Appears in Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)
San Jose, California, USA, October 21-25, 2006.

A Regulated Transitive Reduction (RTR) for Longer Memory Race Recording

Min Xu[†]
Electrical and Computer Engr. Dept.
University of Wisconsin-Madison
mxu@cae.wisc.edu

Rastislav Bodík
Computer Science Division, EECS
University of California, Berkeley
bodik@eecs.berkeley.edu

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

Abstract

Multithreaded deterministic replay has important applications in cyclic debugging, fault tolerance and intrusion analysis. Memory race recording is a key technology for multithreaded deterministic replay. In this paper, we considerably improve our previous always-on Flight Data Recorder (FDR) in four ways:

- **Longer recording** by reducing the log size growth rate to approximately one byte per thousand dynamic instructions.
- **Lower hardware cost** by reducing the cost to 24 KB per processor core.
- **Simpler design** by modifying only the cache coherence protocol, but not the cache.
- **Broader applicability** by supporting both Sequential Consistency (SC) and Total Store Order (TSO) memory consistency models (existing recorders support only SC).

These improvements stem from several ideas: (1) a **Regulated Transitive Reduction (RTR)** recording algorithm that creates stricter and vectorizable dependencies to reduce the log growth rate; (2) a **Set/LRU** timestamp approximation method that better approximates timestamps of uncached memory locations to reduce the hardware cost; (3) an **order-value-hybrid** recording method that explicitly logs the value of potential SC-violating load instructions to support multiprocessor systems with TSO.

Categories and Subject Descriptors. C.4 [**Computer Systems Organization**]: Performance of Systems — *Measurement Techniques*; D.2.5 [**Software Engineering**]: Testing and Debugging — *Debugging Aids*;

General Terms. Algorithms, Measurement, Reliability

Keywords. Multithreading, Determinism, Race Recording

1. Introduction

Deterministic replay of multithreaded programs has several important uses. First, determinism can help developers effectively debug multithreaded programs using cyclic debugging [23] because the erroneous executions can be repeated. Furthermore, determinism is also necessary in fault detection [30], fault recovery [15], and replay-based intrusion analysis [8].

To faithfully replay a multithreaded execution, we need to replay the following information: (1) program *initial states*; (2) program *inputs*; and (3) *memory races* among threads. A recorder records these types of information through three mechanisms: *checkpointing*, *input logging* and *memory race recording*.

Existing software-based recorders are often limited to offline uses due to their prohibitive runtime overheads. As transistors get cheaper, spending a small amount of chip area on a hardware-based, low-overhead recorder is becoming more economical. In a previous paper, we proposed a hardware-based *Flight Data Recorder (FDR)* [33], which includes checkpointing, input logging, and memory race recording with little runtime overhead. In this paper, we improve FDR's memory race recording with significant reduction in the log size and hardware cost. We focus on memory race recording for three reasons.

1) Race recording limited FDR. The log size of FDR's memory race recorder is approximately 2 MB/1GHz-processor/second (compressed). The size of both the memory race log and the checkpoint log limits FDR to only one second of recording, which may be sufficient for debugging [20]. However, other applications may need much longer recording [8, 30]. To do that, we need to reduce the memory race log and the checkpoint log. It is relatively easy to reduce checkpoint log because it can be eliminated (if one records from the beginning of an execution) or amortized by longer checkpoint intervals (the checkpoint size is a constant for each checkpoint interval). Therefore, it is more important to reduce the memory race log. Furthermore, FDR's race recorder requires the sequential consistency (SC) memory model, which is supported by a limited subset of existing multiprocessor systems. Therefore, it is important to develop a new race recorder that supports more memory models.
2) Race recording requires non-trivial hardware. FDR's memory race recorder is integrated with the memory caches. This adds undesirable complexity [22] to the caches, which are performance-critical structures. FDR increases the chip area of the caches by 6.25% (or 256 KB for a 4 MB L2). A hardware race recorder is more attractive if its hardware cost is reduced. Therefore, it is important to develop a new recorder that has lower complexity and requires smaller chip area.
3) After BugNet [20], race recording is a priority. BugNet is another recorder, which improves FDR's checkpointing and input logging. By logging first-time-load-values, BugNet reduces the log size of checkpointing and input logging, as well as the hardware cost (reduced to 16 KB per processor). Considering these improvements, memory race recording is likely a bottleneck for future recorders.

This paper presents several improvements over FDR.

Regulated Transitive Reduction (RTR). Netzer's *Transitive Reduction (TR)* [21], generates the most compact log among existing partial order race recording algorithms. TR logs only conflicts. We show that further reduction of the log size is possible if we judiciously log *stricter and vectorizable dependencies*, which are not necessarily conflicts. We propose a new partial order

[†] Now at VMware.

ASPLOS'06 October 21–25, 2006, San Jose, California, USA.

recording algorithm—*Regulated Transitive Reduction (RTR)*. RTR logs *stricter dependencies*, which are stricter because enforcing them is sufficient but not necessary for faithful replay. RTR creates stricter dependencies so that a large number of dependencies are *vectorizable*. Vectorized dependencies are reminiscent to vectorized computations—one type of computation is performed on multiple data. This vectorization compacts the log. We show that RTR is better than TR and reduces the log size by **28%** on average (Section 10.2.2).

Set/LRU Timestamp Approximation. In order to find those dynamic instructions that race with each other, Netzer's recorder incurs non-trivial hardware cost by storing the last read/write "timestamps" for each memory block [21]. FDR reduces this hardware cost by storing only those timestamps of cached blocks and approximates the timestamp if a block is uncached. FDR's approximation method often causes a false race to be logged whenever a block is missed in the cache, *i.e.*, FDR's log growth rate goes up as the hardware cost (timestamp memory size) goes down. In this paper, we propose a new timestamp approximation method, which *simultaneously* reduces the log size and the hardware cost. This new method approximates the missing timestamps using the Least-Recently-Used (LRU) block's timestamp within the same associative set. We call this *Set/LRU Timestamp Approximation*, which reduces the hardware cost to **24 KB** per core (Section 10.2.1) and, together with RTR, reduces the log size by **96%** over FDR (Section 10.2.2).

Decoupled Timestamp Memory. FDR couples timestamps with cache blocks. This can introduce much design complexity to the memory caches, which are performance-critical components. We move timestamps out of cache (decoupling), which allows us to independently size the timestamp memory and potentially reduce the overall design complexity (Section 10.2.1).

Support TSO Consistency Model. When used with the Total-Store-Order (TSO) memory consistency model, existing race recorders can run into replay deadlocks because they assume Sequential Consistency (SC). We propose a new *order-value-hybrid* recording method to support TSO (which is x86-like). The hybrid method avoids replay deadlocks and deterministically replays TSO executions by recording additional information of the *load values* when load instructions potentially violate the SC ordering (Section 6).

We make the following contributions in this paper.

1) We improve FDR's race recording algorithm, through RTR and Set/LRU. The new algorithm enables significant log size and hardware cost reductions.
2) We improve a race recorder implementation by lowering the design complexity and supporting TSO.
3) We design and evaluate a race recorder on a four-way Chip MultiProcessor (CMP) system using full-system simulations and commercial workloads.

Next, we review race recorders in Section 2. We present the new recording algorithm with small log size and low hardware cost in Sections 3-4. We propose moving the timestamps out of processor caches and present a method to support TSO in Sections 5-7. We describe a concrete recorder: RTR/CMP, a CMP-based hardware race recorder (Section 8-9). We evaluate RTR/CMP and explore a design space in Section 10. We discuss related work in Section 11 and conclude in Section 12.

2. Background: Hardware Race Recorders

Conflicts and Memory Races. In multithreaded executions, two memory accesses *conflict* if and only if they are executed by different threads, access the same memory location, and at least one of them is a write. Therefore, three types of conflicts exist: read-after-write (RAW), write-after-read (WAR) and write-after-write (WAW). Herein, we assume programs are divided into RISC-like instructions and we label a dynamic instruction with a tuple `TID:IC`. `TID` is the thread ID of the thread that executes a dynamic instruction. *Dynamic Instruction Count (*`IC`*)* is a monotonically increasing number that we give to each dynamic instruction executed by a thread in the program order. A conflict is denoted `i:x`→`j:y`, where → denotes the order of the conflict.

Informally, *memory races* (or simply, *races*) are those conflicts in which the instruction order is not determined by the program. Because all races are conflicts (the converse is not true), it is sufficient to record all conflicts in order to record races. But, why do we want to record races?

Figure 1a shows an example conflict (`i:2`→`j:3`*)*, which happens to be a race. Two threads `i` and `j` both write a variable `z`. The ICs are shown in the numbered circles. The final value of `z` is determined by which thread "loses" the race. In general, the *nondeterminism* caused by races can prevent multithreaded executions from being deterministically replayed.

Race Recorders. In order to deal with this nondeterminism, researchers have proposed to use race recorders to enable faithful replay [23]. The goal of race recorders is to record the races that occur in the original executions and use the recorded information to enable deterministic replay. Existing recorders assume SC, as do we until we generalize our recorder to TSO in Section 6.

FDR. In 2003, we proposed a hardware recorder called Flight Data Recorder (FDR) [33]. FDR augments each cache block with a timestamp, which is the IC of the instruction that last accessed the cache block. Then, by piggybacking the timestamp on cache coherence messages, FDR can detect and log conflicts on-the-fly.

For example, Figure 1b shows how the conflict (`i:2`→`j:3`) in Figure 1a is detected by FDR. The two threads (`i` and `j`) execute on two processors (P1 and P2). The variable `z` can be cached in the processor caches (denoted by rectangles). We illustrate each step in the conflict detection using a dark numbered circle. First, after P1 writes `z` (IC_{P1}=2), the cache block is in the Modified (M) state in P1's cache. P1 also timestamps the block with the IC of the write instruction. When P2 attempts to write `z` (IC_{P2}=3), it issues a

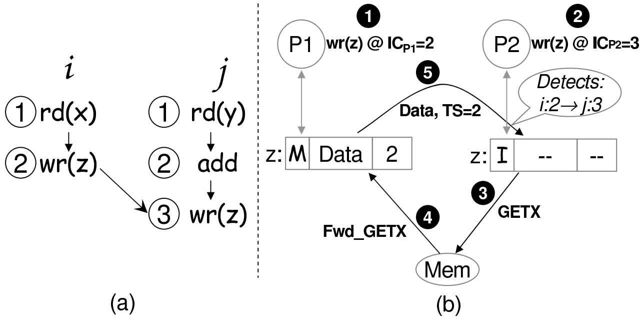


Figure 1: In (a), a multithreaded execution is shown. In (b), FDR's hardware race recorder detects the race. The notations `rd(x)` and `wr(x)` denote read and write to memory location `x`.

GETX (get exclusive) request to the directory at the memory, because the block is in the Invalid (I) state in P2's cache. The directory then forwards the GETX to the block's current owner (P1). Upon receiving the forwarded request, P1 sends the data block and the timestamp (TS=2) to P2. Finally, FDR detects this conflict when P2 receives P1's message.

3. Reducing Log Size Using RTR

To ensure deterministic replay, it is sufficient to log every conflict. We call the log the *Dependence Log* because the logged conflicts serve as interthread dependencies, which need to be enforced during replay to resolve the races. However, on modern machines with fast interthread communication, conflicts can happen frequently, which causes this log-them-all method to suffer from generating huge logs. This section presents a log size reduction algorithm, which is illustrated through an example in Figure 2. To start, Figure 2a shows an execution with five conflicts. Assuming each IC is 64-bit wide, the log-them-all method generates a dependence log of five dependencies.

3.1. Netzer's Transitive Reduction (TR)

One way to reduce the dependence log is to record only a *subset* of the conflicts. Netzer observed that some conflicts are transitively implied by others and he proposed the *Transitive Reduction (TR)* method to reduce the number of conflicts to be recorded [21]. Figure 2b shows the conflict i:1→j:4 is implied by the conflict i:2→j:3, because i:1, i:2 and j:3, j:4 are in program order. Enforcing i:2→j:3 during a replay execution enforces i:1→j:4. Therefore, conflict i:1→j:4 is removed from the log. After this reduction, the log is reduced to four dependencies. In practice, Netzer showed that TR reduces the log size significantly (82% - 99.8%) over the log-them-all method [21].

3.2. Regulated Transitive Reduction (RTR)

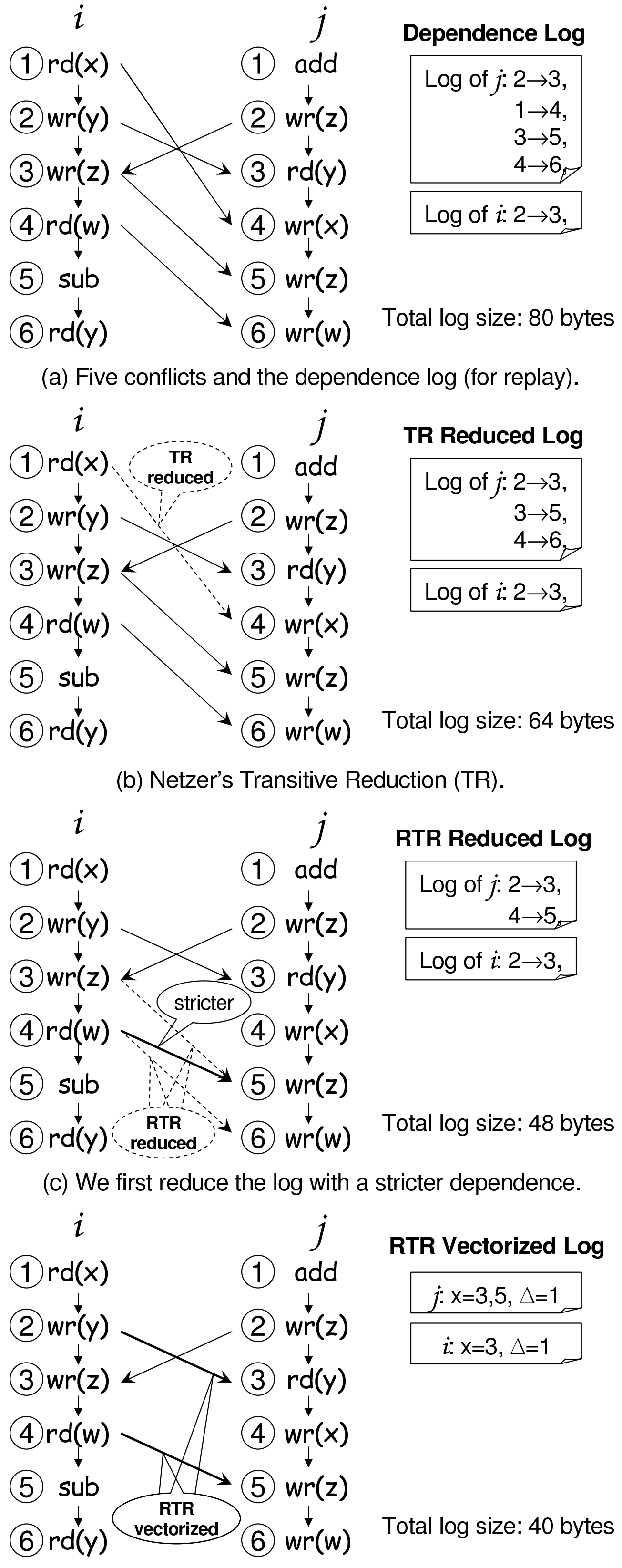
In this paper, we go beyond recording a subset of the conflicts and find that *actively creating* artificial dependencies can further reduce the log. Our recorder creates *stricter* and *vectorized* dependencies in the log. The new algorithm is named *Regulated Transitive Reduction (RTR)*, because it regulates the replay.

3.2.1 Log Size Reduction

Stricter Dependencies. Figure 2c shows how RTR enables more transitive reduction by creating artificial dependencies in the log. In the recording, after the conflict i:3→j:5 is detected, RTR writes a stricter dependence i:4→j:5 in the log. The dependence is stricter than i:3→j:5 because enforcing it during replay is sufficient but not necessary to satisfy i:3→j:5. As a result of this stricter dependence, i:3→j:5 and i:4→j:6 are transitively reduced. Thus, RTR reduces the log to three conflicts. This is not possible if we log only a subset of the original conflicts.

Vectorized Dependencies. But why does RTR decide to create i:4→j:5 out of many possibilities? RTR creates i:4→j:5 because this dependence and the previous dependence (i:2→j:3) are *vectorizable*. Dependencies are vectorizable if and only if the dependencies have the same *IC stride* (herein denoted by Δ). As we show visually in Figure 2d, i:4→j:5 and i:2→j:3 both have their IC stride equal to one. Vectorized dependencies help reduce the log with a compact format:

$$\{(x\text{-}\Delta)\rightarrow x \mid x=\{3,5\}, \Delta=1\}$$



(a) Five conflicts and the dependence log (for replay).

(b) Netzer's Transitive Reduction (TR).

(c) We first reduce the log with a stricter dependence.

(d) We also reduce the log with vectorized dependencies.

Figure 2: Four steps in reducing the dependence log: log-them-all; TR; stricter dependencies; vectorizing.

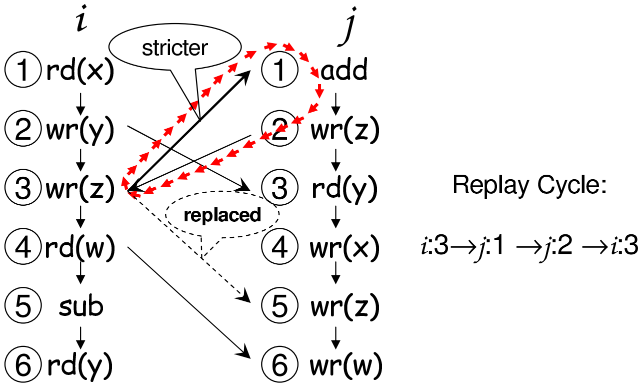


Figure 3: An overly strict dependence causes a deadlock.

where (*x*-Δ)→*x* is the template of the vectorized dependencies; *x* is substituted with a set of logged ICs (*i.e.*, {3,5}) during replay.

With both stricter and vectorized dependencies, RTR enables a greater log size reduction over TR. In practice, about 1/3 of the reduction is from transitivity of the stricter dependencies and 2/3 of the reduction is from the vectorized compact format.

The vectorized dependencies work by exploiting the regularity within multithreaded executions, *e.g.*, once a thread loses in a race, the loser tends to lose the next race because the winner and the loser spend a similar amount of time computing before the next race. Thus, the two races tend to be vectorizable.

3.2.2 *Replay Correctness and Performance*

Correctness. For correctness, we consider `i:v`→`j:w` stricter than `i:x`→`j:y` if and only if that in program order, (1) `i:v` is after or equal to `i:x`; (2) `j:w` is before or equal to `j:y`.

However, RTR must not create *overly strict* dependencies, or it will cause replay deadlocks. For example, in Figure 3, let's assume RTR creates `i:3`→`j:1` to replace `i:3`→`j:5`. At a glance, `i:3`→`j:1` may appear correct because it is stricter than `i:3`→`j:5`. However, a cycle of dependence, namely `i:3`→`j:1`→`j:2`→`i:3`, is created after this dependence is added. Note that a program order dependence (`j:1`→`j:2`) is in the cycle. This cycle will cause replay to deadlock because one cannot decide whether `i:3` or `j:1` should be executed first.

To avoid replay deadlocks, RTR conservatively ensures that all stricter dependencies are in a total order. It is natural to follow the SC total order in a live execution. This avoids any cycle of dependencies, because the SC total order is a topological sort of all dependencies (within and between threads) in replay.

Replay Performance. During replay of a recorded execution, each dependence, say `i:x`→`j:y`, requires a check between two threads. The check incurs additional delay to operation `j:y` only if `i:x` is not yet finished when `j:y` is ready. RTR affects replay performance in two ways. First, creating stricter dependencies can add more delays to replay, because stricter dependencies are sufficient but not necessary for replaying the conflicts. On the other hand, RTR may improve replay performance with reduced number of checks between threads. We leave a quantitative evaluation of the impact on replay performance to future work.

3.2.3 *The RTR Algorithm*

We now provide a high-level RTR algorithm in Table 1, assuming infinite cache capacities, in-order processor cores, no counter wrap-around, *etc.* Compared with the TR algorithm in FDR, we believe RTR is only slightly more complex than TR. Nevertheless, two major differences exist.

Table 1: RTR Algorithm on Ideal Hardware.

States at processor *j*

IC: Instruction Count of processor *j*

LOG: Log buffer at each processor *j*

CTS[M]: Cache TimeStamp. Last access timestamps (scalar) of blocks in *j*'s cache. **M** is the total number of memory blocks

VC: Vector Clock. Maximal timestamp (vector) of processor *i* that is ordered before an IC of processor *j*. VC has **P-1** elements, where **P** is the total number of processors.

Δ_min[P]/Δ_max[P]: Sliding window of the IC stride for vectorizable dependencies from another processor to *j*.

Actions at processor *j*

```
On commit of an instruction insn {
    IC++ // After, IC is insn's dynamic instruction count
    if (insn is a memory operation) {
        // insn.block is the block accessed by insn
        CTS[insn.block] := IC
    }
}
On sending a coherence response for block b to proc k {
    // Send both last access IC and processor current IC
    ID := j
    last_access_TS := CTS[b]
    current_processor_IC := IC
    SEND(ID, last_access_TS, current_processor_IC)
}
On receiving a coherence response for block b from proc i {
    // 1. Drop the conflict if it is transitive reducible
    // 2. Create a vectorizable stricter dependence if possible
    // 3. Otherwise, log the previous groups of conflicts
    (ID, last_access_TS, current_processor_IC) := RECEIVE()
    min := IC - current_processor_IC
    max := IC - last_access_TS
    if (last_access_TS <= VC[i]) {
        // Transitively reduced, do nothing
    } else if ( [min, max] overlaps [Δ_min[i], Δ_max[i]] ) {
        Δ_min[i] := MAXIMAL(min, Δ_min[i])
        Δ_max[i] := MINIMAL(max, Δ_max[i])
        VC[i] := IC - Δ_max[i] // for transitive reduction
        APPEND(LOG, ID, IC) // a conflict ending is logged
    } else {
        APPEND(LOG, Δ_max[i]) // a stride is logged
        Δ_min[i] := min // reset the stride min
        Δ_max[i] := max // reset the stride max
        VC[i] := IC - Δ_max[i] // for transitive reduction
    }
}
```

1) When sending a coherence response, a processor sends *both* the last access timestamp of the block and the processor's current IC, instead of just the single last access timestamp as in FDR.

2) A processor maintains a *sliding window* of stricter dependencies that it is allowed to create. The window is represented by two IC strides: from the IC stride of the precise dependence to the IC stride of the strictest dependence allowed by the SC total order. A vectorizable stricter dependence is created if a conflict has a window that *overlaps* with the

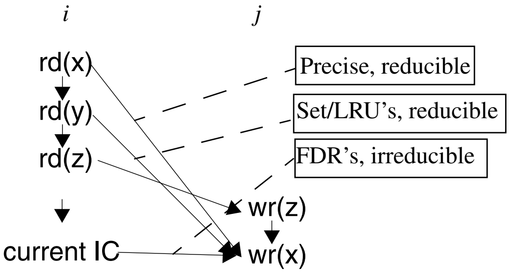


Figure 4: In contrast to FDR, the Set/LRU approximation method likely creates reducible dependencies.

window of preceding conflicts. Otherwise, preceding vectorized dependencies are flushed out to the log. As in FDR, the variable VC[*i*] is used in transitive reduction, but its value is computed based on the stricter dependencies created by RTR.

In more detail, the algorithm in Table 1 keeps a timestamp *CTS[b]* for every cache block *b* at each processor. The timestamps are updated when any memory instruction that accesses *b* commits. Later, if another processor requests *b*, the timestamp and the current processor IC are sent to the requestor in a coherence response message. When the response message arrives at the requestor, the conflict is detected and one of the following three actions are performed. First, transitivity reduction is done based on the previously-logged dependencies (a maximal-received-timestamp, *VC[i]:=IC-*Δ*_max[i],* was computed). If the conflict is irreducible and it has a window of stricter dependencies that overlaps with the current sliding window (denoted by Δ*_max[i]*/Δ*_min[i]* at each processor), a stricter and vectorizable dependence is created and the sliding window is updated. Otherwise, in the final case, the existing group of stricter and vectorized dependencies are flushed out to the log and the sliding windows is reset according to the conflict.

4. Reducing Hardware Cost Using Set/LRU

The RTR algorithm in Table 1 assumes infinite caches. In reality, however, cache size is limited and the cost of storing timestamps for all memory blocks is prohibitive. Therefore, whenever a processor P receives an invalidation request for a cache block b that is no longer cached at P (this can happen when the directory still thinks that P is a sharer of b), P must respond to the request with an ACK message containing the timestamp that approximates the timestamp that b would have if b was still cached (and b's timestamp was maintained precisely).

In FDR [33], we observed that it is safe to approximate an uncached block's timestamp (herein called the *precise timestamp*) with any timestamp within the inclusive range from the precise timestamp to the current IC of the responding processor. This observation is called *Send Observation*. Send Observation is safe for the same reasons that RTR can create stricter (and not overly strict) dependencies. FDR conservatively chooses to always use the current IC to approximate the precise timestamp. This is safe but unfortunately always creates a dependence that is irreducible by transitivity, meaning the dependence will be logged. FDR's approximation method presents an undesired trade-off between the log size and the hardware cost (*i.e.*, keeping few timestamps helps reduce the hardware cost of the recorder, but tends to generate many extra irreducible dependencies, which increase the log size).

We now propose a new timestamp approximation method, called *Set/LRU Timestamp Approximation* (Set/LRU, in short). Instead of using the current IC, like in FDR, Set/LRU approximates the precise timestamp with the timestamp of the Least Recently Used (LRU) block that maps to the same associative cache set as the uncached block. Because the new approximations tend to preserve the transitive reducibility of the dependencies, Set/LRU outperforms the current IC approximation method by reducing both the hardware cost and the log size.

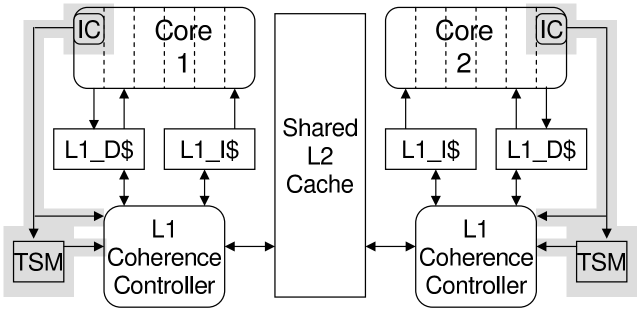


Figure 5: A decoupled timestamp memory (D-TSM) and its data paths (shaded) in a 2-way CMP.

For example in Figure 4, a thread i on processor P1 serially reads three different memory blocks x, y and z, which map to the same cache set. If P1's cache associativity is two, by the time i reads z, x needs to be evicted. After x is evicted, if another thread j on processor P2 serially writes z and x*,* first, a conflict on block z from i to j is detected and the corresponding dependence is logged. Second, the conflict on block x is detected if the directory remembers P1 as a sharer of x after x is evicted. Using the Set/LRU method, P1 chooses the timestamp of y of approximate the precise timestamp of x. The approximated timestamp is guaranteed to be in the range allowed by the Send Observation and to be the closest to the precise timestamp among the timestamps of the cached blocks in the cache set. This is so because of the LRU replacement policy. More importantly, Set/LRU preserves the reducibility of the conflict on x. Unlike the current IC approximation used in FDR, no extra dependence is logged. Therefore, it is possible for Set/LRU to significantly reduce the hardware cost by storing only a small number of timestamps that likely correspond to the small number of irreducible conflicts. As described in Xu's Ph.D. dissertation [32], non-LRU replacement is also supported.

5. Reducing Complexity Using D-TSM

FDR integrates the timestamp memory with the memory caches. The caches record, for each block, the timestamp of the latest read or write access to the block. In this section, we describe a *Decoupled Timestamp Memory* (D-TSM), which is simpler and more flexible than the integrated timestamp memory.

Figure 5 shows the high level structures of the D-TSMs in a 2-way CMP. The D-TSMs have both read and write ports, connecting respectively to the processor pipeline and the coherence controller. When the processor commits a memory instruction, it writes the IC of the instruction to the timestamp memory, which is indexed by memory block address. When coherence requests and responses arrive at the L1 coherence controller, the controller looks up the corresponding timestamp from the timestamp memory, then either sends the timestamp to the requestor or conditionally writes the log. The D-TSM latency needs to be as fast as the memory caches, just like the case of the integrated timestamp memory. This speed can be met relatively easily since the size of the D-TSMs is usually small (Section 10.2.1).

Decoupling has two benefits. First, D-TSM may reduce the design complexity. Memory caches are performance-critical. Integrated timestamp memory increases design complexity, which may affect the cache performance and may generate push-back from the designers [22]. A dedicated timestamp memory is simpler than memory caches, because (1) it does not need to modify the cache and can be designed separately "on the side"; (2) it does not need to keep any MOESI state[1]; (3) it does not need to support invalidations; (4) decoupling allows D-TSMs to implement the LRU replacement policy, while the cache to implement another policy. The second benefit of decoupling is the flexibility. We can vary the size (number of associative sets and associativity) of the D-TSMs independently with respect to the size of the caches. In Section 10.2.1, we show that the D-TSMs can be sized much smaller than the caches, thanks to the Set/LRU method. A drawback of decoupling is the duplicated address tag arrays, which is insignificant due to the small size.

6. Broadening Applicability: Supporting TSO

We now extend our recorder to supporting the Total Store Order (TSO) memory model [31]. We focus on TSO because it is well defined [31] and perhaps widely implemented in the popular x86 architecture as a valid implementation of the Processor Consistency (PC) model [14].

We have three goals in extending RTR to TSO systems.

1) Recording TSO executions without forcing the executions to conform to SC.
2) (At most) Modestly increasing the log of the recorder.
3) (At most) Modestly increasing the hardware complexity and cost of the recorder.

Because our simulation infrastructure supports only the Sequential Consistency (SC), we do not quantitatively evaluate the TSO recording method described in this section. But, we qualitatively argue that the TSO recording method achieves these goals.

6.1. TSO and its Impact on Race Recording

TSO relaxes write-to-read ordering to the shared memory. With TSO, a processor can implement a hardware first-in-first-out (FIFO) *write buffer*. Informally, we say a store instruction *commits* by placing its write value in the write buffer. Later, the store instruction is *ordered* (at the memory) when the instruction exits the write buffer and updates the memory. Furthermore, TSO requires a processor's stores to be *ordered* in *commit* order (*i.e.*, FIFO write buffer). Load instructions from the same processor can *bypass* the memory system from the write buffer.

A load instruction *commits* and is *ordered* (at the memory) after all instructions before the load instruction commit. When a load returns its value from the memory system, we call it a load$_M$ (load-memory), otherwise a load$_B$ (load-bypass). (If a load instruction partially bypasses from the write buffer, we break it into smaller load$_M$ and load$_B$ sub-operations.)

Our race recording algorithm (as presented so far) can fail under TSO systems. In particular, some load$_M$ instructions can potentially cause *instruction reordering*; some Load$_B$ instructions can potentially cause *store atomicity violation*. Arvind and Maessen have shown that TSO and SC differ *only* in the properties of instruction reordering and store atomicity [2]. Therefore, we believe a solution is complete if it handles both instruction reordering and store atomicity violations. We now use examples to illustrate the impact of instruction reordering and store atomicity violations to race recording in more detail.

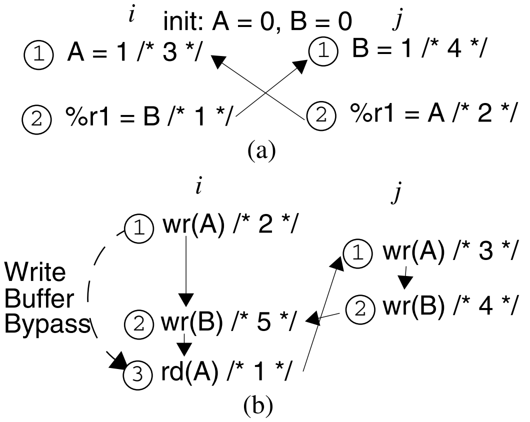


Figure 6: Example TSO executions that are not SC. (a) The store instructions are delayed by the write buffer. (b) the load instructions reads its value directly from the write buffer. In both (a) and (b), a load instruction is ordered (at the memory) before a preceding store. The numbers in /* */ denote the memory ordering.

In an SC system, instructions (executed by the same thread) commit and are ordered in the program order. In a TSO system, even though the instructions commit in program order, the write buffer can cause independent load and store instructions to order differently from the program order. This reordering can cause a multithreaded execution to violate the strong SC semantics. We call those multithreaded executions that are allowed by the TSO memory model the *TSO executions*.

Figure 6a shows an example TSO execution that is not an SC execution. In this execution, thread i first writes memory location A then reads a different memory location B, thread j first writes B then reads A. Let us assume thread i and j run on two different processors. Because of the write buffers, the two load$_M$ instructions are ordered (*i.e.*, commit) before the two store instructions are ordered. The numbers in /* */ denote the memory ordering. For this execution, our race recorder would log two interthread dependencies i:2→j:1 and j:2→i:1. During the replay, if the replayer follows the SC order, the recorded dependencies cause a deadlock (*i.e.*, cycle of dependencies).

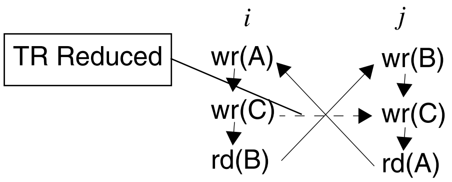
Figure 6b shows another TSO execution that is not an SC execution caused by a load$_B$ instruction. The load$_B$ instruction i:3 reads A's value from i:1 through the write buffer during recording. Although i:3 is ordered (at 1) before i:2 (ordered at 2), our recorder logs the dependence i:3→j:1, rather than i:1→j:1, because i:3 is after i:1 in the program order (with a larger IC). Thus, a cycle of dependencies is recorded due to the write buffer bypassing.

Therefore, recording only interthread dependencies can potentially deadlock a SC-replay for TSO executions.

6.2. An Order-Value-Hybrid Recorder

We propose an order-value-hybrid recorder to handle TSO executions. Our key observation is that some load instructions cause replay deadlocks, because these load instructions are ordered (at the memory) differently with respect to the program order.

[1] All timestamps are initiated to zero.



The recording order:

j:rd(A) → *i*:rd(B) → *i*:wr(A) → *j*:wr(B) → *i*:wr(C) → *j*:wr(C)

The values of *j*:rd(A) and *i*:rd(B) are logged.

The conflict *i*:wr(C) → *j*:wr(C) is reduced by *i*:rd(B) → *j*:wr(B)

The *incorrect* replay order:

j:wr(B) → *j*:wr(C) → *j*:rd(A) → *i*:wr(A) → *i*:wr(C) → *i*:rd(B)-from value log

Figure 7: TR and TSO executions. Applying TR to TSO executions directly can cause incorrect replay. The dependence *i*:wr(C)→*j*:wr(C) is transitively reduced by the WAR dependence (*i*:rd(B)→*j*:wr(B)). However, because the WAR dependence is removed to break the cycle of dependencies, as shown by the incorrect replay order, the dependence *i*:wr(C)→*j*:wr(C) is not satisfied. Our solution is to avoid using those removable WAR dependencies in transitive reduction in the recording.

Therefore, we give special treatment to those "problematic" load instructions, in effect, replaying these load instructions by value, rather than by ordering.

In addition to recording dependencies (the orders), our recorder records the value of a load instruction if the load instruction may violate the SC ordering. The hybrid recording method deals with the load_M and load_B instructions uniformly: the recorder monitors the loaded memory location from the (physical) time `t1` to (physical) time `t2`, where time `t1` is when a load_M instruction becomes ordered or the store that feeds a load_B instruction becomes ordered; time `t2` is when all preceding instructions (in program order) of the load_M or the load_B are ordered. During this monitored time period, if the loaded memory location is written by another thread, the recorder logs the *value* of the load instruction. The load instruction is called a potential SC-violating load. If the memory location is not written by another thread, the recorder does not log the load value, because the load instruction is logically ordered in program order, which means the dependencies will ensure correct replay of the load.

The definition of time `t1` is different for load_M and load_B instructions. It is relatively easier to understand why we start to monitor a load_M instruction after the load_M is ordered. After all, if no other thread writes the loaded memory location in the time interval [`t1`, `t2`], the instruction reordering logically did not happen. On the other hand, we choose to start monitoring a load_B after the store (say `store_x`) feeding the load_B becomes ordered, rather than when the load_B is ordered. This is because the loaded value is from the `store_x`. A potential SC violation happens if the value is written in [`t1`, `t2`].

From a hardware perspective, to achieve this value recording, we augment the processor core with additional hardware that monitors the accessed cache line after the value of a load instruction is returned by the cache or a bypassed store value is written into the cache line. Should the cache line be written (invalidated) by another processor before all preceding (in the program order) store instructions (which are delayed by the write buffer) are ordered, we treat the load as a potential SC-violating load. For these loads, our recorder logs the loaded value and omits logging the WAR dependence that sources from the load instruction, which may cause replay deadlocks. The detection circuitry is similar to the misspeculation detection circuitry in the SC systems (*e.g.*, MIPS R10000 [34]) that utilize speculative execution techniques [9]. The difference is that our hardware logs the load values of potential SC violations rather than triggering recovery.

We now apply this hybrid recording method to the example in Figure 6a. The hybrid recorder would monitor the memory locations `A` and `B`, at thread `j` and thread `i` respectively, after the two load_M instructions are ordered. Because `A` is written by thread `i` before the store instruction `j:1` is ordered, the recorder would log the load value of `j:2` (`A=0`). Similarly, for the example in Figure 6b, the hybrid method starts monitoring the memory location `A` after `i:1` is ordered. Because `A` is written before `i:1` (which precedes `i:3`) is ordered, the recorder would log the load value of `i:3`.

During replay, the logged values are used to overrule the (potentially incorrect) values read from the memory. In Figure 6a, without the WAR dependence `j:2`→`i:1`, the replay will not deadlock. When it comes time to execute `j:2`, however, the replayer uses the logged value to overrule the value loaded from the memory. Therefore, in addition to the changes in the recorder, the order-value-hybrid method requires a (small) change in the replayer so that the replayer can overrule the value of a potential SC-violating load instruction using the value log.

The hybrid recording algorithm has the following properties.

This order-value-hybrid recording method does not force TSO executions to conform to the SC model. Instead, the new method logs additional values to deal with the potential deadlocks during the in-order replay.

This order-value-hybrid recording method should (at most) slightly increase the log size of the recorder. Several studies have shown that, even under the consistency models that are weaker than TSO, cache lines of load instructions are rarely invalidated before the all preceding instructions are ordered [6, 10]. Therefore, the load values are infrequently logged.

Without applying TR and RTR, the hybrid recorder is correct because it logs all conflicts except those WAR conflicts source from potential SC-violating loads. The recorder omits logging a subset of the WAR dependencies. The omission does not affect the replay correctness of the thread that wrote the conflicting memory location (the write is still executed in the right order with respect to other writes). It does affect the reading thread, because removing the WAR dependence causes the load instruction to the see a new version of the memory location. However, our value log supplies the correct value to the load instruction. Therefore, all load instructions get the same values in the original and the replay executions. As a result, the hybrid recording method provides a successful replay for both SC and TSO executions. The hybrid recording algorithm is given in Table 3-4 in Xu's dissertation [32].

Next, we extend the hybrid recording method to the TR and RTR. The basic principle is that we suppress the TR and RTR optimizations for potential SC-violating loads.

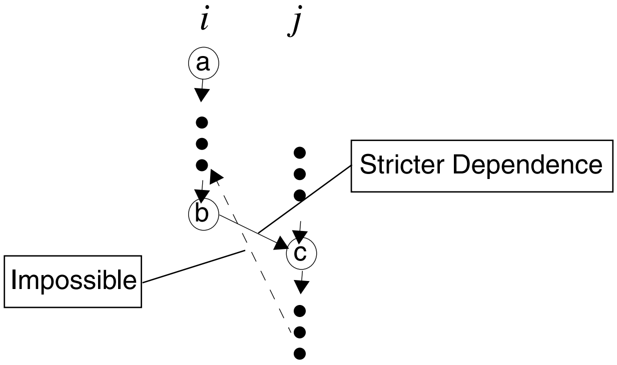


Figure 8: A narrower sliding window for RTR.

6.3. TR and the Hybrid Recording

As shown in Figure 7, It is incorrect to apply TR to the WAR dependencies that are later omitted. The dependence `i:wr(C)`→`j:wr(C)` is transitively reduced by another WAR dependence (`i:rd(B)`→`j:wr(B)`). However, if the WAR dependence is omitted in the recording to avoid replay deadlocks, an incorrect replay order can violate the dependence `i:wr(C)`→`j:wr(C)`.

We can solve this problem by avoiding using those omitted WAR dependencies in TR. In particular, our recorder performs TR by comparing the timestamps received from a remote processor. When a processor receives an invalidation for a memory block that is currently being monitored for a potential SC-violating load, the processor piggybacks the older timestamp of the block, rather than the timestamp of the potential SC-violating load. This way, not only does the requestor avoid logging the WAR dependence, but also the correctness of TR is maintained.

6.4. RTR and the Hybrid Recording

Recall that RTR relies on the SC total order to avoid logging overly-strict dependencies. In TSO executions, the SC total order does not exist. To extend RTR to TSO execution, we modify RTR so that it creates the stricter dependencies more conservatively to avoid replay deadlocks. In an SC system, RTR is allowed to create stricter dependencies sourced from the precise timestamp to the most recently committed IC. In a TSO system, because of the write buffer, committed instructions may still be unordered at the memory. Therefore, RTR is allowed to create stricter dependencies only from a narrower sliding window, which is the largest window of ordered and consecutive instructions.

Figure 8 depicts this new sliding window, say [`i:a`, `i:b`], where `i:a` is the conflicting instruction to `j:c` and `i:b` is the latest instruction that satisfies the condition that all instructions between `i:a` and `i:b` (in program order) are ordered and next instruction after `i:b` (in program order) is unordered. There must not be any cycle of dependencies, because such dependence like the one shown in the dashed arrow, requires either (i) an instruction between `i:a` and `i:b` ordered after `j:c`, or (ii) an instruction after `j:c` ordered before `j:c`. By definition of the new sliding window, case (i) is impossible. Case (ii) is possible for potential SC-violating loads, but the cycle is avoided by omitted the WAR.

6.5. Insufficiency for More Relaxed Models

The hybrid recording method cannot be directly applied to Processor Consistency (PC) and other more relaxed consistency models for reasons documented in Xu's dissertation [32]. Supporting these consistency models is an open problem.

7. Other Timestamp Memory Optimizations

This section briefly describes two other optimizations of the decoupled timestamp memory. They each allow us to tune a parameters of the D-TSMs in Section 10.1. More details of these optimizations can be found in Xu's dissertation [32].

7.1. Two (Read and Write) vs. One Timestamp

The RTR algorithm in Table 1 stores a single last access timestamp in the *CTS[b]* variable for each block *b*. In fact, this is a simplification. More precisely, both the *last-read timestamp*s and the *last-write timestamp*s can be stored in the timestamp memory to improve the preciseness of the conflict detection. For example, only the last write timestamp is used, when a RAW conflict is detected, even though the last read timestamp may be more recent for the block.

However, storing two timestamps per block doubles the hardware cost of the timestamp memory. In this paper, we show that the best choice of whether one or two timestamps should be stored depends on the size of the timestamp memory. We discuss the trade-off in more detail in Section 10.1.

7.2. Partial Timestamps

Like in FDR [33], we use partial timestamps to reduce the hardware cost. Our recorder stores only the least significant bits (LSBs) of the timestamp. For example, instead of storing timestamp 0x1234 in the timestamp memory, we store only 0x34. When the partial timestamp is read back, it is concatenated with the most significant bits from the current processor IC. Say, the current processor IC is 0x4321, the concatenation produces a timestamp 0x4334. Since 0x4334 is larger than the current processor IC, we use 0x4$\underline{2}$34 to approximate the original timestamp (0x1234), because 0x4234 is the largest possible IC that can produce the partial timestamp 0x34. If the concatenation is less than the current IC, we simply use it to approximate the original timestamp. This approximation again creates stricter (and not overly strict) dependencies. We explore the design space of the width of the partial timestamps in Section 10.1.

8. Example Recorder: RTR/CMP

We now describe a specific design of a hardware race recorder called *RTR/CMP*. RTR/CMP differs from FDR (and other existing race recorders) in following ways: (1) use of the RTR algorithm; (2) use of the Set/LRU approximation; (3) use of the D-TSMs; (4) support of Chip MultiProcessor (CMP). Due the simulator limitation, RTR/CMP does not implement the order-value-hybrid recording method.

8.1. The Baseline CMP System

Our baseline system (without the recorder) is a single-chip CMP system. Single-chip CMP systems, such as Sun UltraSPARC T1 [11], do not support memory coherence between multiple CMP chips. The complexity of the cache coherence protocols for single-chip CMP systems is more manageable than that of the multichip CMP systems [18]. For RTR/CMP, a single-chip CMP baseline allows simple design and easy deployment because no off-chip change is required.

If race recording is desired on large scale servers consisting of multiple CMPs, we believe coherence piggybacking is possible but

Table 3: Commercial Workloads

Apache is a static web serving workload. We use Apache 2.0.43, configured to use pthread locks and minimal logging as the web server. We use SURGE [4] to generate web requests. We use a repository of 20,000 files (totalling ~500 MB). We simulate 3200 clients, each with 25 ms think time between requests, and warm up for ~2 million requests before taking measurements for 600 requests.
Online Transaction Processing (OLTP) models database activities of a wholesale supplier, with many concurrent users performing transactions. Our setup uses TPC-C v3.0 benchmark and IBM's DB2 v7.2 EEE database management system. We use a 5 GB database with 25,000 warehouses stored on eight raw disks and an additional dedicated database log disk. We reduced the number of districts per warehouse, items per warehouse, and customers per district to allow more concurrency provided by a larger number of warehouses. We simulate 128 users, and warm up the database for 100,000 transactions before taking measurements for 200 transactions.
SPECjbb is a server-side java benchmark that models a 3-tier system, focusing on the middle-ware server business logic. We use SUN's HotSpot 1.4.0 Server JVM. Our experiments use 1.5 threads and 1.5 warehouses per processor (6 for 4 processors), a data size of ~44 MB, a warm-up interval of 200,000 transactions and a measurement interval of 10,000 transactions.
Zeus is another static web serving workload driven by SURGE. Zeus uses an event-driving server model. Each processor of the system is bound by a Zeus process, which is waiting for web serving event (*e.g.*, open socket, read file, send file, *etc.*). The rest of the configuration is the same as Apache's.

more complex. Both the on-chip and off-chip coherence protocols must work together to detect all conflicts with precise timestamps or safe timestamp approximations.

At a high level, the baseline cache coherence protocol works as follows. The shared on-chip L2 cache keeps track of sharer and owner information of cache blocks in private L1 caches (similar to Piranha [5]). The MOSI coherence protocol implements notifying replacement, *i.e.*, the L2 cache directory keeps a precise list of L1 sharers for each block.

8.2. Recorder Implementation

Like FDR, RTR/CMP piggybacks the race recording function onto the cache coherence hardware and implements the RTR algorithm (Section 3), rather than Netzer's TR algorithm. In addition, RTR/CMP employs the Set/LRU timestamp approximation (Section 4), and the D-TSMs (Section 5). The major implementation differences from FDR is the following.

1) **L2 Cache Replacement.** Unlike the coherence directory used in FDR, our directory (at L2) removes sharer and owner information when a cache block is evicted from all caches of a chip. This is a problem for conflict detection, because when the block is brought back on to the chip, the L2 does not have the information about which processor cores were the owner or the sharers of the block. We solve this problem by conservatively assuming that all processor cores (except the requestor) had cached this block before. We then make the requestor the new owner of the block and subsequent requests to the block will conflict with only the requestor. This causes false conflicts to be detected, but never misses a true conflict. These false conflicts are likely reducible, because the Set/LRU approximation returns old timestamps for these conflicts.
2) **Notifying L1 Replacement.** In FDR, when a cache replaces a read-only block, it does not notify the memory. If the block is later written by another processor, the memory sends an invalidation to all sharers regardless of whether the block is still cached by the original sharers. In RTR/CMP, however, a sharer notifies the L2 when a read-only block is replaced. If the block is later written by another core, the L2 does not need to send invalidations to sharers-who-replaced. This is a problem for conflict detection, because although the L2 can keep extra information about the sharers-who-replaced, it cannot provide the requestor with the necessary timestamps of the block (unless it sends extra messages to the sharers-who-replaced). We solve this problem by piggybacking block access timestamps in the replacement notification messages and caching the timestamps of the replaced blocks at the L2. This solution adds four extra D-TSMs to the L2 (one per core), but avoids extra coherence messages, which affect performance.
3) **Coherence Message Overhead.** RTR/CMP uses 64-bit timestamps to avoid overflows. As shown in Table 1, RTR requires piggybacking two timestamps per coherence message. This incurs high bandwidth overhead. We reduce the overhead by encoding the timestamps with their first order differences (24-bit). Because messages may be re-ordered in the interconnection network, we also add a sequencing number (8-bit) to each message to ensure correct re-construction of the full timestamps from the differences.

Table 2: Simulation Parameters

Cores	Four 1 GHz, 2-way, in-order superscalar
Private L1 Caches	Split I & D, each 64 KB 4-way set associative with LRU replacement, 64-byte lines, 1-cycle
Shared L2 Cache	Unified 16 MB, 4-way set associative with LRU replacement, 64-byte lines, 15-cycle
Memory	4 GB of DRAM, 80ns off-chip access time
Timestamp Memory	Decoupled, parameters vary by design, but we keep L1 and L2 TSMs have the same parameters

9. Evaluation Methods

We use Wisconsin GEMS full system simulation infrastructure [17] to evaluate RTR/CMP. GEMS (through Simics [16]) models an enterprise-level SPARC multiprocessor system in sufficient detail to run the unmodified Solaris 9 operating system. Table 2 summarizes the system configuration we simulate.

We exercise RTR/CMP with four commercial workloads summarized in Table 3. We counter the workload variabilities using a pseudo-random perturbation method [1]. For performance simulations, we report the mean results and 95% confidence intervals from 20 randomized runs of approximately 100 million instructions per core for every workload. We report the average log growth rate of RTR/CMP in MegaBytes/core/second and Bytes/kilo-instructions.

We do not directly compare the log size of RTR/CMP to the log size reported in the FDR paper, because RTR/CMP and FDR are based on different systems (*i.e.*, CMP versus CC-NUMA). Instead, we approximate FDR by disabling RTR and Set/LRU, as well as sizing up the timestamp memory appropriately (details in Section 10.2). For both recorders, the log is first compressed with a first-order-difference encoding and then LZ77 [37]. The encoding is similar to a run-length encoding proposed by Ronsse *et al.* [29].

Table 4: RTR/CMP vs. FDR - Hardware Cost

Recorder Parameters	FDR	RTR/CMP
Algorithm (RTR or TR)	TR	RTR
Set/LRU	No	Yes
# of Timestamps per Block	1	2
Partial Timestamp Bits	32-bit	24-bit
Timestamp Memory Associativity	4-way	64-way
Timestamp Memory Size/core	256 KB	12 KB+12 KB

10. RTR/CMP Results

In this section, we first explore a design space of RTR/CMP by varying the configuration of the decoupled timestamp memories (D-TSMs). Then, we select a specific configuration of RTR/CMP and evaluate the improvements over another configuration that approximates FDR. We believe both results are significant: the design space exploration shows RTR/CMP is tunable to meet different design goals; the specific design shows the effectiveness of RTR and Set/LRU as well as the improvements over FDR.

10.1. Optimizing RTR/CMP

We vary four parameters of the D-TSMs:

1) **Size** of the D-TSMs (same size for L1 and L2 D-TSMs);
2) **Bit-width** of partial timestamps;
3) **Number** of timestamps per block (one or two);
4) **Associativity** of the D-TSMs.

We perform several sensitivity studies to determine the effects of these parameters. We report the results from a typical workload. In Figure 9a, we vary the size and number of timestamps per block for the D-TSMs. For comparison, we experimented with both the RTR and TR algorithms. Both the RTR and TR algorithms exhibit similar trends: (1) small D-TSMs (no more than 64 KB) achieve most of the log size reduction; (2) separate (two) timestamps per block should be used after the cache size is large enough that Set/LRU becomes effective. In Figure 9b, we vary the number of bits of partial timestamps. Although the full width of a timestamp is 64-bit, it is economical to store only the least significant 24 bits. In Figure 9c, we vary the associativity of the D-TSMs. Unlike the Set/LRU, the current IC approximation does not benefit from higher associativity[2]. In Section 10.2, we choose 64-way because higher associativity has a diminishing effect on the log size reduction.

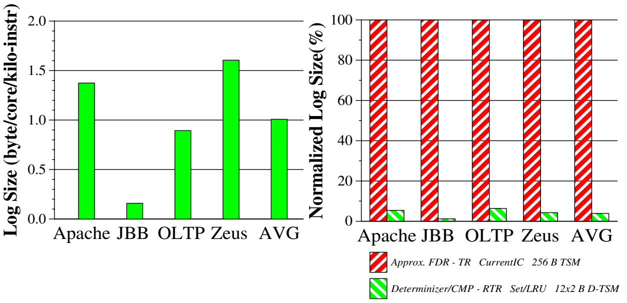


Figure 10: RTR/CMP—Log Size.

These results show a tunable trade-off between the log size and the hardware cost in designing race recorders.

10.2. Improvements of RTR/CMP

We select a specific configuration of RTR/CMP and compare it to the approximated FDR. Table 4 summarizes the parameters of the selected RTR/CMP and the approximated FDR, which does not use RTR, Set/LRU and the D-TSMs.

10.2.1 Hardware Cost

Table 4 shows the relative hardware cost of RTR/CMP and FDR. RTR/CMP requires two small timestamp memories per core. Each memory has 32 sets and 64 ways. With two timestamps per block and 24-bit partial timestamps, the total size of the timestamp memory is 24 KB–a significant hardware cost reduction over FDR. Further reducing the hardware cost is possible at the expense of larger log sizes. As we show in Figure 9a, reducing the timestamp memory to 2 KB increases the log growth rate to about 0.5 MB/core/s.

10.2.2 Log Size

Figure 10-left shows the log grows about one byte per kilo-instruction for RTR/CMP. Comparing with FDR, RTR/CMP significantly reduces the log growth rate for all workloads (Figure 10-right). On average, the log size reduction is 96%—a factor of 25! The log size reduction is a result of the RTR algorithm and the Set/LRU approximation. Next, we evaluate them independently.

TR vs. RTR. To isolate the effects of the RTR algorithm from the Set/LRU approximation, we give the recorder infinite timestamp memories, *i.e.*, no timestamp memory misses, two timestamps per block, and full-width timestamps. Figure 11-left shows the log size reduction from changing the recording algorithm from TR to RTR. For all workloads, RTR has a lower log growth rate. The average

[2] Because of encoding inefficiency in the log, RTR performs worse than TR when Set/LRU is not used. With 64 KB D-TSMs and without Set/LRU, RTR cannot vectorize most of the dependencies.

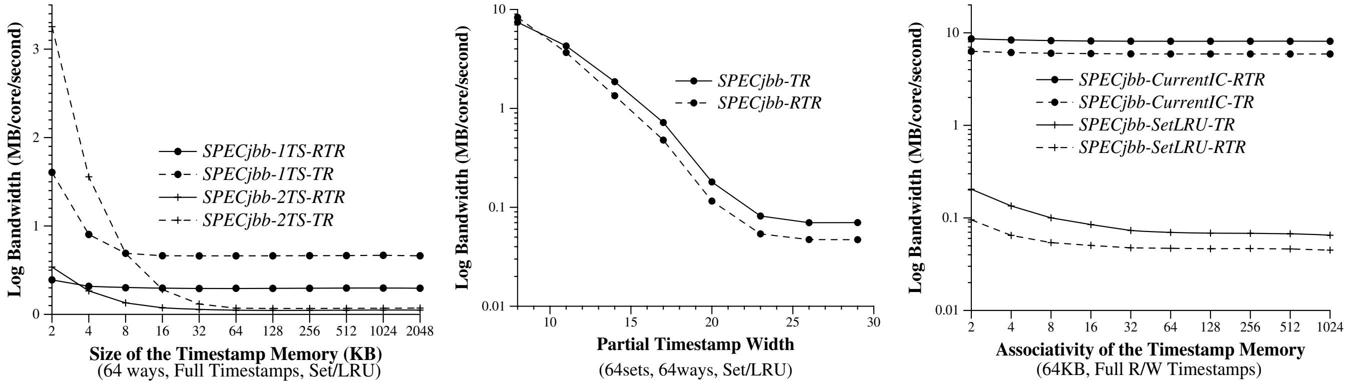


Figure 9: Exploring the design space of the timestamp memory of RTR/CMP.

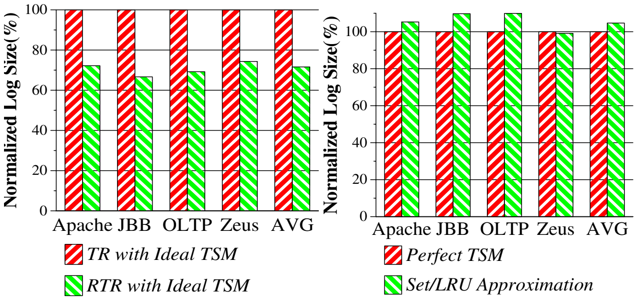


Figure 11: Effectiveness of RTR and Set/LRU.

reduction is 28%, which is a result of RTR's ability to create many groups of dependencies with the same IC stride.

Set/LRU. Figure 11-right shows the effectiveness of the Set/LRU approximation. With two 12 KB D-TSMs, the log growth rate is increased by no more than 10% over the perfect (infinite) D-TSMs. Set/LRU slightly outperforms perfect D-TSMs for Zeus. This is possible because RTR is a greedy heuristic and precise timestamps do not always help in transitive reduction. In other words, an approximated timestamp may enable better transitive reduction.

Additional impacts of Set/LRU can be seen in Figure 9c. In that figure, we use 64 KB timestamp memories and vary the timestamp approximation method and the timestamp memory associativity. The figure reveals three insights: (1) Set/LRU dramatically reduces the log size, because the dependencies are much more likely reducible by both TR or RTR; (2) The associativity of the D-TSMs helps Set/LRU, because higher associativity enables better approximations; (3) The RTR algorithm works better if it is combined with Set/LRU, because Set/LRU enables more flexibility for RTR to create stricter dependencies.

10.2.3 Runtime and Bandwidth Overheads

Like FDR, due to the hardware assistance, RTR/CMP has negligible runtime overhead and modest interconnect overhead. Figure 12 shows the runtime performance overhead is less than 2%; the interconnect overhead is about 10%.

11. Related Work

One of the primary applications of memory race recording is debugging multithreaded software. In the past, software debuggers/recorders [23, 7, 27, 28] for parallel programs have been researched extensively. More recently, hardware-assisted debuggers/recorders [3, 19, 25, 33, 36, 35, 26, 20, 24] have gathered more attention. Among them, FDR [33] and BugNet [20] (using FDR's implementation for race recording) aim to provide efficient and low-cost solutions to multithreaded execution recording. Thus, they can benefit directly from our better and cheaper memory race recording. ReEnact [25] and CORD [24] aim to provide hardware-assisted online data race detection. We believe they can benefit indirectly from our memory race recording and the proposal of decoupled timestamp memory.

ReVirt [8] and ExtraVirt [15] are two emerging applications, in which our new memory race recorder can facilitate intrusion analysis and fault tolerance. We note that both ReVirt and ExtraVirt are built on virtual machine techniques. We believe virtual machine techniques provide a good platform, on which race recording and deterministic replay can be implemented and used.

The RTR algorithm in this paper is related to both *partial order recording* and *total order recording* algorithms. Instant Replay [13] proposed *partial order recording* of parallel executions by logging the orders of parallel events, not the data associated with such events. Netzer [21] proposed and proved the correctness of transitive reduction on partial order recordings and dramatically reduced the log size. Both papers focused on recording only the conflicts. We observe that one does not have to record only the conflicts. As a result, our RTR algorithm further reduces the log size. RecPlay [28] is a race detector based on recording and replaying thread synchronization. RecPlay uses *total order recording*, which uses Lamport Scalar Clocks [12] to record a total order of events. The RTR algorithm can provide a unified view of the partial order recording and the total order recording approaches. In fact, RecPlay's algorithm is a special case of RTR, who creates stricter and vectorized dependencies to an extreme degree so that a partial order relation is reduced to a total order. It is interesting to note that partial order replay does not necessarily degrade replay performance, while the total order replay of every memory accesses is likely slow, because it allows only sequential (or lockstep-parallel) replay.

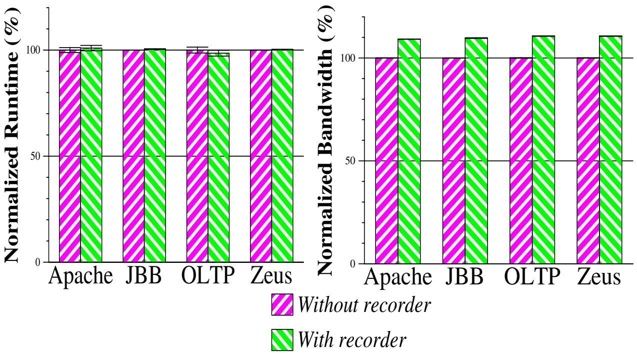


Figure 12: Runtime and interconnect overhead of RTR/CMP.

12. Conclusions and Future Work

To combat nondeterminism caused by memory races in multithreaded programs, we have proposed a new recording algorithm that significantly improves memory race recording in four aspects: the log size, the hardware cost, the complexity and the applicability. The Regulated Transitive Reduction (RTR) regulates how memory races are replayed to reduce the log size. RTR's key novelty is in creating stricter and vectorizable dependencies in the dependence log. The Set/LRU timestamp approximation method computes more accurate timestamps for uncached blocks and enables a significant reduction in both the log size and the hardware cost. Moreover, we propose decoupled timestamp memory (D-TSM) to reduce the hardware complexity of the recorder. Finally, we extend race recording to handle TSO (x86-like), as well as SC to broaden its applicability. We are optimistic that our improved race recorder can benefit many applications of deterministic replay and may encourage adoptions of hardware race recorders.

Future work includes (1) supporting Simultaneous Multithreading (SMT), snooping coherence protocols, and more relaxed memory consistency models, (2) evaluating RTR's impact on replay performance, (3) exploring new recording methods with more compact representations of dependencies or with the assumption of out-of-order replay and (4) applying race recording to more applications in real-life scenarios.

13. Acknowledgements

We would like to thank Remzi Arpaci-Dusseau, Mikko Lipasti, Barton Miller, and David Wood for comments that helped improve this paper and Xu's dissertation. We would like to thank David Bacon, Joel Emer, Babak Falsafi, Phillip Gibbons, Kevin Lepak, Michael Kozuch, Todd Mowry, Chris Newburn, Lacky Shah and Radu Teodorescu for comments and discussions. We are grateful to Mike Marty, Kevin Moore, Dan Gibson and Luke Yen for proofreading the paper and other fellow architecture students at UW for continuous support. We thank all anonymous reviewers of ASPLOS'06 for constructive and highly professional reviews.

This work is supported in part by the National Science Foundation (NSF), with grants CCF-0085949, CCR-0093275, CCR-0105721, EIA/CNS-0103670, CCR-0105721, EIA/CNS-0205286, CNS-0225610, CCR-0243657, CCR-0324878, CCR-0326577 and an award from University of California MICRO program, the Okawa Research Award, as well as donations from IBM, Intel, Microsoft, and Sun Microsystems. This work has also been supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract No. NBCHC020056. Hill has a significant financial interest in Sun Microsystems. The views expressed herein are not necessarily those of DARPA, IBM, Intel, Microsoft, NSF, or Sun Microsystems.

References

[1] A. R. Alameldeen, *et al*. Simulating a $2M Commercial Server on a $2K PC. *IEEE Computer*, 36(2):50–57, Feb. 2003.

[2] Arvind and J.-W. Maessen. Memory Model = Instruction Reordering + Store Atomicity. In *Proceedings of the 33nd Annual International Symposium on Computer Architecture*, June 2006.

[3] 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.

[4] P. Barford and M. Crovella. Generating Representative Web Workloads for Network and Server Performance Evaluation. In *Proceedings of the 1998 Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 151–160, June 1998.

[5] L. A. Barroso, *et al*. Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, June 2000.

[6] H. W. Cain and M. H. Lipasti. Memory Ordering: A Value-Based Approach. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, June 2004.

[7] J.-D. Choi and H. Srinivasan. Deterministic Replay of Java Multithread Applications. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT-98)*, Aug. 1998.

[8] G. W. Dunlap, *et al*. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation*, pages 211–224, Dec. 2002.

[9] K. Gharachorloo, *et al*. Two Techniques to Enhance the Performance of Memory Consistency Models. In *Proceedings of the International Conference on Parallel Processing*, volume I, p355–364, Aug. 1991.

[10] C. Gniady, *et al*. Is SC + ILP = RC? In *Proceedings of the 26th International Symposium on Computer Architecture*, May 1999.

[11] P. Kongetira, *et al*. Niagara: A 32-Way Multithreaded Sparc Processor. *IEEE Micro*, 25(2):21–29, Mar 2005.

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

[13] 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.

[14] K. Lepak. Personal Communication, Mar. 2006.

[15] D. Lucchetti, *et al*. ExtraVirt: Detecting and recovering from transient processor faults. In *2005 Symposium on Operating System Principles work-in-progress session*, Oct. 2005.

[16] P. S. Magnusson et al. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, Feb. 2002.

[17] M. Martin, *et al*. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *Computer Architecture News*, pages 92–99, Sept. 2005.

[18] M. R. Marty, *et al*. Improving Multiple-CMP Systems Using Token Coherence. In *Proceedings of the Eleventh IEEE Symposium on High-Performance Computer Architecture*, Feb. 2005.

[19] S. L. Min and J.-D. Choi. An Efficient Cache-based Access Anomaly Detection Scheme. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 235–244, Apr. 1991.

[20] S. Narayanasamy, *et al*. BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging. In *Proceedings of the 32nd International Symposium on Computer Architecture*, June 2005.

[21] R. H. B. Netzer. Optimal Tracing and Replay for Debugging Shared-Memory Parallel Programs. In *Proceedings of the Workshop on Parallel and Distributed Debugging (PADD)*, p1–11, 1993.

[22] C. Newburn. Personal Communication, Oct. 2003.

[23] C. M. Pancake and R. H. B. Netzer. A bibliography of parallel debuggers, 1993 edition. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging (PADD)*, p169–186, 1993.

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

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

[26] F. Qin, S. Lu, and Y. Zhou. SafeMem: Exploiting ECC-Memory for Detecting Memory Leaks and Memory Corruption During Production Runs. In *Proceedings of the Eleventh IEEE Symposium on High-Performance Computer Architecture*, Feb. 2005.

[27] B. Richards and J. R. Larus. Protocol-based Data-race Detection. In *SIGMETRICS symposium on Parallel and Distributed Tools*, 1998.

[28] M. Ronsse and K. De Bosschere. Non-intrusive On-the-fly Data Race Detection using Execution Replay. In *AADEBUG*, Nov. 2000.

[29] M. Ronsse, *et al*. Efficient coding of execution-traces of parallel programs. In *Proceedings of the ProRISC & IEEE-Benelux workshop on Circuits, Systems and Signal Processing*, p251 – 258, Mar. 1995.

[30] M. Rosenblum. Virtual is Better Than Real. http://www.vmware.com/vmworld/2005/keynote_rosenblum.pdf.

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

[32] M. Xu. *Race Recording for Multithreaded Deterministic Replay Using Multiprocessor Hardware*. PhD thesis, http://www.cs.wisc.edu/multifacet/theses/min_xu_phd.pdf, University of Wisconsin-Madison, 2006.

[33] M. Xu, *et al*. A "Flight Data Recorder" for Enabling Full-system Multiprocessor Deterministic Replay. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, 2003.

[34] K. C. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, 16(2):28–40, Apr. 1996.

[35] P. Zhou, *et al*. AccMon: Automatically Detecting Memory-related Bugs via Program Counter-based Invariants. In *Proceedings of the 37th Annual International Symposium on Microarchitecture*, 2004.

[36] P. Zhou, *et al*. iWatcher: Efficient Architectural Support for Software Debugging. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, page 224, June 2004.

[37] J. Ziv and A. Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, 23(3):337–343, May 1977.