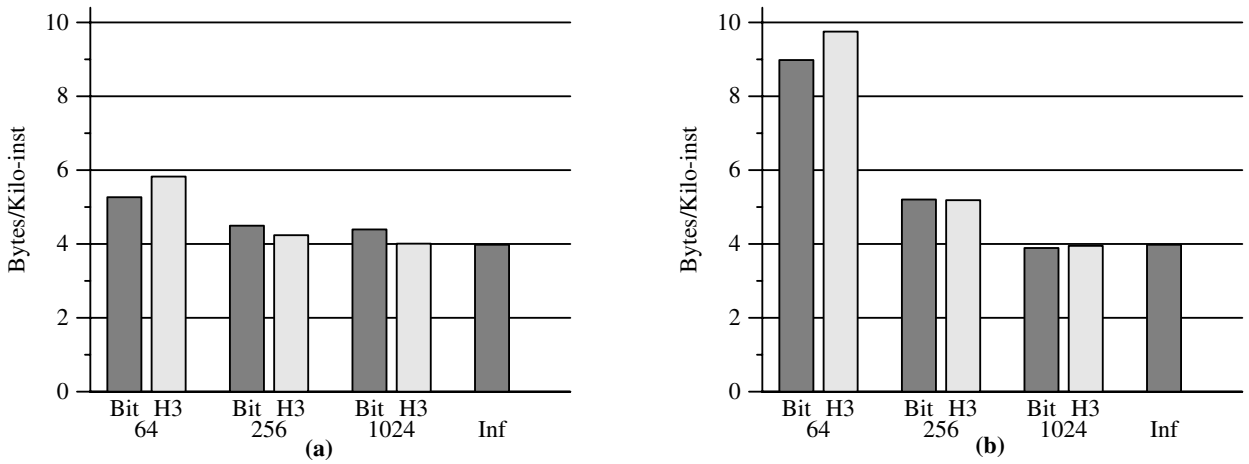

Figure 5: Write (a) and Read (b) filter sensitivity analysis for Bloom filters of varying size and hashing function. Sizes are presented in bits. Hashing functions are *Bit Selection* (Bit) and H_3 (H3) with four hash functions per filter. Log sizes are in bytes/1000-instructions (without compression).

Write and Read Filters (`WF`/`RF`). Write/read filters with a high rate of false positives could degrade the logging performance of Rerun. Figures 4b and 4c show the cumulative distribution functions for episode write and read set sizes, which have a direct impact on false positive rate. The 99th percentile of the distributions are 70 blocks for the write set and 113 blocks for the read set.

The amount of aliasing incurred by the WF and RF Bloom filters is influenced by three design dimensions: the size of the bit field, the number of hashing functions, and the type of hashing function. We empirically examine `WF`/`RF` alternatives by varying filter size from 64 to 1024 bits and the number of hash functions from one to four. We consider two different hashing functions, namely *bit selection*, where the hash comes from a simple subset of the block address, and H_3, which is a near-universal hash function that produces more uniformly distributed hash values [4, 33].

We ran experiments varying all three Bloom filter design dimensions to determine the impact on logging performance, though due to space constraints, only show results for filters using four hash functions in Figure 5. Results for filters with less than four hash functions performed equal to or worse than their four-hash counterparts in all cases. Filters of length 1024 bits with four different hash functions perform on par with a perfect (infinite size) filter for both the read and write sets. However, for the write set, filters with only 256 bits and either two or four hash functions also perform nearly identically to a perfect filter. These results are consistent with Sanchez et al. [35], whose analytic equations predict that Bloom filter sizes should be much larger than the mean read/write sets avoid significant aliasing.

Our results in Section 5.3 will assume that Rerun implements write and read filters using four H_3 hash functions, where each `WF` is 256 bits (32 bytes) and each `RF` is 1024 bits (128 bytes).

5.2. Sensitivity to Private Cache Size

Rerun is designed for systems where each core has a private L1 and may have a private L2 cache. Designers select cache sizes based on many considerations. Ultimately, this cache size choice will affect Rerun's logging performance. Smaller caches can lead to more frequent evictions, any one of which could prematurely end an episode if the address of the evicted block hits in a write/read filter. To evaluate the impact of the private cache size on logging performance, we performed a sensitivity study on a system with perfect filters and a 16-bit `REFS` counter. For this study, we model a private L1/L2 hierarchy as an artificially-large private L1 cache. As Figure 6 shows, log sizes increase for private caches smaller than our base case of 32K-bytes and decrease for larger caches.

Importantly, Rerun log size is acceptable across all cache sizes studied. Thus, Rerun will operate robustly with cache sizes 8K-bytes and larger. We also note that, on average, log sizes improve until 256K-bytes and then plateau. This size presumably represents the point when all episodes grow large enough so that they often end due to conflicts with other threads. We attribute the occasional increase in log size from a smaller to a larger cache capacity in our results (e.g. zeus 256K to 512K) to simulation variance, as the observed differences are within a 5% margin of error.

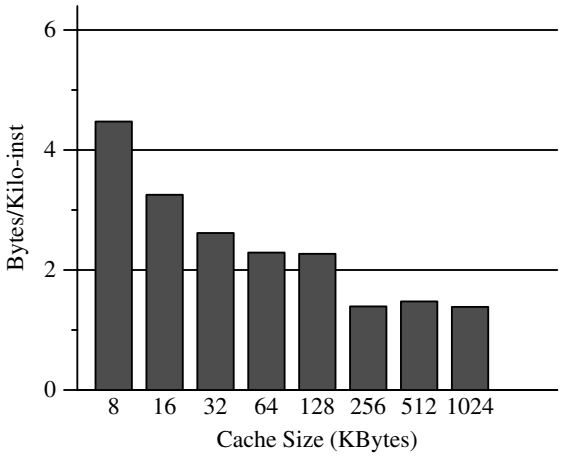

Figure 6: Private cache size sensitivity.

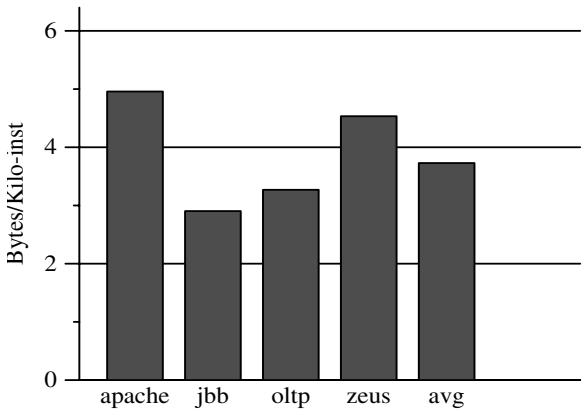

Figure 7: Refined Rerun log size.

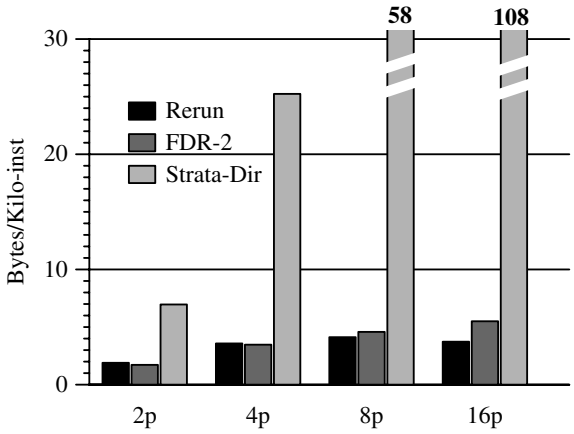

Figure 8: System scalability comparison.

5.3. Performance Analysis

Refined Rerun. Using empirical data from Section 5.1, we select the following parameters for an effective Rerun implementation: 1024-bit H_3 read filter with four hash functions, 256-bit H_3 write filter with four hash functions, a 32-bit timestamp, and a 16-bit reference counter. In total, we add 166 bytes/core of state to the base system (not including a small amount for the log bookkeeping).

Rerun Logging Performance. As shown in Figure 7, Rerun achieves a log growth rate just under four bytes/1000 instructions on average for our workloads on the base 16-core system (see Table 1). Results for other workloads will differ. In particular, workloads with more-frequent sharing will log more bytes per instruction. Because there is not a notable difference among the workloads, we average results for further performance analysis.

Rerun vs. FDR-2 and Strata-Dir. Figure 8 shows the log growth rate of Rerun, FDR-2, and Strata, as the number of cores varies from 2 to 16. (All logs are uncompressed.) In the four core system previously evaluated by FDR and Strata, we see that Rerun produces a log of comparable size to FDR-2 and significantly less than Strata. Importantly, Rerun achieves a log size on par with FDR with a small fraction of the hardware cost.

Scaling the Number of Cores. Figure 8 also shows how the log performance of Rerun, FDR-2, and Strata-Dir changes as the number of cores scales from two to 16 (the current limit of our workloads). Both Rerun and FDR-2 scale well since their techniques exploit local interactions among cores and create log entries sized independently of the number of cores. Nevertheless, precise values can go up or down, depending how workload behavior changes with the number of cores (e.g, we omit zeus from the 16-core FDR-2 average, since its behavior is anomalous). Strata-Dir, however, scales less well, because any interaction among a pair of cores logs a stratum whose size is proportional to the number of cores (see Section 6 for a sketch of the Strata algorithm).

Hardware. Table 2 gives an overview of the per-core hardware requirements for Rerun and several versions of previously proposed systems. State overheads are calculated based on an implementation extending the system in Table 1. Importantly, Strata-Snoop requires a broadcast snooping protocol instead of a directory.

Figure 9 shows how the hardware overheads scale in large systems. Strata-Dir results assume each doubling of the number of cores adds 2MB to L2 cache size. Results show that Rerun and Strata-Snoop require much less state than Strata-Dir, FDR-1, and FDR-2. Thus, Strata-Snoop may be preferred for near-term multicore chips using snooping. Recalling log

Table 2: Hardware added to the base system for each recorder

System	Added Structures	State/ Core (bytes)
Strata - Snoop [25]	**Per Core:** 1 dependence bit / cache line Dynamic Mem. Instruction Counter Eviction Bloom filter	190
Strata - Dir [25]	**Per Core:** 1 dependence bit / cache line Dynamic Mem. Instruction Counter **Per Directory:** 1 dependence bit / directory entry Memory IC vector	1.25K
FDR-1 [39]	**Per Core:** 1 timestamp / cache line Dynamic instruction counter Vector of instruction counts	4K
FDR-2 [40]	**Per Core:** Timestamp Memory, Dynamic instruction counter Vector of instruction counts Sliding Window	24K
Rerun (this paper)	**Per Core:** Dynamic Mem. Instruction Counter Timestamp Register Read/Write Bloom Filters **Per L2 Bank:** Timestamp Register	166

performance results of Figure 8, however, Rerun should be preferred in larger multicore chips.

Bandwidth. Rerun and FDR-2 append payloads of identical size to the same subset of coherence messages. Thus, the interconnection bandwidth overhead due to piggybacked timestamps is the same in both systems. Including the additional bandwidth needed to write the log to the memory system, Rerun requires just under a 10% increase in total on chip bandwidth. This is similar to FDR-1 and FDR-2, which both report a 10% increase, and Strata-Snoop and Strata-Dir, which report a 10% and 12% increase, respectively.

6. Related Work

Rerun builds on the pioneering work in hardware memory race recording. In 1991, Bacon and Goldstein [2] recorded all coherence traffic on a snooping bus. Rerun has much smaller logs and does not require a bus. The Flight Data Recorder (FDR) was proposed in 2003 [39], used in BugNet [26], and enhanced in 2006 [40]. FDR adapts Netzer's transitivity reduction [27] so that coherence messages can carry scalar, not vector, instruction counts and relies on substantial per-core state to reduce log size. Rerun also uses coherence messages to carry values, but these values are globally-meaningful Lamport Clocks. More importantly, Rerun

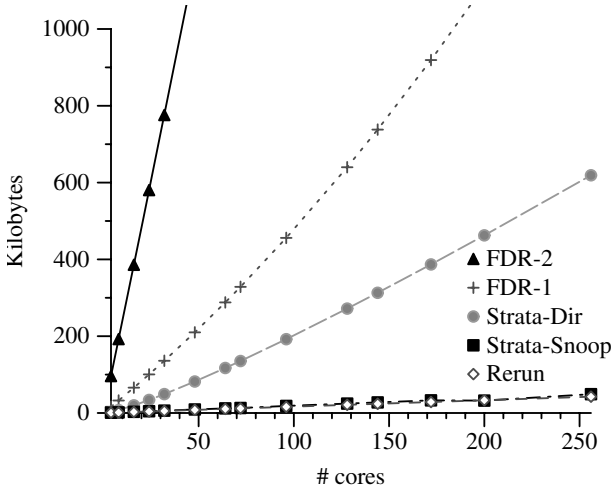

Figure 9: Hardware overheads as the number

substantially reduces per-core state by tracking non-conflicting episodes rather than conflicting memory accesses. At the same time, Rerun modestly decreases log size. ReEnact [31] uses thread level speculation techniques to record and replay a multiprocessor execution. This required substantial hardware support, including versioning caches and rollback capability. CORD [30] uses Lamport clocks to order memory races, but does so at the granularity of a cache block. Rerun uses one logical timestamp to represent the entire memory image, which substantially lowers the required hardware overhead. Strata [25] partitions a multiprocessor execution into global strata, where a stratum ends when a core seeks to read or write a block written by another core in the current stratum. While Rerun also ends its episodes on such conflicts, episodes differ from strata. Most importantly, each Rerun episode is local to a core, while each stratum is global across the system. In part for this reason, our results show that as the number of cores per system increases, Rerun's log size scales better than Strata's.

Rerun also builds on software race recorders that used Lamport Clocks to establish a partial ordering of the system execution. ROLT [18] applied ordering only to synchronization primitives and not general memory operations. As a result, deterministic replay is provided only for executions that are data-race-free. InstantReplay [17] also used Lamport Clocks for ordering in the system, although unlike ROLT, InstantReplay recorded races between all memory operations by wrapping every shared object access with monitor code. However, due to the extra monitor code, the execution overhead of InstantReplay was obtrusive. In contrast, Rerun uses hardware to record memory races at low overhead. ReVirt [8] records a

multiprocessor execution on a page granularity. Write permissions are granted exclusively, allowing the system to record races by remembering the grant order. This mechanism limits concurrency during recording, resulting in significant slowdowns.

Rerun's episodes derive inspiration from aggressive implementations of sequential consistency. BulkSC [6] builds *chunks* of dynamic instructions that it seeks to commit atomically by transmitting signatures to other cores or a memory controller. On success, the chunk commits "in bulk;" on failure, a chunk aborts "in bulk" (e.g., back to a checkpoint). In contrast, Rerun never aborts an execution, but rather passively records the core's execution. Also, Rerun's Bloom filters are never transmitted. Store-wait-free [37] builds small atomic sequences when memory references are performed out of order, which then either commit or abort atomically. Once again, Rerun never alters or aborts execution. Furthermore, Rerun's episodes are always active and much larger than store-wait-free's atomic sequences.

Rerun also builds on ideas from hardware transactional memory. First, Rerun's episodes look somewhat like the implicit transactions some systems use to execute critical sections in parallel [32, 21]. However, these systems trigger aborts (Rerun doesn't) and do not record races (Rerun does). Second, Transaction Coherence and Consistency (TCC) [11] assumes "all transactions all the time" and can enable deterministic replay by recording the total order in which transactions obtain the "commit token." While Rerun assumes "all episodes all the time," all its episodes are transparent to programmers, never abort, and are ordered via a distributed implementation of Lamport Clocks without a commit token implementation. Third, Rerun's read and write set filters borrow from transactional memory implementations that use signatures [5, 42, 23, 35]. Rerun uses these filters to passively record execution and not to enable aborts of concurrent conflicting programmer-specified transactions.

7. Conclusions

We develop Rerun, a memory race recorder that uses episodes to efficiently log memory reference order. Episodes are identified by length, determined by how long a thread executes without conflicting with other threads, and ordered with Lamport Clocks. We show Rerun uses small hardware state, generates a small race log, and scales well as number of cores per system grows. Future work will seek to speed replayer execution and reduce coherence protocol overhead.

Acknowledgements

We thank Dan Gibson, Mike Marty, Dan Sorin, Mike Swift, Min Xu, the Wisconsin Multifacet group, and the Wisconsin Computer Architecture Affiliates for their comments and/or proofreading. Finally, we thank the Wisconsin Condor project, the UW CSL, and Virtutech for their assistance.

This work is supported in part by the National Science Foundation (NSF), with grants CCR-0324878, CNS-0551401, and CNS-0720565, as well as donations from Intel and Sun Microsystems. Hill has in significant financial interest in Sun Microsystems. The views expressed herein are not necessarily those of the NSF, Intel, or Sun Microsystems.

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. 5th Workshop on Computer Architecture Evaluation Using Commercial Workloads*, pages 30–38, Feb. 2002.

[2] D. F. Bacon and S. C. Goldstein. Hardware-Assisted Replay of Multiprocessor Programs. *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, pages 194–206, 1991.

[3] T. C. Bressoud and F. B. Schneider. Hypervisor-based Fault Tolerance. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating Systems Principles*, 1995.

[4] J. L. Carter and M. N. Wegman. Universal Classes of Hash Functions (extended abstract). In *Proceedings of the 9th Annual ACM Symposium on Theory of Computing*, pages 106–112, 1977.

[5] L. Ceze, J. Tuck, C. Cascaval, and J. Torrellas. Bulk Disambiguation of Speculative Threads in Multiprocessors. In *Proc. of the 33nd Annual Intnl. Symp. on Computer Architecture*, June 2006.

[6] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. BulkSC: Bulk Enforcement of Sequential Consistency. In *Proc. of the 34th Annual Intnl. Symp. on Computer Architecture*, June 2007.

[7] G. W. Dunlap, S. T. King, S. Cinar, M. Basrai, and P. M. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *Proc. of the 2002 Symp. on Operating Systems Design and Implementation*, pages 211–224, Dec. 2002.

[8] G. W. Dunlap, D. Lucchetti, P. M. Chen, and M. Fetterman. Execution Replay on Multiprocessor Virtual Machines. In *International Conference on Virtual Execution Environments (VEE)*, 2008.

[9] K. Gharachorloo, A. Gupta, and J. Hennessy. Two Techniques to Enhance the Performance of Memory Consistency Models. In *Proceedings of the International Conference on Parallel Processing*, volume I, pages 355–364, Aug. 1991.

[10] C. Gniady, B. Falsafi, and T. Vijaykumar. Is SC + ILP = RC? In *Proc. of the 26th Annual Intnl. Symp. on Computer Architecture*, pages 162–171, May 1999.

[11] 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 *Proc. of the 31st Annual Intnl. Symp. on Computer Architecture*, June 2004.

[12] Intel. Intel 64 Architecture Memory Ordering White Paper. Technical Report SKU 318147-001, Intel Corp., Aug. 2007. http://developer.intel.com/products/processor/manuals/318147.pdf.

[13] A. Kamil, J. Su, and K. Yelick. Making Sequential Consistency Practical in Titanium. In *Proc. of SC2003*, pages 15–30, Nov. 2003.

[14] C. Kozyrakis and K. Olukotun. ATLAS: A Scalable Emulator for Transactional Parallel Systems. In *Workshop on Architecture Research using FPGA Platforms*, Feb. 2005.

[15] L. Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[16] J. R. Larus and R. Rajwar. *Transactional Memory*. Morgan & Claypool Publishers, 2007.

[17] T. J. Leblanc and J. M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE Transactions on Computers*, C-36(4):471–482, Apr. 1987.

[18] L. Levrouw and K. Audenaert. Minimizing the Log Size for Execution Replay of Shared-Memory Programs. In *Lecture Notes In Computer Science; Vol. 854, Parallel Processing: CONPAR 94 - VAPP VI, Third Joint International Conference on Vector and Parallel Processing, Linz, Austria, September 6-8, 1994, Proceedings*, pages 76–87, 1994.

[19] D. Lucchetti, S. K. Reinhardt, and P. M. Chen. ExtraVirt: Detecting and recovering from transient processor faults. In *2005 Symp. on Operating System Principles work-in-progress session*, Oct. 2005.

[20] 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.

[21] J. F. Martínez and J. Torrellas. Speculative Synchronization: Applying Thread-Level Speculation to Explicitly Parallel Applications. In *Proc. of the 10th Intnl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 18–29, Oct. 2002.

[22] A. McDonald, J. Chung, B. Carlstrom, C. C. Minh, H. Chafi, C. Kozyrakis, and K. Olukotun. Architectural Semantics for Practical Transactional Memory. In *Proc. of the 33nd Annual Intnl. Symp. on Computer Architecture*, June 2006.

[23] C. C. Minh, M. Trautmann, J. Chung, A. Mcdonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An Effective Hybrid Transactional Memory System with Strong Isolation Guarantees. In *Proc. of the 34th Annual Intnl. Symp. on Computer Architecture*, June 2007.

[24] M. J. Moravan, J. Bobba, K. E. Moore, L. Yen, M. D. Hill, B. Liblit, M. M. Swift, and D. A. Wood. Supporting Nested Transactional Memory in LogTM. In *Proc. of the 12th Intnl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 359–370, Oct. 2006.

[25] S. Narayanasamy, C. Pereira, and B. Calder. Recording Shared Memory Dependencies Using Strata. In *Proc. of the 12th Intnl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 229–240, Oct. 2006.

[26] S. Narayanasamy, G. Pokam, and B. Calder. BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging. In *Proc. of the 32nd Annual Intnl. Symp. on Computer Architecture*, pages 284–295, June 2005.

[27] R. H. B. Netzer. Optimal Tracing and Replay for Debugging Shared-Memory Parallel Programs. In *Proc. of the ACM/ONR Workshop on Parallel and Distributed Debugging (PADD)*, pages 1–11, 1993.

[28] J. Oplinger and M. S. Lam. Enhancing Software Reliability with Speculative Threads. In *Proc. of the 10th Intnl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 184–196, Oct. 2002.

[29] D. A. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, and N. Sastry. Recovery-Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies. Technical report, UC Berkeley Computer Science Technical Report UCB//CSD-02-1175, Mar. 2002.

[30] M. Prvulovic. CORD: Cost-effective (and nearly overhead-free) Order Recording and Data race detection. In *Proc. of the 12th IEEE Symp. on High-Performance Computer Architecture*, Feb. 2006.

[31] M. Prvulovic and J. Torrellas. ReEnact: Using Thread-Level Speculation Mechanisms to Debug Data Races in Multithreaded Codes. In *Proc. of the 30th Annual Intnl. Symp. on Computer Architecture*, pages 110–121, June 2003.

[32] R. Rajwar and J. R. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *Proc. of the 34th Annual IEEE/ACM International Symp. on Microarchitecture*, Dec. 2001.

[33] M. Ramakrishna, E. Fu, and E. Bahcekapili. Efficient Hardware Hashing Functions for High Performance Computers. *IEEE Transactions on Computers*, 46(12):1378–1381, 1997.

[34] 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 *Proc. of the 9th ACM Symp. on Parallel Algorithms and Architectures*, pages 199–210, June 1997.

[35] D. Sanchez, L. Yen, M. D. Hill, and K. Sankaralingam. Implementing Signatures for Transactional Memory. In *Proc. of the 40th Annual IEEE/ACM International Symp. on Microarchitecture*, Dec. 2007.

[36] D. L. Weaver and T. Germond, editors. *SPARC Architecture Manual (Version 9)*. PTR Prentice Hall, 1994.

[37] T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Mechanisms for Store-wait-free Multiprocessors. In *Proc. of the 34th Annual Intnl. Symp. on Computer Architecture*, June 2007.

[38] M. Xu. *Race Recording for Multithreaded Deterministic Replay Using Multiprocessor Hardware*. PhD thesis, University of Wisconsin, 2006.

[39] 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*, pages 122–133, June 2003.

[40] 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*, pages 49–60, Oct. 2006.

[41] 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.

[42] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood. LogTM-SE: Decoupling Hardware Transactional Memory from Caches. In *Proc. of the 13th IEEE Symp. on High-Performance Computer Architecture*, pages 261–272, Feb. 2007.