RelaxReplay: Record and Replay for Relaxed-Consistency Multiprocessors

Nima Honarmand and Josep Torrellas
University of Illinois at Urbana-Champaign
{honarma1,torrella}@illinois.edu
http://iacoma.cs.uiuc.edu

Abstract

Record and Deterministic Replay (RnR) of multithreaded programs on relaxed-consistency multiprocessors has been a long-standing problem. While there are designs that work for Total Store Ordering (TSO), finding a general solution that is able to record the access reordering allowed by any relaxed-consistency model has proved challenging.

This paper presents the first complete solution for hardware-assisted memory race recording that works for any relaxed-consistency model of current processors. With the scheme, called *RelaxReplay*, we can build an RnR system for any relaxed-consistency model and coherence protocol. RelaxReplay's core innovation is a new way of capturing memory access reordering. Each memory instruction goes through a post-completion in-order counting step that detects any reordering, and efficiently records it. We evaluate RelaxReplay with simulations of an 8-core release-consistent multicore running SPLASH-2 programs. We observe that RelaxReplay induces negligible overhead during recording. In addition, the average size of the log produced is comparable to the log sizes reported for existing solutions, and still very small compared to the memory bandwidth of modern machines. Finally, deterministic replay is efficient and needs minimal hardware support.

Categories and Subject Descriptors C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors) - MIMD Processors; D.1.3 [**Programming Techniques**]: Concurrent Programming - Parallel Programming

Keywords Memory Race Recording, Record and Deterministic Replay, Relaxed Consistency

1. Introduction

Record and Deterministic Replay (RnR) of multithreaded programs in multiprocessors is a concept that involves logging enough of a parallel execution to be able to replay it later deterministically. RnR has broad uses in parallel program debugging [1, 13, 29], security analysis [8, 11, 12], and fault-tolerant, highly-available systems [5, 7]. RnR for a program typically requires logging two sources of non-determinism during execution, namely, external inputs to the program (e.g., results of system calls) and the interleaving of shared-memory accesses from different processors. The latter consists of capturing the relative order of conflicting memory accesses. Hence, it is called Memory Race Recording (MRR).

While input recording is done by the Operating System (OS), MRR typically requires special hardware. This is because inserting software instrumentation for MRR (*e.g.*, [16, 22, 32]) results in significant execution slow down or increases the number of processors required. In addition, all these schemes distort the timing of execution, which can be a drawback in concurrency debugging. Finally, they require modifying the binary of the program to be recorded. For these reasons, there are several proposals for hardware-assisted MRR (*e.g.*, [3, 6, 9, 10, 15, 17–21, 23–25, 27, 33, 35, 36]). These schemes typically identify conflicting accesses by leveraging cache coherence protocol transactions. Most recent proposals record a processor's execution as a series of *Chunks* (or Blocks or Episodes) of instructions executed between coherence actions with other processors. This approach results in low recording overhead and small logs.

The majority of current proposals for hardware-assisted MRR require that the recorded execution obeys Sequential Consistency (SC) [14]. Under SC, memory-access instructions execute in program order, which substantially simplifies what events need to be logged and when. Unfortunately, commercial machines almost universally use more relaxed memory consistency models, allowing loads and stores to reorder. Recording such execution is especially challenging.

There have been a few proposals for MRR under non-SC models. All but one of them require the Total Store Ordering (TSO) memory model [3, 6, 10, 17, 24, 25, 36], which only allows loads to bypass stores. Such proposals either log which stores are bypassed [24, 25], or log the values read by the bypassing loads [3, 6, 10, 36], or use off-line analysis to identify the actual order that occurred [17]. The other proposal, called Rainbow [27], focuses on detecting SC violations as they happen, and recording enough information

ASPLOS '14, March 1–5, 2014, Salt Lake City, Utah, USA.

http://dx.doi.org/10.1145/2541940.2541979

to replay them. However, this scheme requires a coherence protocol that is centralized and that needs substantial hardware changes. Moreover, the operation of the scheme's major components is not clearly described in the paper. All these schemes are discussed in detail in Section 6. Overall, the long-standing problem of finding a general MRR solution that works for any relaxed-consistency model (such as that of ARM [2], Power [26] or Tile [30] processors) is still open.

This paper contributes with the first complete solution for hardware-assisted MRR that works for any relaxed-consistency model of current processors. With the scheme, called *RelaxReplay*, we can build an RnR system that works for any relaxed-consistency model and any cache coherence protocol. RelaxReplay's key innovation is a new approach to capture memory access reordering. Specifically, each memory instruction goes through a post-completion in-order counting step that detects any reordering, and efficiently records it in the log. We present two designs, called *RelaxReplay_Base* and *RelaxReplay_Opt*, with different emphases on hardware requirements, log size, and replay speed.

Several salient characteristics of the RelaxReplay mechanism to capture memory access reordering are:

- It only relies on the write atomicity property of coherence protocols, and not on knowing the detailed specifications of the particular relaxed-consistency model. Such specifications are often high-level and hard to map to implementation issues.
- It can be combined with the specific chunk-ordering algorithm of any existing chunk-based MRR proposal. As a result, that proposal, designed for a certain coherence protocol, can now record relaxed-consistency executions.
- It has modest hardware requirements. Its hardware is local to the processors and requires no change to the cache coherence protocol.
- It produces a compact log representation of a relaxed-consistency execution.
- The resulting log enables efficient deterministic replay with minimal hardware support.

We evaluate RelaxReplay with simulations of an 8-core Release-Consistent (RC) multicore running SPLASH-2 applications. The results show that RelaxReplay induces negligible overhead during recording. In addition, the average size of the log produced is 1–4x the log sizes reported by existing SC- or TSO-based MRR systems. Hence, the bandwidth required to save this log is still a small fraction of the bandwidth provided by current machines. Finally, deterministic replay using this log is efficient: the sequential replay of these 8-processor executions with minimal hardware support takes on average 6.7x as long as the parallel recording.

This paper is organized as follows: Section 2 provides a background; Sections 3 and 4 presents RelaxReplay's design and implementation, respectively; Section 5 evaluates RelaxReplay and Section 6 discusses related work.

2. Background on Chunk-Based Recording

State-of-the-art proposals for hardware-assisted MRR record each processor's execution as a series of *Chunks* (also called Blocks or Episodes) of instructions executed between communications with other processors [3, 9, 10, 18, 19, 23–25, 33]. The chunks of different processors are ordered in a graph based on inter-processor data dependences. A typical chunk-based recorder provides three main functionalities: (1) establishes chunk boundaries such that each chunk's execution appears atomic, (2) establishes a proper order between chunks that captures all data dependences (to ensure correct replay) and has no cycles (to avoid replay deadlocks), and (3) represents chunks in the log in an efficient format.

Chunk boundaries are set at points where the executing processor communicates with other processors. Chunk-based recorders usually keep track of the read and write operations performed by the instructions of the current chunk. Often, the addresses of these operations are hashed in Bloom filters [4] and stored as read and write signatures. At the same time, the hardware checks for cache-coherence transactions that conflict with the read or write set of the current chunk. When one does, we have detected an inter-processor data dependence. Then, in simple designs, the current chunk is terminated and a new chunk starts. There are optimizations that allow chunks to grow beyond the conflicts.

Chunk-based recorders must ensure that the chunk containing the source of a dependence is ordered before the chunk containing the destination of it. For this, some schemes piggyback ordering information on coherence messages (e.g., [10, 33]) or add new messages [23]. Specifically, when an incoming coherence request conflicts with the local chunk, the global order of the local chunk is sent to the requesting processor (or broadcasted to all processors in [23]), so that its chunk orders itself after the local one. Alternatively, other schemes rely on a globally-consistent clock (e.g., [9, 24, 25]) that is available to all processors to establish chunk ordering. In both cases, by replaying the chunks according to their global order, all data dependences will be enforced.

Chunk-based recorders log chunks in a very efficient format. Specifically, a chunk is represented as the number of instructions (or memory operations) performed in the chunk, together with the recorded global ordering of the chunk.

2.1 Advantages of Chunk-Based Recording

Chunk-based recorders have at least three advantages over non chunked-based ones that have made them popular. Firstly, their operation lends itself to a relatively simpler hardware implementation in the cache hierarchy, while still generating small log sizes. Secondly, they support application-level RnR especially well because their recording hardware can be easily virtualized [9, 19, 25] and shared by multiple independent applications.

A third advantage is that the resulting logs can be efficiently replayed with minimal hardware support [9]. Specif-

ically, all that they need is a counter that counts the number of instructions (or memory-access instructions) executed, and then triggers a synchronous interrupt when the number of instructions in the chunk are exhausted. In this way, instructions can be replayed natively by the hardware rather than being simulated by an instruction simulator. At the same time, a simple software module can enforce the recorded chunk order [9]. This combined hardware/software solution enables efficient native replay. Moreover, with the appropriate design, the resulting logs can be replayed in parallel [3, 9], and deliver fast replay.

2.2 Main Limitation: Access Reordering

In its basic form, chunk-based recording (as well as non chunk-based one) relies on the assumption that processors expose their memory operations to the coherence subsystem in program order, providing a sequentially-consistent environment [14]. Hence, any execution that violates SC cannot be captured by these recorders.

Unfortunately, commercial machines almost universally use more relaxed memory models, allowing loads and stores to perform out of program order. For example, to show how aggressive modern processors are, Figure 1 shows the fraction of memory-access instructions that are performed out of program order — i.e., with some earlier memory instructions still pending. The details of the experiment are discussed in Section 5.1. Of all the memory instructions, on average, 59% are out-of-order loads and 3% are out-of-order stores.

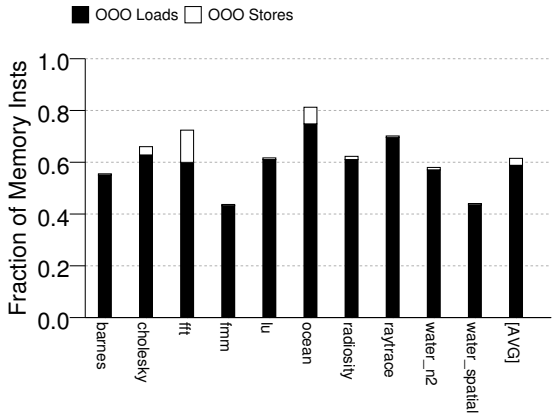


Figure 1. Fraction of all the memory-access instructions that are performed out of program order.

To begin to address this problem, there have been a few proposals for MRR under non-SC models. As indicated in Section 1 and discussed in Section 6, however, these proposals address only a conservative memory model (TSO), or are otherwise limited. To help popularize RnR, we need to find a general solution for MRR that works for any of the relaxed-consistency models used in current processors (such as ARM [2], Power [26] or Tile [30]). The rest of this paper presents a solution to this problem that is compatible with the use of chunk-based recording.

3. RelaxReplay Design

3.1 Concept of Interval

To understand RelaxReplay, we define the concepts of *performing* and *counting* a memory-access instruction, and the notion of an *Interval*. A load instruction *performs* when the data loaded returns from the memory system and is deposited into a register. Later, the load retires when it reaches the head of the Reorder Buffer (ROB) and has already performed. A store instruction retires when it reaches the head of the ROB and its address and data are available. At this point, the store is deposited into the write buffer. Depending on the memory consistency model, the store can be merged with the memory system right away, or has to wait to do so until all earlier stores have been removed from the write buffer. Merging may trigger a coherence transaction. When the coherence transaction terminates (i.e., when all the necessary replies and acknowledgments have been received), the store has *performed*. Finally, in RelaxReplay, each retired load and each performed store in the processor goes through an additional *logical* stage in program order that we call *Counting*. Counting records the completion of the instruction in program order. Hence, each memory-access instruction has a *Perform* event and a *Counting* event.

An *Interval* in the execution of a processor is the *period of time* between two consecutive communications of the processor with other processors. An interval has a *Perform Set* and a *Counting Set*. These are the sets of perform and counting events, respectively, that took place in the processor during the interval. The set of perform events in an interval may correspond to memory-access instructions that are *not* contiguous in program order. This is because, in a relaxed-consistency machine, accesses can perform out of order. This is in contrast to the instructions of a chunk in a conventional chunk-based recorder, which are required to be contiguous. However, the set of counting events in the interval do correspond to consecutive memory-access instructions, since counting is done in program order.

3.2 Main Idea in RelaxReplay

In an RnR environment that supports general relaxed consistency models, working with chunks of contiguous instructions, as in conventional chunk-based recorders, is inconvenient. Instead, we propose to use the interval abstraction, which directly corresponds to the work performed between communications. To show the usability of intervals, we make two observations.

Observation 1: In memory-consistency models that support write atomicity, the perform event of a given access can only be placed in a single interval.

The property of write atomicity means that a write operation by a processor can be observed by another processor only if it has been made visible to all other processors, and that writes to the same location are serialized [28]. This prop-

erty, which is typically enforced by the coherence substrate, is provided by all the popular multiprocessor systems in use today. It implies that the execution of a memory access can be thought of as atomic, and can only be placed in a single interval, namely the one where the access *performs*. As a result, we can record the execution of a processor as a sequence of intervals, where each access is assigned to the interval where it performs.

Unfortunately, representing an interval as a set of perform events is inefficient. Indeed, since memory instructions are performed out of program order, we would have to record the complete list of such events. To reduce the state we need to log, it is better to record the interval as a set of counting events (which can be efficiently represented as a *range* of consecutive in-order memory instructions) plus some reorder information. A second observation allows us to keep this additional reorder information that we need to record to a minimum.

Observation 2: For the large majority of memory-access instructions, we can logically move the perform event forward in time to coincide with its counting event.

Given a memory-access instruction by a processor (P_1), we can logically move its perform event forward in time to coincide with its counting event if no other processor (P_j) has observed the access between the two points in time. P_j observes the access if it issues a conflicting access to the same (line) address that causes a coherence transaction that reaches P_1 between the two points. By "moving", we mean that, as far as the other processors are concerned, the instruction can be assumed to have performed at the point of its counting. Since the access has not yet been observed by any other processor, this assumption will not affect any of the inter-processor dependences and, therefore, is correct. Fortunately, in practice, the large majority of accesses are not observed between the two events.

As an example, Figure 2(a) shows a store (ST) and a load instruction (LD) from a processor in program order, and their perform (P) and counting (C) event times. It also shows the time when an external communication occurs and, therefore, the interval terminates. In the figure, the perform events are in order. Figure 2(b) shows the case when the perform events are out of order. In both cases, each perform event happens in the same interval as its corresponding counting event and, thus, can be trivially moved to its counting time. Therefore, in both cases, we can concisely represent this interval as including the two accesses in program order.

In Figure 2(c), the load has its perform and counting events in two different intervals. In this paper, we present two version of RelaxReplay, depending on how we deal with this case. In a base design with simpler hardware, called *RelaxReplay_Base*, the perform event is never moved across intervals to its counting event; in an optimized design with more hardware, called *RelaxReplay_Opt*, the perform event is still moved across intervals to its counting event if none of

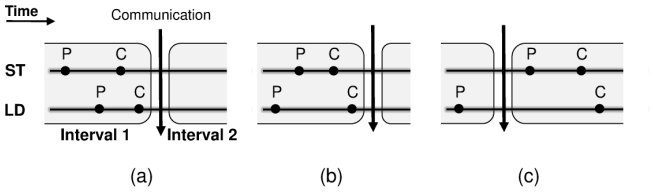


Figure 2. Examples of a two-instruction pattern with different timings for their perform (P) and counting (C) events.

the coherence transactions received between the two events conflicts with the (line) address of the access.

If RelaxReplay is able to move all the perform events to their counting events, each interval is concisely logged as comprising a certain number of accesses in program order — irrespective of the *actual access reordering* that occurred during recording due to the relaxed consistency model. Otherwise, the log entry for an interval also includes additional information on what accesses were counted in the interval but were out of order. We will discuss the exact representation later. Since, for the large majority of accesses, RelaxReplay is able to move the perform events to the counting events, the RnR log of intervals is both *stored and replayed* efficiently.

Overall, RelaxReplay is able to record an execution under any memory consistency model with write atomicity, and store it in a log for efficient deterministic replay. RelaxReplay relies on hardware that tracks the perform and counting events of each memory access and, while watching for conflicting accesses from other processors, tries to combine them before storing a compact representation of intervals in the log.

Note that RelaxReplay's goal is to record intervals. For a full MRR solution, we also need a mechanism to establish a proper order between intervals of different processors. For this, we can use any of the existing chunk-based recording schemes. Such schemes now use coherence messages and read/write signatures to establish a proper order between intervals rather than chunks.

Next, we describe the architecture that processes perform and counting events, how it handles store-to-load forwarding, and how we replay a RelaxReplay log.

3.3 Tracking Instruction Events in RelaxReplay

Intuitively, the RelaxReplay architecture requires a longer ROB that keeps each memory-access instruction in the processor beyond retirement, and until it is ready to be counted. At that point, if the instruction's perform event can be moved to its counting event, the instruction is included in the current interval as an in-order access, and logged as such. Otherwise, the instruction is included in the current interval as a reordered access, with enough state added to the log so that it can be correctly replayed.

In practice, rather than enlarging the ROB, RelaxReplay adds a hardware structure to the processor that works in parallel with the ROB for memory-access instructions. The structure is a circular FIFO called *Tracking Queue* (TRAQ)

(Figure 3). As a memory-access instruction is inserted in the ROB, it is also inserted in the TRAQ. A memory-access instruction is removed from the TRAQ when it is at the head of the TRAQ and is ready to be counted — i.e., for a load, it is performed and retired, and for a store, it is retired and performed. At that point, the instruction is counted and added to the log record for the interval. Note that the TRAQ can contain both non-retired and retired accesses. The ROB-like structure of the TRAQ enables RelaxReplay to handle the squashing of speculative instructions easily, as we explain in Section 4.

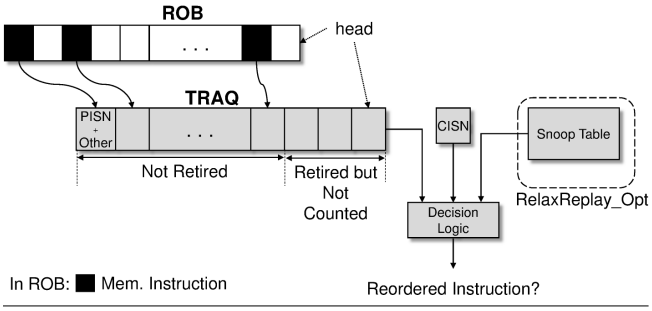


Figure 3. High-level architecture of RelaxReplay.

RelaxReplay keeps in a register the ID of the interval that is currently being processed at the head of the TRAQ. This ID is a counter called *Current Interval Sequence Number* (CISN) (Figure 3). Every time the processor communicates with another processor, the current interval is terminated, its information is stored in the memory log, the CISN is incremented, and a new interval starts.

The fundamental operation of the RelaxReplay hardware is simple. When a memory-access instruction is performed, the current value of the CISN is copied to the instruction's TRAQ entry. It is stored in a field called *Performance Interval Sequence Number* (PISN). When the instruction reaches the TRAQ head and is counted, its PISN is compared to the CISN. At this point, there are several possible outcomes.

First, if the two values are the same, the interval has not changed since the perform event. Hence, RelaxReplay logically assumes that the memory-access instruction performs at the point of counting. In this case, RelaxReplay simply increments the count of consecutive memory-access instructions that have executed in this interval. Such count will be included in the log record for the interval that will be stored to memory when the interval terminates. An example of this case is shown in Figure 4(a), which depicts a load that performs and is counted in interval 10, and whose perform point is logically moved by RelaxReplay to its counting point.

Second, if the PISN and CISN are different, the interval has changed because the processor has communicated between the perform and counting events. In RelaxReplay_Base, we process the access as reordered, as we will see later. In RelaxReplay_Opt, the hardware checks if the access is indeed reordered by comparing its (line) address to the (line) addresses of all the coherence transactions that the processor received since the PISN interval. Such addresses are collected in hardware in a structure called *Snoop Table* (Figure 3). This structure is only present in RelaxReplay_Opt, and is described in detail in Section 4.2.

If the comparison shows that no transaction conflicting with that address has been received, then RelaxReplay logically assumes that the memory-access instruction performs at this point, as in the first case. As before, RelaxReplay increments the count of consecutive memory-access instructions that have executed in this interval. An example is shown in Figure 4(b), which depicts a load that performs in interval 10 and is counted in 12. Since the processor has received no transaction that conflicts with this address in the meantime, the perform point is logically moved.

However, if the comparison finds that a conflicting transaction has been received, or the machine only supports RelaxReplay_Base, then the hardware records a reordered access. The following sections describe the cases of a reordered load and a reordered store separately.

3.3.1 Reordered Loads

To be able to record reordered loads, RelaxReplay needs to retain the values that loads obtain as they perform, until the loads' counting time. Such values are stored in the corresponding TRAQ entries, as part of what Figure 3 refers to as *Other*.

When RelaxReplay counts a load and finds that it is reordered, it does not increment the count of consecutive memory-access instructions executed in this interval. Instead, it adds a special type of entry in the log record for the interval. The entry contains the value that was returned by the load as it performed (and was retained in the TRAQ). Later, when the execution is deterministically replayed, the value is read from the log and supplied to the destination register of the load. In this way, the replay of the load in program order can correctly reproduce what happened in the recorded execution out of program order. If, instead, during the replay, the load tried to access the memory system as it replayed, it might read an incorrect value. Note that any consumers of the load, as they are replayed in program order, will obtain the correct value. Xu et al. [36] used this approach of recording the values returned by out-of-order loads in the log for TSO machines.

An example is shown in Figure 4(c). A load performs in interval 10, and the processor later receives a coherence event that conflicts with the loaded address. The load is counted in interval 12. RelaxReplay then takes the value read by the load and stores it in the log record for interval 12.

3.3.2 Reordered Stores

To be able to record reordered stores, RelaxReplay needs to retain the values they write and the addresses they write to, until the writes' counting time. Such values are saved in the TRAQ entries as part of the *Other* fields.

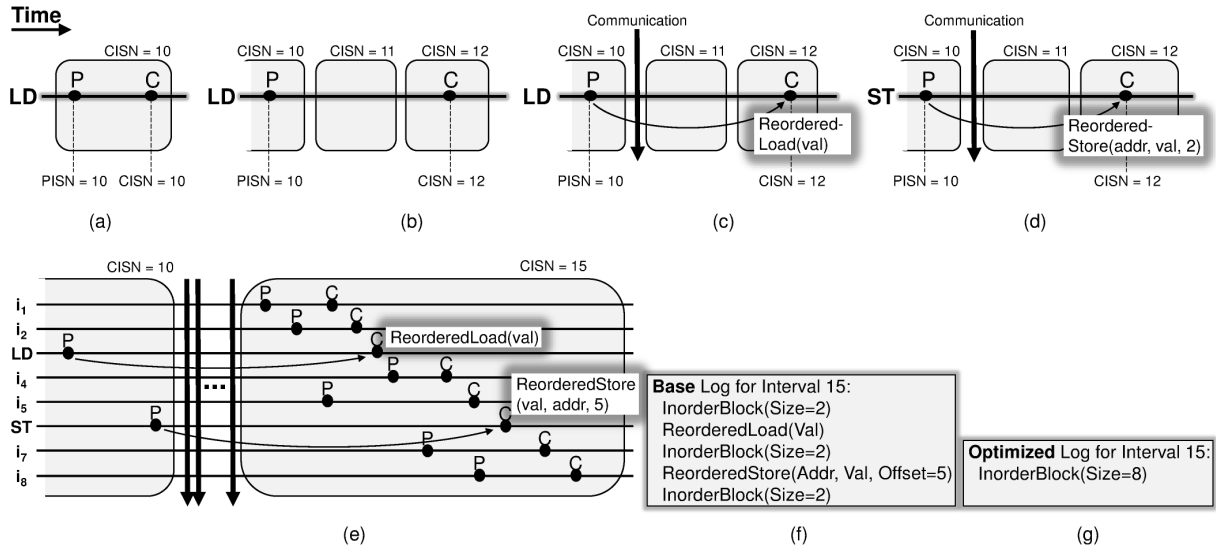


Figure 4. Examples of RelaxReplay operation with perform (P) and counting (C) events.

When RelaxReplay counts a store and declares it reordered, it does not increment the count of consecutive instructions executed in this interval. Instead, it adds another special type of entry in the log record for the interval. The entry contains the address written to, the value written, and the difference between CISN and the value of PISN in the store's TRAQ entry. We call this difference *Offset*; it denotes how many intervals ago the store performed.

Before this log can be used for deterministic replay, this entry needs to be extracted from this interval's record and inserted in the record of an earlier interval — specifically, at the end of the interval that is *Offset* positions earlier, which is the interval when the store performed. In the interval where the store is counted, we leave a dummy entry so that the store is not re-executed there. This "patching" step can be done as an off-line pass or on the fly as the log is read for replay.

After this change is made, the log is ready for replay. The store entry is found in the interval when it was performed, and the log contains the value to store and the address to store to. The store is thus executed, exactly reproducing the conditions in the recorded execution. In the interval where the store was counted, the store instruction is skipped (as indicated by the dummy entry mentioned above).

Figure 4(d) shows an example of a store that performs at interval 10, and the processor later receives a conflicting coherence event. The store is counted in interval 12. RelaxReplay then takes the value and address from the TRAQ and, together with an offset of 2, stores them in the log record for interval 12.

3.3.3 Example

To understand the format of the log record for an interval, Figure 4(e) shows the more extensive example of an interval that counts 8 memory-access instructions. Of these, i_1, i_2, i_4, i_5, i_7, and i_8 both perform and are counted in interval 15. However there is a load (*LD*) and a store (*ST*) that perform in interval 10 and are counted in interval 15. Assume that none of the communications between intervals 10 and 15 conflict with the addresses accessed by *LD* or *ST*.

If we use RelaxReplay_Base, the hardware does not know that there is no conflict and assumes that *LD* and *ST* are reordered. Hence, as shown in Figure 4(e), as it counts *LD*, it reads the value that *LD* loaded and saves it in the log record. As it counts *ST*, it reads the value that *ST* stored and the address it stored to, computes the offset of 5, and saves the three values in the log record.

Figure 4(f) shows the resulting log record for interval 15. It contains several entries, which are inserted in order as instructions are counted in order. As i_1 and i_2 are counted, they increment the counter of consecutive instructions that have executed in this interval. As RelaxReplay reaches *LD* and finds it reordered, it saves the counter in an entry of type *InorderBlock*, and resets the counter. This means that there is a group of 2 in-order instructions executed. Then, it records an entry of type *ReorderedLoad* with the value of the load. This means that the next instruction in program order is a reordered load. Then, for instructions i_3 and i_4, RelaxReplay records another entry of type *InorderBlock* with size 2. Then, RelaxReplay records an entry of type *ReorderedStore* for *ST*, with its address, value, and offset. This entry signifies that the next instruction in program order is a reordered store. Finally, RelaxReplay stores another entry of type *InorderBlock* with size 2 for i_7 and i_8. This information is enough for the deterministic replay of these instructions. As will be seen later, this log format enables efficient replay.

Figure 4(g) shows the log for the same interval using RelaxReplay_Opt. Since RelaxReplay_Opt discovers that none of the intervening coherence transactions conflicts with the addresses of *LD* or *ST*, it records *LD* and *ST* as in-order accesses. In general, since the number of truly reordered accesses is very small, this log format is often very compact.

More details of the hardware and logging are presented in Section 4. In particular, the *InorderBlock* entries count the number of total instructions in order, not just memory-access instructions. This design eases replay.

3.4 Handling Store-to-Load Forwarding

Modern superscalar processors typically allow store-to-load forwarding, whereby a load gets its value from an older store of the same processor that is pending in the write buffer. Such a load is not serviced off the coherent memory; it obtains its value from the non-coherent write buffer. In this section we show that RelaxReplay correctly records such loads.

Figure 5 shows the timing of a forwarding instance, where a load (*LD*) obtains its value from an older store (*ST*). Following RelaxReplay's operation, *LD* performs as soon as it gets the forwarded data, before *ST* merges with the memory system and performs. Later, *ST* is counted and *LD* is counted.

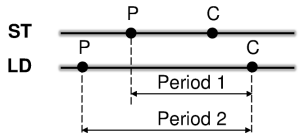


Figure 5. Timing of store-to-load forwarding.

RelaxReplay seamlessly supports this case. Since *LD* gets its value from *ST*, we can assume it *logically* performs at the same time as *ST*. Thus, in order to correctly record *LD*, we only need to monitor conflicting accesses between *ST*'s perform event and *LD*'s counting event (Period 1 in Figure 5). However, this period is properly contained between *LD*'s perform and counting events (Period 2 in Figure 5). Thus, if there is a change of interval (in RelaxReplay_Base) or reception of a conflicting coherence transaction (in RelaxReplay_Opt) in Period 2, we conservatively assume that it happened in Period 1. In this case, the hardware saves in the log the value obtained by *LD* at its perform point, and the replay system later uses it at the counting point. Otherwise, the hardware correctly moves *LD*'s perform point to its counting point. No change to RelaxReplay is needed.

3.5 Replaying a RelaxReplay Log

The log generated by RelaxReplay is very compact and enables efficient replay using only minimal hardware support. To replay an execution, we use a module in the OS as in the Cyrus system [9]. Specifically, during replay, the OS reads the log of intervals and enforces the order of the intervals. As the OS reads the record for an interval, before launching its execution, it waits until all intervals ordered before this interval finish executing. This can be accomplished using software synchronization through condition variables or semaphores. In addition, the OS also injects the application inputs that were recorded in the original execution.

The log record for an interval can have three types of relevant entries: *InorderBlock*, *ReorderedLoad*, and *ReorderedStore*. If an *InorderBlock* entry is found, the OS configures a hardware counter to generate an interrupt when the number of executed instructions equals the size of the block. This approach, proposed in the Cyrus system, requires a synchronous interrupt from the counter — *i.e.*, the interrupt should be triggered upon (and before) executing the first instruction after the block. When the block is complete, the interrupt transfers the control back to the OS. This instruction counting mechanism is similar to performance counters available in modern commercial microprocessors. It is the only hardware support needed to replay RelaxReplay logs.

If a *ReorderedLoad* entry is found, the OS reads the value from the log. It then saves it in the destination register of the load that is part of the architectural context of the application saved in the OS. Recall that the application context was saved upon entering the OS and will be restored before exiting the OS. The OS also advances the program counter, which is also stored as part of the architectural context.

If a *ReorderedStore* entry is found, the OS reads the address and value from the log and performs the memory update. Recall that we are now in the interval where this store *performed*, not where the store was counted. This is because this entry was processed earlier by a "patching step" (Section 3.3.2), which moved it from the store's counting interval to its perform interval. Hence, in the current interval, there is no corresponding store instruction. Therefore, the OS does not advance the program counter. Later, when the OS reaches the interval where the store was counted, the OS will find the corresponding dummy entry. At that point, the OS will take no action beyond advancing the program counter by one.

When all the entries of the interval are processed, the OS uses software synchronization to signal the completion of this interval to its successors. Then, it reads the next interval from the log. Note that the replay process is oblivious to whether the log comes from RelaxReplay_Base or RelaxReplay_Opt; both use the same log format.

3.6 Discussion

RelaxReplay can be used to convert any of the existing chunk-based MRR schemes [3, 9, 10, 23–25, 33] to an MRR solution for relaxed-consistency models. In this case, as shown in Figure 7, RelaxReplay uses the chunk-ordering mechanism of the specific MRR scheme to form and order intervals and, then, uses the techniques outlined in this paper to capture instruction reordering. In doing so, RelaxReplay retains the basic properties of the original chunk-based MRR scheme. For example, if the original scheme admits parallel replay of chunks [3, 9], then the resulting interval-based solution will admit parallel replay of intervals.

We designed RelaxReplay in a way that does not rely on knowing the detailed requirements of a particular relaxed-consistency memory model. RelaxReplay works for any relaxed-consistency model as long as the coherence substrate supports write atomicity. This is a well-established property of current memory subsystems that is likely to hold in future generations of processors. We chose this approach because

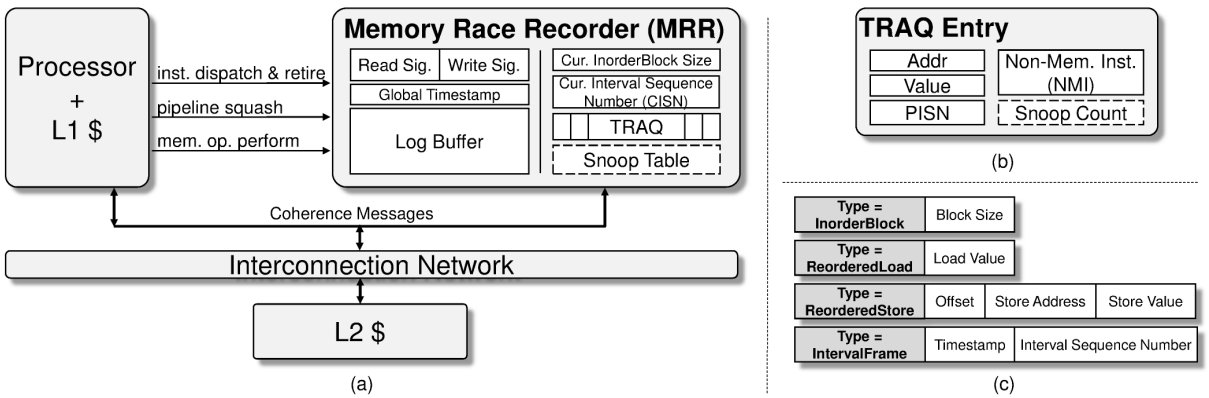


Figure 6. RelaxReplay architecture in detail: per-processor Memory Race Recorder (MRR) (a), TRAQ entry (b), and format of the different entry types in an interval's log record (c). The dashed boxes indicate the components specific to RelaxReplay_Opt.

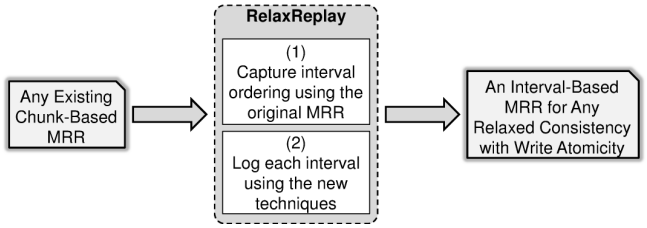


Figure 7. RelaxReplay can be paired with any chunk-based MRR scheme.

the memory models of most commercial processors are ill defined. In addition, new generations of processors may add new instructions to their ISAs that use different memory models than the older instructions. Finally, memory models are often defined in abstract, usually declarative, terms that do not provide much intuition about the implementation techniques used to support them.

4. Detailed Implementation Issues

4.1 Memory Race Recorder

Following previous RnR designs, we place the hardware for recording memory races in a per-processor Memory Race Recorder (MRR) module. This module is shown in the top right part of Figure 6(a). It comprises two parts. On the left side, there is the mechanism for creating and ordering intervals, which can reuse any of the designs proposed by existing chunk-based recorders. On the right side, there is the mechanism to track events within intervals, which is the proper RelaxReplay hardware. The inputs to the MRR module are processor signals (instruction dispatch into the ROB, instruction retirement, memory operation performed, and pipeline squash), and memory system signals (coherence transactions).

For the mechanism to create and order intervals, we show a design that follows the QuickRec [25] approach. This approach records a total order of intervals based on a globally-consistent scalar timestamp. The timestamp associated with each interval is the cycle count of a global clock when the interval was terminated. Intervals are ordered according to their timestamps. The scheme uses a snoopy coherence protocol.

As shown in the figure, the hardware needed is a pair of Bloom filters [4] as the read and write signatures of the current interval, a Global Timestamp counter that counts the number of cycles of a global clock, and a Log Buffer that automatically saves the log records. When memory operations are performed, their line addresses are inserted into the signatures. Snooped coherence transactions are checked against the signatures; if a conflict is detected, the current interval is terminated.

The proper RelaxReplay hardware is on the right side, and records the events within an interval. It comprises the TRAQ, the Current Interval Sequence Number (CISN), the Current InorderBlock Size count, and the Snoop Table. The latter is only needed in RelaxReplay_Opt and will be discussed later. The Current InorderBlock Size count is the number of in-order instructions that have so far been counted for the current block; this count is saved in the next *InorderBlock* entry logged.

Memory-access instructions are inserted into the TRAQ in program order when they are dispatched to the ROB. If the TRAQ is full, instruction dispatch stalls. The TRAQ also receives pipeline flush information from the processor, in order to keep its state consistent with the ROB's. Specifically, if the ROB is flushed, then the TRAQ is also flushed accordingly. This occurs, e.g., on a branch misprediction. If an individual instruction in the ROB is squashed and replayed, the TRAQ takes no action, since its entry in the TRAQ will be correctly overwritten upon the re-execution of the instruction. This occurs, e.g., when a speculative load is squashed and replayed due to memory consistency requirements.

Figure 6(b) shows a TRAQ entry. Each memory-access instruction allocates a TRAQ entry and stores the address accessed, the value read or written, and the PISN. The other two fields in a TRAQ entry are the Snoop Count and the Non-Memory Instruction (NMI) field. The former is only needed in RelaxReplay_Opt and will be discussed later. The NMI field enables RelaxReplay to log block sizes (in *InorderBlock*

entries) in number of instructions rather than in number of memory-access instructions. This support may ease replay because processors are more likely to provide interrupt support for number of instructions executed than for number of memory-access instructions executed.

The NMI field works as follows. When a memory-access instruction (*M*) is dispatched and obtains a TRAQ entry, its NMI field is set to the number of instructions dispatched since the most recent memory-access instruction. Then, when *M* reaches the TRAQ's head and is counted, the Current InorderBlock Size count is incremented by the value in the NMI field (plus one if M is not reordered).

The NMI field has a limited number of bits, which is 4 in our implementation. It is possible that more than 15 instructions appear between two consecutive memory-access instructions. In this case, RelaxReplay allocates a TRAQ entry for each group of 15 such instructions. These TRAQ entries do not correspond to any memory-access instruction, and their NMI field is set to 15.

Figure 6(c) shows the format of the different types of entries in the log record of an interval. An *InorderBlock* entry is recorded for a group of consecutive instructions to be replayed in order. It includes the value of the Current InorderBlock Size count. A *ReorderedLoad* and *ReorderedStore* entry is recorded for each reordered load and store, respectively; their fields have been discussed before. An interval may log multiple instances of each of these three entry types. Finally, when an interval ends, an *IntervalFrame* entry is logged, with the value of CISN to identify the interval. In addition, an IntervalFrame must also contain some ordering information to establish its order among all the recorded intervals. The information required depends on the particular interval-ordering mechanism used. In our case, since we use the QuickRec interval ordering, it suffices to record the current value of the Global Timestamp.

4.2 Extension for RelaxReplay_Opt

RelaxReplay_Opt tracks the coherence transactions that a processor observes between the perform and counting events of a memory-access instruction. If the address of any of them conflicts with the address accessed by the instruction, the latter is declared reordered at counting time. To track transactions, RelaxReplay_Opt adds the *Snoop Table* in the MRR (Figure 6(a)) and the *Snoop Count* in each TRAQ entry (Figure 6(b)).

The Snoop Table consists of two arrays of counters (Figure 8). When the processor observes a coherence transaction, the transaction's line address is hashed, using a different function for each array, and the resulting two counters in the arrays are incremented. We use two arrays to reduce false positives caused by aliasing. When a memory-access instruction performs, its line address is hashed, and the corresponding two counters in the Snoop Table are read. The current values of these two counters are then stored in the Snoop Count field of the TRAQ entry. Later, when the instruction is counted, if its PISN is not equal to CISN, the two counters are read again from the Snoop Table. Their current values are compared to the values saved in the Snoop Count field. If none of the counters has changed or only one has (this case is due to aliasing), the instruction is declared in order; otherwise, it is declared reordered. If it is in order, since we are moving the perform event of the instruction to the current interval, we insert the address accessed by the instruction in the read or write signature (for a load or store, respectively), to ensure proper ordering of intervals.

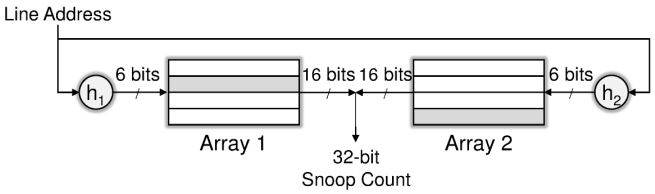


Figure 8. Snoop Table structure in RelaxReplay_Opt.

The counters are allowed to wrap around. Moreover, no action is taken when a line is evicted from the cache: there is no danger of missing a coherence transaction because, in a snoopy protocol, all caches see all the transactions.

Overall, although conservative, this design correctly detects all of the true conflicts. The only problem would be if the counter size was so small that between the perform and counting points, a counter could wrap around and reach exactly the same value. To prevent this case, we use sizable structures: two 64-entry arrays of 16-bit counters. This means that the overall Snoop Table size is 256 bytes. In addition, the Snoop Count field in each TRAQ entry is 4 bytes. For the 176-entry TRAQ that we evaluate, the combined size of all the Snoop Count fields is 704 bytes. These are minor costs for RelaxReplay_Opt's large reduction in log size and increase in replay speed (Section 5).

4.3 RelaxReplay for Directory Coherence

RelaxReplay's mechanism to track events within intervals remains unchanged for directory-based coherence, whether centralized or distributed, as long as write atomicity is guaranteed. The mechanism to order intervals may need to change, and we can use any of the proposed chunk-based MRR schemes that work for directories.

One issue that appears in directory-based protocols is that, after a dirty line is evicted from a cache, the cache is no longer able to observe coherence transactions on the line. In this case, the Snoop Table proposed in Section 4.2 for RelaxReplay_Opt would lose its ability to observe conflicting transactions. To solve this problem, when a dirty line is evicted, its address is hashed and the two corresponding counters in the Snoop Table are incremented. This ensures that any memory-access instruction that performed an access to that address but has not been counted yet, is conservatively declared reordered. Hence correctness is preserved.

4.4 Modest Hardware Complexity

RelaxReplay's hardware tracks events within intervals. Such hardware is local to the processors, rather than being distributed system-wide. Even within a processor, it leverages the well-understood general structure of the ROB. It does not require any changes to the cache coherence protocol of the machine. Moreover, its operation is independent of the scheme used to order intervals. Consequently, when RelaxReplay is paired with an interval ordering scheme that itself does not require modifications to the coherence protocol, such as QuickRec [25] or Cyrus [9], RelaxReplay provides a general MRR solution that works without coherence protocol modifications. This fact substantially lowers its hardware complexity. Finally, the resulting log can be replayed using minimal hardware support, which is very similar to existing performance counters.

5. Evaluation

5.1 Experimental Setup

We evaluate RelaxReplay with a cycle-level execution-driven simulator. We model a multicore with 4, 8 (default), or 16 cores. The cores are 4-issue out-of-order superscalars that use the RC memory model. Cores have a private L1 cache and a shared L2. The interconnection network is a ring that uses a MESI snoopy cache-coherence protocol. Table 1 shows the architectural parameters, including those of RelaxReplay. From the table, we can compute the overall size of the per-processor RelaxReplay structures. Specifically, for RelaxReplay_Base, the overall MRR module of Figure 6(a) is 2.3KB, of which the TRAQ uses 1.8KB; for RelaxReplay_Opt, the MRR is 3.3KB, of which the TRAQ uses 2.5KB. Since the processor has 2 Ld/St units, we design the TRAQ to be written twice (at perform events) and read twice (at counting events) per cycle. We design the Snoop Table to be read twice (typically at perform events) and written once (on a snoop) per cycle. We run SPLASH-2 codes [34].

The effectiveness of RelaxReplay's instruction tracking mechanism depends on the average size of the intervals. To a large extent, this size is determined by the maximum interval size chosen by the chunk-based recorder paired with RelaxReplay. Some recorders, such as Karma [3] and Cyrus [9] set the maximum interval size to a small value, in order to increase replay parallelism. Other schemes, such as CoreRacer [24] and QuickRec [25], use a very large maximum interval size because they replay sequentially, and large intervals have lower overheads. Thus, to assess the sensitivity of RelaxReplay to the maximum interval size, we evaluate RelaxReplay with two different maximum interval sizes: 4K instructions and infinitely large (INF).

To estimate replay performance, we have written a software module to control the replay according to the algorithm outlined in Section 3.5. In a real system, this module would be part of the OS. However, our execution-driven simulation setup does not run OS code. Thus, to measure the overhead of this control software, we link its code with the code of the application. With this setup, we can measure its performance cost. This control software uses the recorded total order of intervals to enforce interval ordering. For this, it uses condition variables. Then, for each interval, it follows the algorithm explained in Section 3.5: it executes the interval's *InorderBlock* blocks on the hardware, and emulates the execution of the reordered instructions.

Processor and Memory System Parameters	
Multicore	Ring-based with MESI snoopy protocol 4, 8 (default), or 16 cores
Core	4-way out-of-order superscalar @ 2GHz 176-entry ROB, 2 Ld/St units 128-entry Ld/St queue
L1 Cache	Private, 64KB, 4-way assoc, 64-entry MSHR 32B line, write-back, 2-cycle round-trip
L2 Cache	Shared, 512KB per core, 16-way assoc 64-entry MSHR 32B line, write-back, 12-cycle avg. round-trip
Ring	32B wide, 1-cycle hop delay
Memory	32B bus, 150-cycle round-trip from L2
RelaxReplay Parameters	
Read & Write Sigs.	Each: 4 × 256-bit Bloom filters with H3 hash
Glob time, Curr bl sz	64bits, 32bits
CISN, Log buffer	16bits, 8 cache lines
TRAQ	176 entries, each is 14.5B (RelaxReplay_Opt)
Snoop Table	2 arrays, 64 entries each, 16-bit entry

Table 1. Architectural parameters.

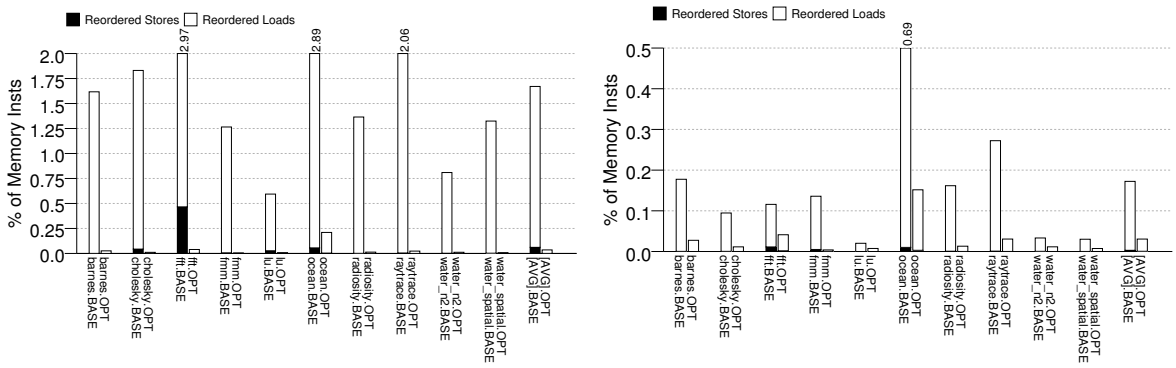
In the following, we first characterize the logs. Then, we evaluate the recording and replay performance. Finally, we analyze RelaxReplay's scalability with the processor count.

5.2 Log Characterization

We start by analyzing how many memory-access instructions are found by RelaxReplay to be reordered. Figure 9 shows the number of such instructions as a fraction of all memory-access instructions, for 4K maximum intervals (Chart (a)) and INF maximum intervals (Chart (b)). Each chart shows bars for all the applications and the average. In each case, we have bars for RelaxReplay_Base and RelaxReplay_Opt.

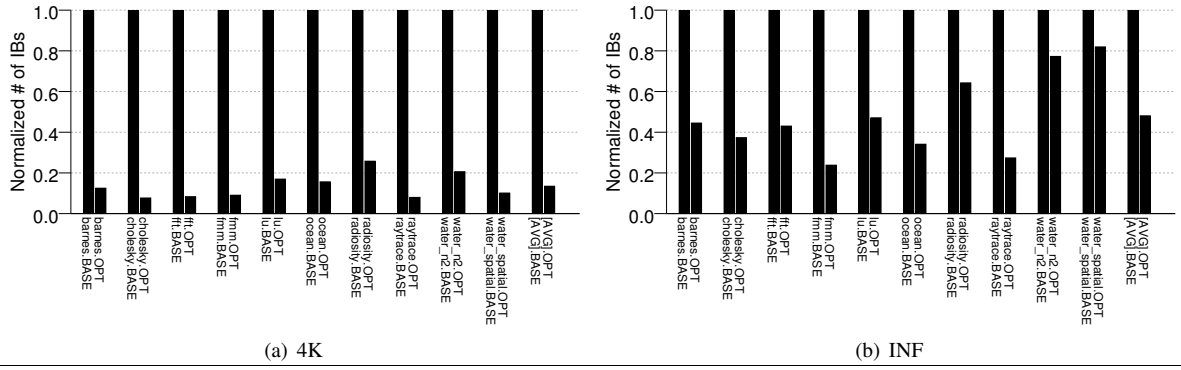
On average, RelaxReplay_Base logs 1.7% and 0.17% of memory instructions as reordered for 4K and INF intervals, respectively. This number is much smaller than the 60% of memory-access instructions that are performed out of program order, as shown in Figure 1. This shows that most of the reorders are invisible to other processors. In fact, RelaxReplay_Opt reduces this fraction even more — to a minuscule 0.03% for both 4K and INF intervals. As we will see, this large reduction has a significant impact on the size of the generated logs and the replay speed.

In all cases, loads dominate the reordered instructions. Comparing the 4K and INF results, we see that larger intervals help RelaxReplay_Base reduce the fraction of reordered instructions. However, RelaxReplay_Opt's effectiveness is independent of the interval size. This is because RelaxReplay_Opt relies on the Snoop Table to detect reordered



(a) 4K (b) INF

Figure 9. Fraction of memory-access instructions found by RelaxReplay to be reordered for 4K (a) and INF (b) intervals.



(a) 4K (b) INF

Figure 10. Number of *InorderBlock* entries (IBs), normalized to RelaxReplay_Base, for 4K (a) and INF (b) intervals.

instructions, rather than on whether perform and counting events are in the same interval.

The number of reordered instructions affects the number and size of the *InorderBlock* entries in the logs. Recall that an *InorderBlock* entry corresponds to a set of consecutive in-order instructions. An *InorderBlock* is terminated by either a reordered access or an interval termination. Hence, if an optimization such as RelaxReplay_Opt reduces the number of reordered accesses, the *InorderBlock* size increases and the number of *InorderBlock* entries goes down.

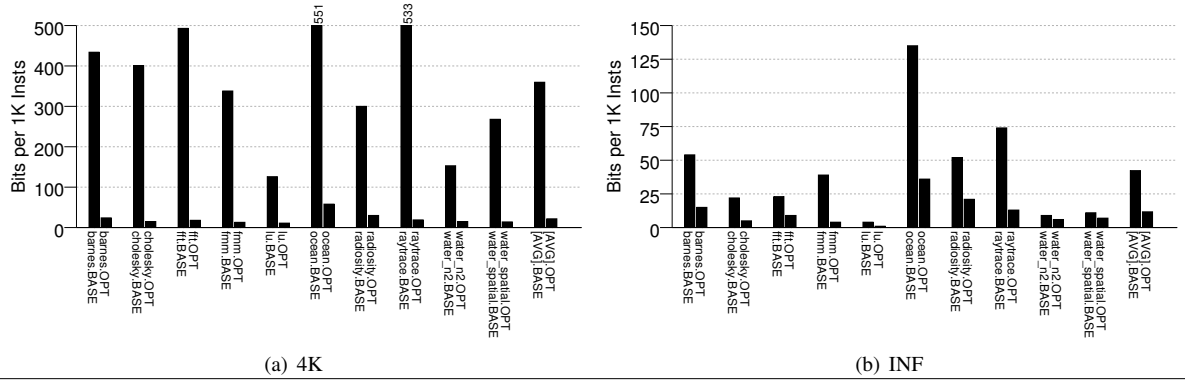
Figure 10 shows the number of *InorderBlock* entries in the logs for 4K (Chart (a)) and INF (Chart (b)) intervals. The figure is organized as the previous one except that, in each application, the bars are normalized to RelaxReplay_Base. The figure shows that RelaxReplay_Opt's ability to reduce the number of reordered access results in many fewer *InorderBlocks*. On average, it only logs 13% and 48% as many *InorderBlocks* as RelaxReplay_Base for 4K and INF intervals, respectively.

Finally, Figure 11 shows the *uncompressed* log size for the two designs, in bits per 1K instructions for 4K (Chart (a)) and INF (Chart (b)) intervals. We see that RelaxReplay_Opt reduces the log size over RelaxReplay_Base substantially — the result of its ability to reduce the number of reordered instructions. For 4K intervals, the average log size per 1K instructions goes down from 360 bits in RelaxReplay_Base to 22 bits in RelaxReplay_Opt; for INF intervals, it goes down from 42 bits to 12 bits. These are substantial reductions in logging needs.

The resulting RelaxReplay_Opt log sizes are 1–4x the log sizes reported for previous chunk-based recorders [6, 9, 10, 18, 23, 24] and are, therefore, comparable to them. This is despite the fact that the previous schemes required the strict SC or TSO models, while RelaxReplay_Opt handles the relaxed RC model. In fact, RelaxReplay_Opt's logs are quite small compared to the several GB/s of memory bandwidth available in modern machines. Indeed, in our experiments, RelaxReplay_Opt generates on average only 48 MB/s and 25 MB/s of logging state for 4K and INF intervals, respectively, which is a small rate. On the other hand, RelaxReplay_Base generates on average 840 MB/s and 90 MB/s. Although we consider the former excessive, the latter is small and shows that the simpler RelaxReplay_Base design is a viable solution if large intervals are acceptable — i.e., when replay parallelism is not required.

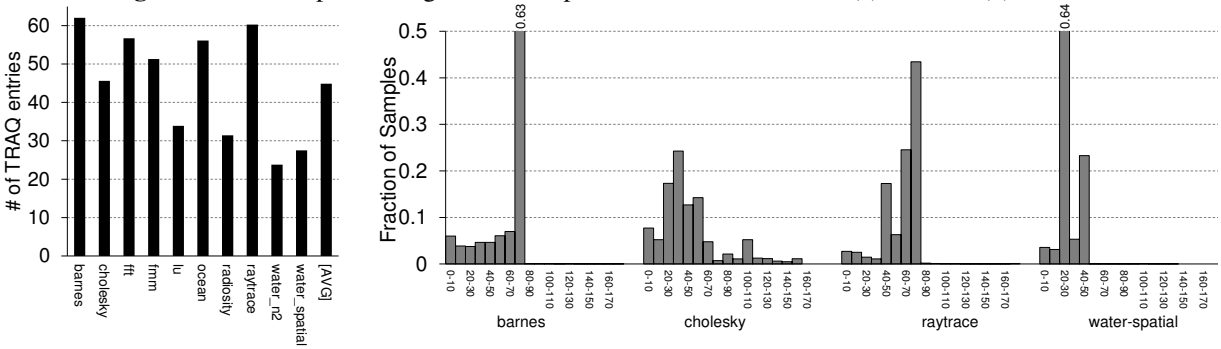
5.3 Characterization of Recording Performance

It can be shown that the execution overhead of recording under RelaxReplay_Opt, or under RelaxReplay_Base with INF intervals is negligible. This is consistent with past proposals for hardware-assisted RnR. To understand why recording overhead is negligible in these cases, consider the two main sources of overhead: memory bus contention as the log is being saved, and stalls due to lack of TRAQ entries. From the previous section, we deduce that the induced memory bus contention is negligible for RelaxReplay_Opt and for RelaxReplay_Base with INF intervals.



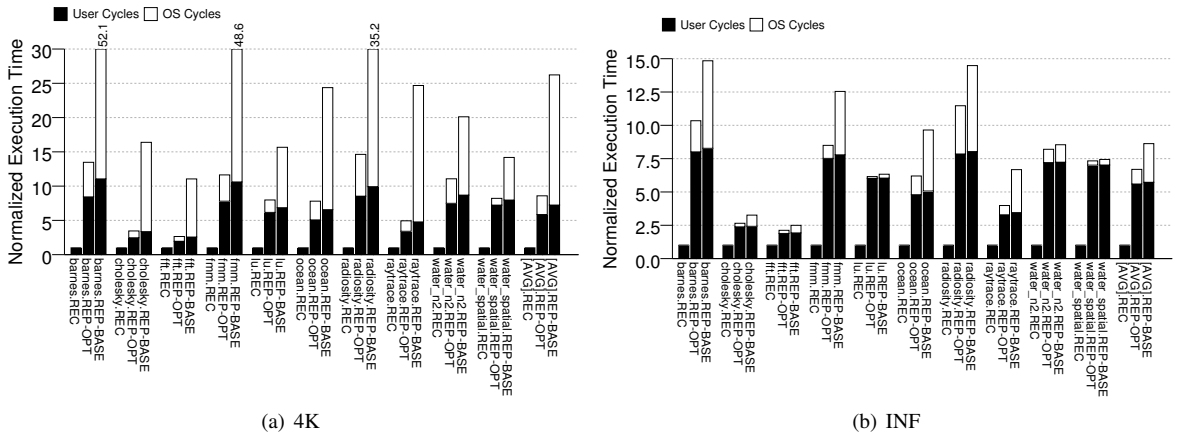
(a) 4K (b) INF

Figure 11. Uncompressed log size in bits per 1K instructions for 4K (a) and INF (b) intervals.



(a) Average TRAQ utilization (b) TRAQ utilization histograms

Figure 12. TRAQ utilization: average (a) and histograms for four representative applications (b).



(a) 4K (b) INF

Figure 13. Replay time with Opt or Base logs, normalized to recording time, for 4K (a) and INF (b) intervals.

To assess the TRAQ stall, Figure 12 shows the TRAQ utilization. The figure applies to both RelaxReplay_Opt and RelaxReplay_Base. In Chart (a), we show the average number of TRAQ entries utilized by each application. We see that, in all cases, this number is less than 64. This is a small number compared to the TRAQ's 176 entries. In Chart (b), we show distributions of number of used TRAQ entries for four representative applications. In the figure, each bar corresponds to the fraction of samples where a certain number of entries (grouped in bins of 10) were used. As can be seen, although different applications have different overall shapes, in all cases, most of the time around 80 or fewer entries are used. Hence, TRAQ-induced stall is very rare. It can be shown that it accounts for less than 0.3% of the execution time for RelaxReplay_Opt and for RelaxReplay_Base with INF intervals.

5.4 Characterization of Replay Performance

Figure 13 shows the time it takes to replay the applications with RelaxReplay_Opt logs or RelaxReplay_Base logs, for 4K (Chart (a)) and INF (Chart (b)) intervals. For each application, the times are normalized to the time it takes to record the application, shown as the leftmost bar of each group. Note that, while recording was done in parallel with 8 processors, replay in these experiments is performed sequentially — because the interval-ordering mechanism records a total order of intervals. Moreover, the replay time is broken down

into execution of the application (*User Cycles*) and execution of our control module that emulates the OS (*OS Cycles*). The latter orders intervals, reads log entries, and emulates reordered instructions.

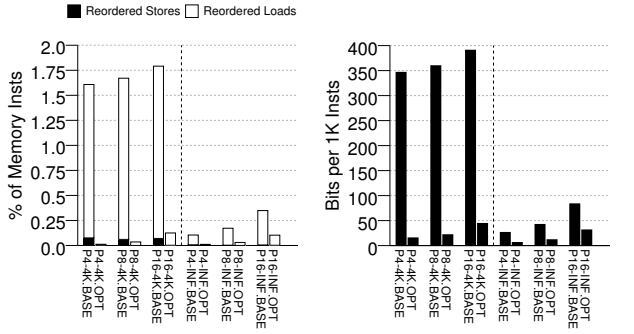
From the figure, we see that replaying with the RelaxReplay_Opt log is fast. Although the replay is performed sequentially, it takes on average only 8.5x and 6.7x as long as the parallel recording for the 4K and INF intervals, respectively. OS time is about a third to a sixth of the replay time.

Replay with the RelaxReplay_Base log is a bit slower for INF intervals and substantially slower for 4K intervals. Specifically, sequential replay takes on average 8.6x and 26.2x as long as the parallel recording for the INF and 4K intervals, respectively. The slowdown is due to the larger fraction of reordered instructions. There is a substantial fraction of OS cycles, as the OS deals with reordered instructions. User cycles are sometimes higher than in the RelaxReplay_Opt bars because there are more pipeline flushes, as end-of-block interrupts transfer execution to the control module.

Overall, using the RelaxReplay_Opt log or the RelaxReplay_Base log with INF intervals ensures efficient replay. If we combine them with interval ordering schemes that admit parallel replay [3, 9], we expect substantially faster replay.

5.5 Scalability Analysis

To analyze RelaxReplay's scalability with the number of processors, we repeat the experiments with 4- and 16-core machine configurations. Figure 14 shows how the fraction of memory-access instructions that RelaxReplay perceives as reordered (Chart (a)) and the log generation rate (Chart (b)) change with 4, 8, and 16 processors (*P4*, *P8*, and *P16*) for RelaxReplay_Base and RelaxReplay_Opt. In each figure, the left- and right-hand sides present the results for the 4K and INF configurations, respectively. Each bar is the average of all the applications.



(a) Reordered instructions (b) Log size

Figure 14. The effect of processor count on recording.

The figures show that both the fraction of reordered instructions and the log size increase with the number of processors. Leaving aside the case of RelaxReplay_Base with 4K intervals, we see that both instruction reordering and log size are still small for up to 16 processors but increase noticeably, although not exponentially.

The reason for the increase is that, with more cores, we have more traffic and, in particular, more coherence traffic. Moreover, in our ring-based snoopy protocol, all processors observe all the traffic. As a result, there is more chance for false positives in the signatures and in the Snoop Table. The former causes additional terminations of intervals, which results in bigger logs and, in RelaxReplay_Base, more reordering; the latter causes RelaxReplay_Opt to count more reordered instructions. As a result, we see fast increases in both parameters. With directory coherence, we expect lower growth rates, as each core only sees coherence messages for the cache lines it accessed.

The case of RelaxReplay_Base with 4K intervals is less sensitive to the number of cores. The reason is that its behavior is largely determined by the small maximum interval size. Adding more coherence transactions only has a marginal impact in further reducing the interval sizes.

6. Related Work

Some of the proposals for RnR of multithreaded programs on multiprocessors do not require any special hardware and, instead, rely on the OS, compiler and/or runtime libraries for RnR (e.g., [16, 22, 32]). MRR with these techniques has a relatively-high performance overhead, perturbs the timing of parallel execution, and typically requires modified binaries. Due to space constraints, we do not discuss them.

Hardware-assisted MRR techniques use special hardware support for recording memory-access interleaving. FDR [35] and RTR [36] record dependences between pairs of communicating instructions. This can result in large log sizes or requires expensive hardware data structures to reduce the log size. Also, the resulting fine-grain ordering constraints can hurt replay efficiency. While FDR only supports SC, RTR supports TSO by recording the value of loads that may violate SC. Similar to our approach, it records a load's value if there is a conflicting access to its memory location between the time the load performs and all of its predecessors perform.

To remedy the large log size and fine-grain ordering constraints of earlier designs, chunk-based schemes were proposed. DeLorean [18] and Capo [19] use the speculative multithreading hardware of Bulk [31]. The underlying hardware enforces SC while allowing aggressive out-of-order execution of instructions. The execution is recorded by logging the order in which processors commit their chunks.

Rerun [10] and Karma [3] are chunk-based techniques for conventional multiprocessors with directory coherence. The papers also include proposals to integrate RTR's solution for TSO recording with their chunk-based schemes. As such, they can be considered as chunk-based recorders for TSO. However, they only provide high-level discussions about how the integration could be done without providing detailed designs or results. Timetraveler [33] builds on Rerun and reduces its log size. While Rerun terminates a chunk upon the first conflicting coherence transaction, Timetraveler allows

the chunk to grow beyond that, to reduce the chunk count and, thus, the log size.

Intel MRR [23], Cyrus [9] and CoreRacer [24] are chunk-based recorders for snoopy protocols. While the first two assume SC, CoreRacer supports TSO by recording the number of stores pending in the processor's write buffer when a chunk terminates. CoreRacer can then correctly account for reordered and forwarded loads by simulating the write buffer's content during replay. However, replay efficiency may suffer because it requires write buffer simulation. Also, recording the pending stores for a chunk is often unnecessary, since the resulting reordering is rarely visible to other processors. QuickRec [25] is an FPGA implementation of a similar approach.

Some schemes have described parallel replay, such as DeLorean [18] and Capo [19] (both through speculative execution), Karma [3] and Cyrus [9].

LReplay [6] does not monitor coherence transactions; it includes a non-trivial centralized module that directly tracks the memory accesses performed by all cores. It relies on this module to detect inter-processor dependences. It supports TSO using RTR's approach. Due to its recording technique, its replay algorithm needs to simulate all instructions.

Strata [21] is designed for SC machines and records a stratum (i.e., one chunk for each processor in the system) when a dependence occurs between any two processors. A stratum is recorded only if the source and destination instructions are not already separated by a stratum. By replaying one stratum at a time, all data dependences can be enforced.

Lee et al. [15, 17] use off-line search to infer inter-thread dependences for SC [15] and TSO [17] executions. They do not record dependences. Instead, they log data fetched on a cache miss (when it is accessed for the first time). This allows independent replay of each thread. They also periodically record some Strata hints to speed-up the off-line search. Using the per-thread replays and the hints, inter-thread data dependences are determined off-line.

Rainbow [27] builds on the sequentially-consistent Strata and improves it in two ways. Firstly, it uses a centralized hardware structure, called Spectrum History, to reduce the number of recorded strata, and thus, improve the log size and replay speed. Secondly, it shows that the same data structure can aid in detecting potential SC violations in order to record non-SC executions. The idea is to record some information about delayed and pending instructions that allows it to replay the situation correctly when an SC violation happens. Although the paper discusses the hardware structures and algorithms required to implement the first improvement fairly extensively, the second part is only explained vaguely and at a very high level. In particular, detailed record and replay algorithms are only presented for the first contribution; the paper does not explain the mechanisms or hardware structures required to track and communicate the pending and delayed instructions that are central to its second contribution.

It is difficult to provide a detailed comparison between Rainbow and RelaxReplay, especially in terms of the hardware data structures, given Rainbow's lack of details and the complexity of its proposed SC-violation handling mechanism. However, it is clear that, similar to RelaxReplay, Rainbow requires write atomicity, since each instruction can only be recorded in a single spectrum. Unlike RelaxReplay, however, Rainbow cannot accommodate distributed directory protocols due to its centralized Spectrum History design. Also, unlike RelaxReplay, it needs to augment the coherence protocol messages and/or add new ones (if write-after-read dependences are to be explicitly recorded). In addition, it is unclear how the Spectrum History can be virtualized in order to accommodate application-level RnR (i.e., recording single applications instead of whole machines). Moreover, unlike RelaxReplay that can be applied to any chunk-based MRR scheme, the SC-violation handling mechanism of Rainbow is particular to its Strata-like design and cannot be directly used in conjunction with other MRR schemes.

7. Concluding Remarks

This paper proposed *RelaxReplay*, the first complete solution for hardware-assisted MRR that works for any relaxed-consistency model with write atomicity. With *RelaxReplay*, we can build an RnR system for any coherence protocol — whether snoopy, centralized directory or distributed directory. RelaxReplay's insight is a new way to capture memory-access reordering. Each memory instruction goes through a post-completion in-order counting step that detects any reordering, and efficiently records it. We presented two designs with different hardware needs, log sizes, and replay speeds.

RelaxReplay's mechanism to capture memory-access reordering does not rely on consistency model specifications. It can be combined with the chunk-ordering algorithm of any existing chunk-based MRR proposal — enabling that proposal to record relaxed-consistency executions. Its hardware is local to the cores and does not change the cache coherence protocol. Finally, it produces a log that is compact and efficient to use for replay with minimal hardware support.

We evaluated RelaxReplay with simulations of an 8-processor RC multicore running SPLASH-2 applications. The results showed that RelaxReplay induces negligible overhead during recording. In addition, the average size of its log was 1–4x the log sizes reported for existing SC- or TSO-based MRR systems, and still a small fraction of the bandwidth of current processors. Finally, deterministic replay is efficient, since the sequential replay of these 8 processors with minimal hardware support took on average only 6.7x as long as the parallel recording.

Acknowledgments

This work was supported in part by NSF under grants CCF-1012759 and CNS-1116237, and Intel under the Illinois-Intel Parallelism Center (I2PC).

References

[1] H. Agrawal, R. A. DeMillo, and E. H. Spafford. An Execution-Backtracking Approach to Debugging. *IEEE Software*, 8(3), May 1991.

[2] ARM. *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R Edition Issue C*, July 2012.

[3] A. Basu, J. Bobba, and M. D. Hill. Karma: Scalable Deterministic Record-Replay. In *ICS*, June 2011.

[4] B. H. Bloom. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *Comm. of the ACM*, 11(7), July 1970.

[5] T. Bressoud and F. Schneider. Hypervisor-Based Fault-Tolerance. *ACM TOCS*, 14(1), February 1996.

[6] Y. Chen, W. Hu, T. Chen, and R. Wu. LReplay: A Pending Period Based Deterministic Replay Scheme. In *ISCA*, June 2010.

[7] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High Availability via Asynchronous Virtual Machine Replication. In *NSDI*, April 2008.

[8] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *OSDI*, December 2002.

[9] N. Honarmand, N. Dautenhahn, J. Torrellas, S. T. King, G. Pokam, and C. Pereira. Cyrus: Unintrusive Application-Level Record-Replay for Replay Parallelism. In *ASPLOS*, March 2013.

[10] D. R. Hower and M. D. Hill. Rerun: Exploiting Episodes for Lightweight Memory Race Recording. In *ISCA*, June 2008.

[11] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen. Detecting Past and Present Intrusions Through Vulnerability-Specific Predicates. In *SOSP*, October 2005.

[12] S. T. King and P. M. Chen. Backtracking Intrusions. In *SOSP*, October 2003.

[13] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging Operating Systems with Time-Traveling Virtual Machines. In *USENIX Ann. Tech. Conf.*, April 2005.

[14] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.*, 28(9), September 1979.

[15] D. Lee, M. Said, S. Narayanasamy, Z. Yang, and C. Pereira. Offline Symbolic Analysis for Multi-Processor Execution Replay. In *MICRO*, December 2009.

[16] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. M. Chen, and J. Flinn. Respec: Efficient Online Multiprocessor Replay via Speculation and External Determinism. In *ASPLOS*, March 2010.

[17] D. Lee, M. Said, S. Narayanasamy, and Z. Yang. Offline Symbolic Analysis to Infer Total Store Order. In *HPCA*, February 2011.

[18] P. Montesinos, L. Ceze, and J. Torrellas. DeLorean: Recording and Deterministically Replaying Shared-Memory Multiprocessor Execution Efficiently. In *ISCA*, June 2008.

[19] P. Montesinos, M. Hicks, S. T. King, and J. Torrellas. Capo: A Software-Hardware Interface for Practical Deterministic Multiprocessor Replay. In *ASPLOS*, March 2009.

[20] S. Narayanasamy, G. Pokam, and B. Calder. BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging. In *ISCA*, June 2005.

[21] S. Narayanasamy, C. Pereira, and B. Calder. Recording Shared Memory Dependencies Using Strata. In *ASPLOS*, Oct 2006.

[22] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. PRES: Probabilistic Replay with Execution Sketching on Multiprocessors. In *SOSP*, October 2009.

[23] G. Pokam, C. Pereira, K. Danne, R. Kassa, and A.-R. Adl-Tabatabai. Architecting a Chunk-Based Memory Race Recorder in Modern CMPs. In *MICRO*, December 2009.

[24] G. Pokam, C. Pereira, S. Hu, A.-R. Adl-Tabatabai, J. Gottschlich, H. Jungwoo, and Y. Wu. CoreRacer: A Practical Memory Race Recorder for Multicore x86 TSO Processors. In *MICRO*, December 2011.

[25] G. Pokam, K. Danne, C. Pereira, R. Kassa, T. Kranich, S. Hu, J. Gottschlich, N. Honarmand, N. Dautenhahn, S. T. King, and J. Torrellas. QuickRec: Prototyping an Intel Architecture Extension for Record and Replay of Multithreaded Programs. In *ISCA*, June 2013.

[26] Power.org. *Power ISA™ Version 2.06 Revision B*, July 2010.

[27] X. Qian, H. Huang, B. Sahelices, and D. Qian. Rainbow: Efficient Memory Dependence Recording with High Replay Parallelism for Relaxed Memory Model. In *HPCA*, Feb 2013.

[28] D. J. Sorin, M. D. Hill, and D. A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool Publishers, 1st edition, 2011.

[29] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou. Flashback: A Lightweight Extension for Rollback and Deterministic Replay for Software Debugging. In *USENIX Ann. Tech. Conf.*, June 2004.

[30] Tilera. *Tile Processor User Architecture Manual Rel. 2.4*, November 2011.

[31] J. Torrellas, L. Ceze, J. Tuck, C. Cascaval, P. Montesinos, W. Ahn, and M. Prvulovic. The Bulk Multicore Architecture for Improved Programmability. *Comm. of the ACM*, 52(12), 2009.

[32] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. DoublePlay: Parallelizing Sequential Logging and Replay. In *ASPLOS*, March 2011.

[33] G. Voskuilen, F. Ahmad, and T. N. Vijaykumar. Timetraveler: Exploiting Acyclic Races for Optimizing Memory Race Recording. In *ISCA*, June 2010.

[34] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *ISCA*, June 1995.

[35] M. Xu, R. Bodik, and M. D. Hill. A "Flight Data Recorder" for Enabling Full-System Multiprocessor Deterministic Replay. In *ISCA*, June 2003.

[36] M. Xu, R. Bodik, and M. D. Hill. A Regulated Transitive Reduction (RTR) for Longer Memory Race Recording. In *ASPLOS*, 2006.