

CoreRacer: A Practical Memory Race Recorder for Multicore x86 TSO Processors

Gilles Pokam
Intel Corporation
gilles.a.pokam@intel.com

Cristiano Pereira
Intel Corporation
cristiano.l.pereira@intel.com

Shiliang Hu
Intel Corporation
shiliang.hu@intel.com

Ali-Reza Adl-Tabatabai
Intel Corporation
ali-reza.adl-
tabatabai@intel.com

Justin Gottschlich
Intel Corporation
justin.e.gottschlich@intel.com

Jungwoo Ha^{*}
Google
jwha@google.com

Youfeng Wu
Intel Corporation
youfeng.wu@intel.com

ABSTRACT

Shared memory multiprocessors are difficult to program because of the non-deterministic ways in which the memory operations from different threads interleave. To address this issue, many hardware-based memory race recorders have been proposed that efficiently log an ordering of the shared memory interleavings between threads for deterministic replay. These approaches are challenging to integrate into current processors because they change the cache subsystem or the coherence protocol, and they mostly support a sequentially consistent memory model.

In this paper, we describe *CoreRacer*, a chunk-based memory race recorder architecture for multicore x86 TSO processors. *CoreRacer* does not modify the cache subsystem and yet it still integrates into the x86 TSO memory model. We show that by leveraging a specific x86 feature, the invariant timestamp, *CoreRacer* maintains ordering among chunks without piggybacking on cache coherence messages. We provide a detailed implementation and evaluation of *CoreRacer* on a cycle-accurate x86 simulator. We show that its integration cost into x86 is minimal and its overhead has negligible effect on performance.

Categories and Subject Descriptors

C.1.0 [Processor Architectures]: General

General Terms

Design Performance Reliability

^{*}Work done while at Intel

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

MICRO'11, December 3-7, 2011, Porto Alegre, Brazil
Copyright 2011 ACM 978-1-4503-1053-6/11/12 ...\$10.00.

Keywords

Memory Race Recorder, Deterministic Replay, Determinism

1. INTRODUCTION

As the number of cores in multiprocessor systems increase, deterministic replay of shared memory parallel programs [22] becomes critical to software development. Deterministic replay enables several new usage models for parallel programs, including debugging [19, 26], testing [5], high availability [4, 27], and forensics analysis [8, 12]. A common technique for deterministic replay records a total order of the shared memory accesses (and other non-deterministic events) during execution and then deterministically replays them.

Recording a total order of the shared memory accesses in software [19, 9, 22] requires instrumentation of every memory access, typically slowing down execution by 100x or more and making software approaches impractical for long running applications or production use. Recent proposals reduce this overhead by disregarding the actual order of shared memory accesses and instead reproducing a buggy execution with high probability [2, 18], but such approaches are not useful for scenarios that require precise deterministic replay.

Hardware solutions [28, 17, 10, 14, 21, 29, 13, 22] can eliminate performance overheads without over-constraining the usage model, but these solutions usually come with high hardware costs, especially for processors that don't support a sequentially consistent (SC) memory model. Besides, most existing hardware proposals [28, 17, 10, 14, 21] assume an SC memory model even though commercial multiprocessors use non-SC memory models; for example, x86 processors implement Total Store Order (TSO). TSO complicates the recording of shared memory access order because a load can be ordered to memory before a prior store. A few prior proposals [29, 13] have addressed this problem for TSO, but their solution comes at the cost of expensive changes to the cache subsystem or the coherence protocol, components that are difficult to validate in modern multiprocessor systems.

This paper proposes *CoreRacer*, a novel hardware approach to recording shared memory access orderings. *CoreRacer* handles TSO without changing the cache subsystem or coherence protocol on modern x86 processors. It uses

chunks [10, 14, 21] to represent the interleavings of shared memory accesses. A chunk represents a block of memory operations that execute atomically (i.e., without an intervening conflicting coherence request).

CoreRacer handles the TSO memory model by logging the number of outstanding stores awaiting commit in the store buffer when a chunk terminates. During offline replay, CoreRacer simulates the store buffer using the logged information, enabling it to reproduce an execution that conforms to the recorded TSO execution.

This paper makes the following contributions:

- It introduces a new chunk-based memory race recorder that uses an existing x86 feature — the *invariant timestamp counter* (TSC) [11] — in a novel way to capture the ordering among chunks from different cores without modifying the cache coherence protocol (Section 3).
- It describes a novel mechanism — called the *Reordered Store Window* (RSW) — for recording TSO execution and a new offline replay algorithm that conforms to the recorded TSO execution (Section 4).
- It describes how CoreRacer is integrated into a modern x86 processor, requiring no changes to the cache subsystem (Section 5).
- It presents an implementation and evaluation of CoreRacer on a full system simulator modeling a modern out-of-order x86-based 8-core multiprocessor system implementing TSO. Our results show that CoreRacer requires minimal hardware changes and has a negligible impact on program execution (Section 6).

2. BACKGROUND

A *memory race recorder* (MRR) logs the order in which the shared memory accesses from different threads interleave. Prior work on hardware support for MRR use either *point-to-point* or *chunk-based* techniques to track the ordering between two instructions that form a data race.

Point-to-point techniques [28, 17] track memory races at the granularity of individual shared memory operations. These techniques augment each cache line with a timestamp that the hardware updates on each access to the line. The MRR hardware piggybacks the timestamps on coherence messages and uses them to order the dependencies between memory instructions on different processors.

Chunk-based techniques [10, 14, 21] track memory access interleavings by observing the number of memory operations that execute atomically (i.e., without interleaving with a conflicting remote memory access). Chunk-based systems typically use signatures [23, 6] to check a snoop request against the set of locally accessed memory locations. Each core has a chunk size counter that tracks the size of the current chunk and a read (write) signature that tracks the cache lines read (written) by the core in the current chunk. To track the global ordering of chunks, each core also maintains a clock that is piggybacked with each coherence message [10] or exchanged at each chunk boundary [21]. On receiving a remote coherence request, a core checks its signatures for a conflict, and on detecting a conflict, it ends its current chunk by clearing its signatures, creating a log entry, and updating its clock. Log entries contain the size

of the ended chunk and a timestamp based on the current clock value.

3. HANDLING SC

This section presents the design of the CoreRacer hardware module and shows how it records chunks on an SC system. Figure 1 shows the block diagram of the CoreRacer module. Load and store addresses are sent to CoreRacer at retirement and memory commit time (i.e., when the store data becomes globally visible), respectively. When CoreRacer receives a load or store address, the MRR logic inserts the address into the appropriate read or write signature buffer (the signature buffer component in Figure 1) and increments the chunk size counter. The memory address insertion operation [21] hash encodes the address into a signature over k bit vectors, where k is the number of hash functions used, and then merges the value into a signature register. Our evaluation (Section 6) models signature register sizes of 1024 bits and 512 bits for the read and write signatures, respectively. For signatures of these sizes, the address insertion operations require simple logic, and our model clocks them in one cycle.

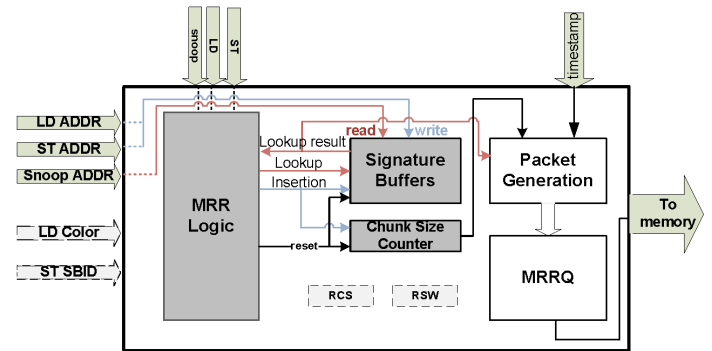


Figure 1: CoreRacer block diagram

On receiving a snoop, the MRR logic looks up the snoop address in the signature buffers. The lookup operation [21] hash encodes the address into a signature with k bits, and then tests if the corresponding bits are set in the signature register. This test operation requires an AND tree, so its timing depends on the size of the signature register. Our model clocks the lookup operation in two cycles.

A successful lookup indicates a conflict, which terminates a chunk and signals the packet generation component to insert a new log entry into a finite sized buffer called the MRR queue (MRRQ). The MRRQ drains to memory using idle memory cycles. After creating the entry, the MRR logic clears the signature buffers and the chunk size counter.

Figure 2 shows the format of a chunk log entry created by the packet generation component. The Type field stores the type of the log entry, the TS field stores a 32-bit timestamp value, the CS field stores the chunk size, and the RSW field stores the re-ordered store window of a chunk (Section 4 gives more details).

The packet generator creates the TS field using the x86 TSC register (available in x86 processors starting from Nehalem). The existing hardware keeps all TSC registers in the system synchronized. This enables CoreRacer to capture chunk ordering without modifying the coherence proto-

Type	TS	CS	RSW
------	----	----	-----

Figure 2: Chunk log entry format

col. This is in contrast to other chunk-based proposals [10, 21], which piggyback a timestamp on coherence messages.

The local timestamp counters run at the same frequency regardless of the CPU clock rate or the various halt or sleep states. When the frequency changes, CoreRacer logs the core-clock to bus-clock ratio with a special packet type. Using simple back-of-the-envelope calculation, one can derive the wall clock time from the logged timestamps and the clock-to-bus ratio.

A software replayer can use CoreRacer’s logs to replay an SC execution. Similar to prior chunk-based proposals [10, 21], the replayer re-executes chunks ordered by their timestamp and replays each instruction in a chunk following program order. A replayer must also handle input non-determinism. Our system handles this using a CAPO-like approach [15], details of which are outside the scope of this paper.

4. HANDLING TSO

This section describes CoreRacer extensions to handle TSO. It first introduces an example that illustrates the problem and then describes the mechanisms for recording and replaying a TSO execution.

4.1 Motivating example

Consider the example in Figure 3(a), which illustrates one possible outcome of executing a program with 2 threads on a multicore TSO system. Figure 3(b) illustrates the corresponding cycle-by-cycle execution, showing how the CoreRacer hardware logs the chunks. In cycles 1 and 2, the stores in cores P1 and P0 retire to processor state and become senior stores (i.e., they wait in the store buffer until they can commit to memory). In cycles 3 and 4, the loads that follow the stores in each core retire, effectively ordering the loads to memory earlier than the prior stores because a load commits to memory before it retires to processor state. When each load retires, its address is inserted into its corresponding processor’s read set and the corresponding chunk size (CS) is incremented. At cycle 6, the store in P0 commits to memory and both the write set and the chunk size are updated accordingly. When P1 receives the snoop caused by P0’s store, CoreRacer detects a conflict in P1’s read set. As a result, CoreRacer logs a chunk ($\langle ts = 1, cs = 1 \rangle$) and resets P1’s chunk size and signatures. In cycle 7, P1’s store commits to memory and terminates P0’s chunk in a similar way. Cycle 8 simply represents the last cycle in the example in which P1’s chunk terminates.

Figure 3(c) shows the resulting chunks and the order in which each memory operation commits to memory (the circled numbers show this order). Figure 3(d) shows the replay order of these chunks, which incorrectly results in $R1 = 1$ and $R2 = 1$. The problem is that the replay order does not match the order in which the loads and stores commit to memory: according to the order in Figure 3(c), the load in P1 commits to memory before its prior store, so during replay, P1’s first chunk should execute the load instead of the store, and its second chunk should execute the store instead of the load.

We can make two observations in this example. First, as long as a load retires in the same chunk as the one in which a prior store commits, there is no potential SC violation because both instructions execute atomically in the same chunk, hence a remote processor cannot observe any re-ordering of the two instructions. This is the case for P0’s chunk where both the load and the prior store commit to memory in the same chunk (Figure 3(c)).

Second, if a load retires in an earlier chunk than the one in which its preceding store commits, then an SC violation is possible. This is because in the global ordering of chunks from different processors, a conflicting remote chunk may interleave between the chunk in which the load retires and the chunk in which its prior store commits, thus exposing the re-ordering to the remote processor. In Figure 3(b), for example, the load in P1’s first chunk retires at cycle 4 but its prior store commits in the second chunk at cycle 7, allowing P0’s chunk to interleave in between and observe the re-ordering.

4.2 The Reordered Store Window

As the previous example illustrates, the problems due to TSO arise when a chunk containing retired loads terminates with stores that have not committed. So to support TSO, CoreRacer tracks the number of pending stores awaiting commit in the store buffer when a load retires. We call this the RSW of the load. When a store commits, the RSW for the load decrements by one until it reaches zero, at which point the ordering between the load and its re-ordered stores becomes consistent with an SC execution.

If at the end of the chunk, $RSW = 0$ for all loads in the chunk or the chunk has no loads, it is called an *SC-chunk*; otherwise, it is a *TSO-chunk*. If all logged chunks are SC-chunks, then no SC violations have occurred because the ordering between the loads of an SC-chunk and all their preceding stores is consistent with an SC execution. Therefore the replay order will match the commit order.

To determine if a chunk is an SC-chunk, it is sufficient to detect that the last load to retire in the chunk has $RSW = 0$ or that the chunk has no loads. This is straightforward to prove: If at the end of the chunk $RSW = 0$ for the last load, then all stores that retired before that load must have committed, and because all other loads in the chunk retired before the last load, then $RSW = 0$ for all loads in the chunk. We refer to the RSW of the last load as the chunk’s RSW, and if a chunk has no loads, then we define its RSW to be zero.

A TSO-chunk’s RSW gives the number of senior stores that retired before the last load of the chunk but are still awaiting commit in the store buffer when the chunk terminates. A TSO-chunk’s size (i.e., its logged CS) does not count these stores — CS only counts retired loads and committed stores that may have retired either in this chunk or in an earlier chunk.

Our offline replay algorithm uses the chunk’s RSW to simulate the store buffer during replay (see Section 4.3). For instance, in Figure 3(b), when CoreRacer logs the first chunk in P1 at cycle 6, it detects that the chunk’s RSW is 1 because there is one prior store awaiting commit in the store buffer when the last load in the chunk retired at cycle 4. When that chunk is replayed in Figure 3(d) (first chunk in P1), the RSW indicates that the store buffer should contain one store waiting to commit. The replayer, therefore, does

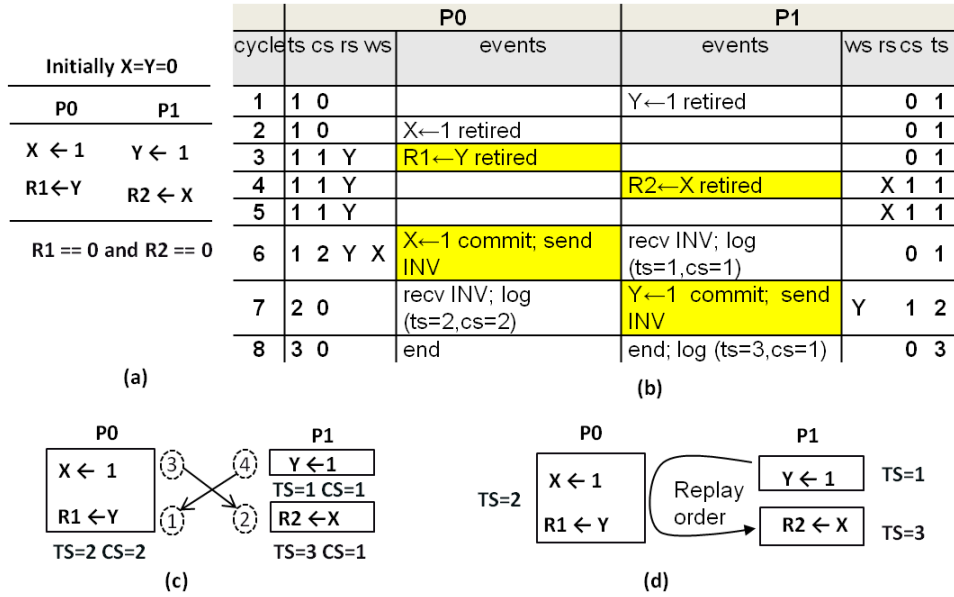


Figure 3: (a) Programmer’s view (b) CoreRacer operational view during execution: load and store addresses update CoreRacer’s cs, rs and ws at retirement and commit time, respectively. CoreRacer clears cs, rs and ws at chunk termination (c) Recorded chunks after the execution (numbers in circles show commit order of memory operations and solid arrows denote cross-processor data dependencies) (d) Chunk replay order (result is $R1 == 1$ and $R2 == 1$)

not execute the first store as part of the chunk and instead keeps it in a simulated store buffer and continues to execute the load. Because the chunk size is 1, no further memory operations are executed for that chunk. The replayer then executes the next chunk (P0’s chunk). Finally, the replayer executes the last chunk from P1, committing the store held in P1’s simulated store buffer.

4.3 Replaying TSO-chunks

Replay executes chunks in timestamped order and holds stores in a simulated store buffer until they can commit. After executing the instructions in a chunk, the replayer drains the simulated store buffer until it has RSW stores left in it.

Algorithm 1 describes how to replay TSO-chunks. The $numLoad$ variable counts the number of load instructions executed in the chunk. The $memInstCnt$ variable tracks the number of memory instructions remaining to be executed in the chunk. It is initialized as shown on line 2. This variable accounts for the loads and stores that will commit in the chunk, and the stores that will retire before the last load in the chunk but that will not commit to memory (i.e., RSW). In addition, senior stores that have been added to the store buffer in previous TSO-chunks, but not yet committed, must be discounted from $memInstCnt$. This is necessary so that the appropriate number of stores are added to the store buffer as the replayer executes the instructions in the chunk. For example, if a TSO-chunk’s RSW is 4 but the store buffer already includes 1 store, then only 3 stores need to be added to the store buffer as $memInstCnt$ instructions are processed.

The TSO-chunk replay algorithm executes instructions until all $memInstCnt$ memory instructions have executed (lines 3 – 15). Loads read their value from the store buffer

on a match, otherwise they read from memory (lines 5 – 9). Line 10 increments the number of committed loads. Stores inserts its value into the store buffer (lines 11–12). Once the TSO-chunk replay algorithm has executed $memInstCnt$ instructions, it verifies that it has committed CS memory instructions from the chunk. At this point, it has already committed all loads (i.e., $numLoad$ loads), therefore any remaining memory instructions to be committed are stores that must be drained from the store buffer. Lines 16 – 19 handle stores that commit in the chunk.

Algorithm 1 TSO-chunk replay algorithm (STB is the store buffer)

```

Require:  $CS, RSW$ 
1:  $numLoad = 0$ 
2:  $memInstCnt = CS + RSW - NumElements(STB)$ 
3: while ( $memInstCnt > 0$ ) do
4:   if (instruction is a load) then
5:     if (load found in STB) then
6:       Get value from STB
7:     else
8:       Read value from memory
9:     end if
10:     $numLoad := numLoad + 1$ 
11:   else if (instruction is a store) then
12:     Insert store into STB
13:   end if
14:    $memInstCnt := memInstCnt - 1$ 
15: end while
16: while ( $CS > numLoad$ ) do
17:   dequeue one store from STB, writing its value to memory
18:    $CS := CS - 1$ 
19: end while

```

4.4 Replaying SC-chunks

Algorithm 2 describes how to replay SC-chunks. An SC-chunk contains either committed stores, or loads and stores that obey SC order (in which case the store buffer is empty

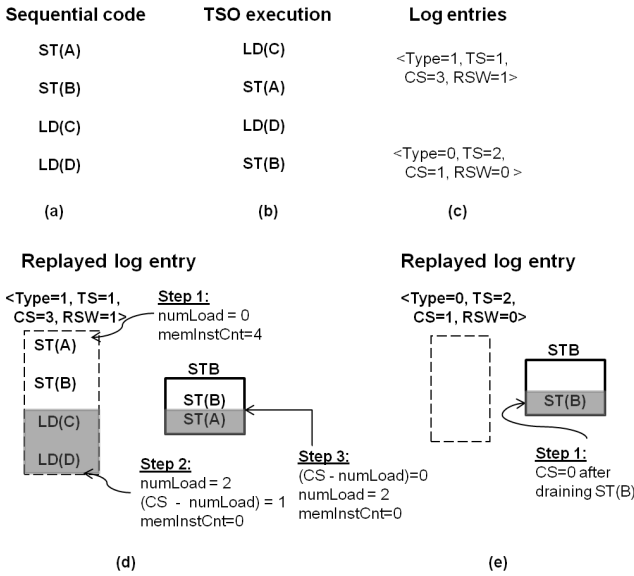


Figure 4: (a) Sequential code (b) TSO execution order (c) Chunks logged (d) TSO-chunk replay (e) SC-chunk replay

at the end of the chunk). Lines 1-4 drain the store buffer based on the chunk size. Line 5 executes any remaining instructions following program order without buffering stores in the store buffer, because the store buffer must be empty at the end of the chunk.

Algorithm 2 SC-chunk replay algorithm (STB is the store buffer)

```

1: while (there are stores in STB) do
2:   drain one store from STB, committing its value to memory
3:   CS := CS - 1
4: end while
5: Execute remaining CS instructions from chunk in program order

```

4.5 Replay example

Figure 4 illustrates an example of how the instructions in a TSO-chunk and an SC-chunk are replayed. The sequential code is shown in Figure 4(a) and one outcome of a TSO execution is shown in Figure 4(b). The chunk log entries are shown in Figure 4(c). The first chunk is a TSO-chunk and the second is an SC-chunk. Note that in this example, $LD(C)$ was reordered before the stores $ST(A)$ and $ST(B)$ and $LD(D)$ was reordered before $ST(B)$ only. At chunk termination, $ST(A)$ was already committed to memory but $ST(B)$ was not, hence $RSW = 1$ for the TSO-chunk.

When replaying the TSO-chunk, as shown in Figure 4(d), $numLoad = 0$ (step 1). Because the store buffer is empty, $memInstCnt = 4$. We then execute 4 instructions, redirecting stores to the store buffer and executing the loads. As a load is executed, $numLoad$ is updated appropriately. After $memInstCnt$ instructions have been executed (step 2), $numLoad = 2$ loads have been processed, while two stores have been inserted in the store buffer. However, $CS - numLoad$ is still one, which means one remaining memory instruction must be drained from the store buffer (step 3).

When the chunk terminates (i.e., we have committed CS

memory instructions), the chunk’s store buffer has $RSW = 1$ store awaiting commit. The resulting instructions’ replayed order in the chunk is $LD(C) \rightarrow LD(D) \rightarrow ST(A)$, while the TSO execution order shown in Figure 4(b) is $LD(C) \rightarrow ST(A) \rightarrow LD(D)$. As described in Section 4.6, the TSO execution order and the replay order are equivalent. Finally, the last chunk is replayed (Figure 4(e)). Because this is an SC chunk, we first drain one instruction from store buffer and decrement CS accordingly. The chunk’s store buffer at the end of the execution is empty.

4.6 Correctness of Replay

When replaying a TSO-chunk, stores commit to memory after the loads, regardless of the execution order between the loads and the stores in the recorded TSO execution. Clearly, if the commit order of the loads and stores in the chunk is such that stores commit after loads, then the replay order provided by Algorithm 1 conforms to the TSO execution.

In the case where the stores do not necessarily commit after the loads in the chunk, the replayer still provides a replay order that is consistent with the TSO execution. The replayer exploits the property of chunk-based execution that guarantees that instructions in a chunk execute in isolation (i.e., they do not have side effects that are visible outside of the chunk). This atomicity guarantee means that instructions within the chunk can be reordered as long as they do not conflict¹. If a load depends on a prior store in the same chunk, then the replayer guarantees that the store buffer supplies the correct value to the load (line 6). In this case, reordering the store after the dependent load is legitimate. If a load is not dependent on a prior store, then reordering the store after the load is also legitimate because these instructions do not conflict.

5. X86 INTEGRATION

This section describes how the CoreRacer hardware module interfaces with an x86 core modeled after a P6-based [24] Nehalem system, and how it handles some of the complex aspects of the x86 ISA. The processor system in this study has a three-level cache hierarchy with private L1 and L2 caches, and a shared L3 cache. The L2 data cache is inclusive of the L1. The snoop coherence protocol implements an invalidation-based MESI protocol with L2 as the point of coherence.

5.1 Handling loads and stores

Figure 5 shows how the CoreRacer module interfaces with the cache and core for handling loads, stores, snoops, and evictions. To observe load and store addresses, CoreRacer monitors the execution of memory operations in the memory order buffer (MOB), composed of a store buffer (STB) and a load buffer (LDB). The STB holds in-flight stores until they commit to memory, at which point their address is sent to CoreRacer for insertion into the write signature. The commit time for a store, which corresponds to the time the store data becomes globally visible, occurs after the store has retired to processor state. At retirement, a store is promoted to a senior store and waits in the STB until it commits to memory in program order.

Similar to the STB, the LDB tracks in-flight loads until they retire to processor state, at which point their address is

¹The BulkCompiler [1] also exploits this property.

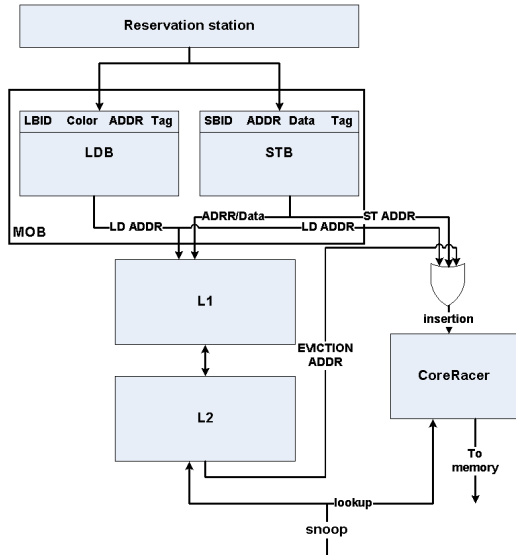


Figure 5: Memory order buffer and cache interfaces to CoreRacer

sent to CoreRacer for insertion into the read signature. The retirement point of a load occurs after the load commits to memory with respect to prior memory operations. This corresponds to the completion of a L1 cache access, indicated by a valid bit in the tag field of the corresponding load entry in the LDB. Once completed, the processor retires the load as soon as it becomes the oldest instruction in the ROB.

When the L2 cache evicts a line, CoreRacer looks up the line address in its signatures and terminates the chunk on a match because a core may not receive subsequent snoops for an evicted line. There exist techniques to mitigate this problem but they introduce additional complexity [21].

5.2 RSW calculation

To obtain RSW, we leverage an existing mechanism (called store coloring) found in modern x86 implementations for ordering loads and stores executing in the same core. Store coloring (also known as aging) colors each load that issues to the LDB with the *store buffer ID* (SBID) of the youngest store that executes prior to the load in program order. Given the color of a load (i.e., the SBID of the senior store immediately preceding the load in program order) and the SBID of the most recently committed store, the difference between these two SBIDs (modulo wrap around of SBIDs) is the number of senior stores in the store buffer when the load retires, which is also the number of prior stores that will commit to memory after the load.

To compute a chunk’s RSW, the CoreRacer module (Figure 1) contains two registers:

1. An RSW register that tracks the RSW value for the most recently retired load. When a chunk terminates, this register contains the chunk’s RSW. CoreRacer clears this register at the start of each chunk ($RSW == 0$ at the start of a chunk).
2. An RCS (Recent Committed Store) register that tracks the SBID of the most recently committed store.

The interface to the CoreRacer module (Figure 1) also has two inputs that support computing a chunk’s RSW:

the SBID of a committing store and the color of a retiring load. As the STB and LDB already have these values, the MOB simply passes them to CoreRacer alongside a load or store address. CoreRacer, therefore, does not modify existing components in the core to support TSO.

As Algorithm 3 shows, each time a load retires, CoreRacer recomputes the value of RSW by subtracting the value in RCS from the load’s color (adjusting for wrap around of the SBID). In line 4 of the algorithm, STB_SIZE denotes the size of the store buffer.

Algorithm 3 CoreRacer operations for a retiring load

Require: color
1: if ($color \geq RCS$) then
2: $RSW := color - RCS$
3: else
4: $RSW := STB_SIZE + color - RCS$
5: end if

As Algorithm 4 shows, each time a store commits, CoreRacer updates the RCS register with that store’s SBID. It also decrements the value in the RSW register if it is not zero — if the RSW value is not zero, then this store must have been re-ordered with respect to the most recently retired load in the current chunk and that load is now re-ordered with respect to one fewer store. If the decrement causes the RSW value to reach 0, then there are no more senior stores awaiting commit in the STB.

Algorithm 4 CoreRacer operations for a committing store

Require: SBID
1: $RCS := SBID$
2: if ($RSW > 0$) then
3: $RSW := RSW - 1$
4: end if

In our model, the STB has 32 entries, so the inputs and registers required to compute RSW are all 5 bits wide. Computing RSW requires negligible logic: each time a store commits, RSW is decremented if its not zero, and each time a load retires, RSW is updated by subtracting RCS from the load’s color and adding STB_SIZE on a wrap around.

5.3 Interrupts and CPL changes

CoreRacer currently records the interleavings of only user-level memory operations. When execution transitions to a higher privilege level (e.g., CPL=0) due to either a context switch or an interrupt, the hardware terminates the current chunk and disables CoreRacer. Transitions due to a context switch flush the MRRQ to memory using privileged software. This allows CoreRacer to be virtualized across several applications. CoreRacer resumes operations when execution returns to a user mode application that has turned on recording. Application software can enable CoreRacer by programming specific MSR registers.

5.4 Complex x86 instructions

In x86, multi-line operations and multi-operation instructions are those that touch multiple cache line (i.e., straddle cache line boundaries) or need several memory operations to complete (e.g., string moves). The hardware usually breaks

these (macro) operations down into multiple micro operations. Violation of instruction atomicity [20] occurs when a chunk terminates in the middle of such a macro operation. In this case, CoreRacer generates a packet that contains the number of bytes that have been partially transferred by the last macro operation in the chunk. This packet indicates the number of bytes that this macro operation will have to access during replay.

5.5 Memory address types

x86 processors implement three main types of memory: Write-back (WB), Uncachable (UC), and Write-combining (WC). Accesses to WB memory are cachable, while accesses to the other memory types are not. CoreRacer supports only WB memory as most programs use only this memory type. Accesses to non-WB memory terminate the current chunk and create a log entry.

6. EVALUATION

We implemented CoreRacer on a cycle accurate, execution-driven simulator that runs x86 binaries. We modeled an 8-core multiprocessor system in which each core is a P6-like CPU. The memory model implemented in the x86 simulator is TSO. We extended the simulator to model the design of CoreRacer as described in Section 3. Table 1(a) summarizes the configuration parameters of the x86 core and CoreRacer.

We evaluated CoreRacer using the benchmark set described in Table 1(b). These benchmarks were represented in a proprietary checkpoint format, describing the processor architectural state and the state of the system memory, and were used to feed our simulator. Our experiments used checkpoints of approximately 90 million instructions for each application in the benchmark set. To capture the overall application behavior, these traces were created using SimPoint [25]. We also used a fixed cache warm-up file to warm-up the caches before each simulation.

6.1 Performance overhead of recording

Figure 6(a) shows the recording time overhead, which accounts for the cost of creating and sending log entries to memory. Our baseline design uses a MRRQ with 4 entries, where each entry is 8 bytes wide. As shown by Figure 6(a), the recording overhead is relatively low. This is because most of the applications we studied have a low logging frequency, with the exception of Oracle. This low logging frequency translates into the bandwidth requirement shown in Figure 6(b). The average bandwidth requirement for an application in our benchmark set is 0.43 byte per kilo instruction, while the peak bandwidth is about 2.36 bytes per kilo instruction for Oracle. In our system, if the log entries are inserted into MRRQ faster than they can be drained to memory, the processor stalls and stops accepting further snoop requests from other cores until enough entries are drained to continue normal execution. However, this situation is unlikely to happen due to the low-bandwidth requirement of CoreRacer logs. Our measurements show that stalling the machine in this case accounts for less than 0.1% of total simulation cycles.

6.2 TSO-chunk distribution

Figure 7(a) shows the percentage distribution of logged TSO-chunks and SC-chunks. In general, the percentage of logged TSO-chunks is low for most programs, except for

Art (41%) and *Galgel* (33%). Figure 7(b) provides greater insight into the cause of TSO-chunks. It shows that the proportion of TSO-chunks logged due to eviction is high. It also shows that SC violations resulting from WAR dependencies are rare, except for Oracle where it accounts for almost 17% of the TSO-chunks. Note that, in Figure 7(b), PageRank and SVM do not have any TSO-chunks that cause SC violations. This is not an erroneous finding. Instead, it simply means that when the chunk was terminated, there were one or more pending stores in the STB awaiting commit, and each one of these stores ultimately committed to memory without leading to an ordering violation.

6.3 Log size increase due to TSO-chunks

CoreRacer’s SC-chunk is similar in size to a chunk in Intel’s MRR [21] or an episode in Rerun [10]. On average, TSO-chunks represent less than 20% of chunks (Figure 7(a)) and each TSO-chunk requires 5 more bits than an SC-chunk. Compared to Figure 6(b), this translates to a bandwidth of 0.39 byte per kilo instructions without TSO-chunks (i.e., a bandwidth reduction of 9% without TSO-chunks).

6.4 RSW size distribution

Figure 8(a) shows the cumulative distribution of the RSW size for the TSO-chunks shown in Figure 7(a). The distribution varies across benchmarks and shows that, on average, TSO-chunks have an RSW < 7 , 90% of the time. The RSW cannot be larger than the size of the STB, which is 32 in our case. For CoreRacer, a large RSW translates into replay overhead because the store buffer-like structure must be searched for each load, regardless of the existence of an SC violation. Figure 8(b) shows the histogram of the RSW size for TSO-chunks terminated due to an SC violation. In most cases, an SC violation occurs with an RSW of 1. Oracle is an exception and has SC violations with an RSW of up to 15.

6.5 Replay performance

We created a prototype replayer using PIN. Our replayer revealed three sources of overhead compared to a replayer that replays only SC-chunks: (1) searching the STB for forwarding a store value to loads; (2) redirecting stores to a thread-local STB; and (3) copying the store values from the STB to global memory. On average, this overhead slows down replay by 54%. In the worse case, the overhead peaks to 89% in *Art*.

7. RELATED WORK

There has been a notable amount of recent activity developing HW support for recording memory races. Most of the proposed approaches, however, have only considered the SC memory model. FDR [28], BugNet [17], and Strata [16] are some techniques that record the ordering between memory operations on SC. While novel, these approaches have technical limitations that may make their adoption by commercial processors, such as x86 that implements TSO, challenging. To the best of our knowledge, there exists only a few proposals that address the problem of recording memory races on non-SC memory models.

RTR [29] is one of the first approach to address this issue for TSO. RTR is layered on top of a point-to-point approach that records the interleaving between memory operations. As discussed in Section 2, point-to-point logging mechanism

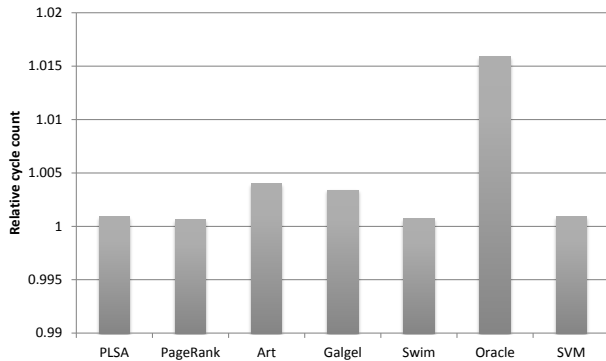
(a) System parameters

Number of cores	8-core P6-type
Clock	2.2GHz
STB/LDB size	32/48
L1 Dcache	32KB private,8-way assoc,64B linesize
L2 cache	256KB private,16-way assoc,64B linesize
L3 cache	8MB shared,16-way assoc,64B linesize
Cache Coherence	Invalidation-based MESI
Memory model	TSO
Signatures	1024-bits read set 512-bits write set 4 XOR-based hash functions

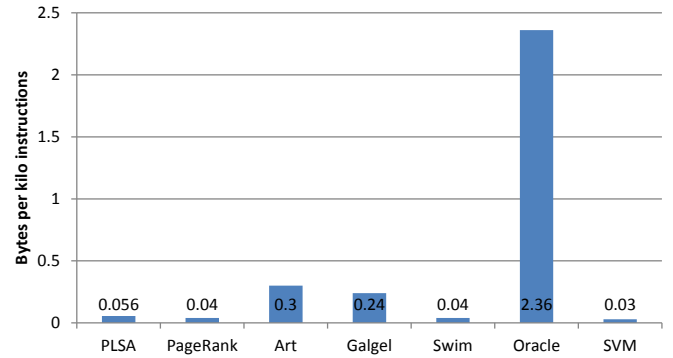
(b) Benchmarks

Domain	Application	Description
Bioinformatics	PLSA	Used to identify differences between two genetic sequences
SPEComp 2001	Art	Image Recognition / Neural Networks
	Galgel	Computational Fluid Dynamics
	Swim	Shallow Water Modeling
Search Engine	PageRank	Google-like search engine for ranking documents
	SVM	Learning methods used for classification and regression
Database	Oracle TPCC	Online Transaction Processing

Table 1: System parameters and benchmarks.

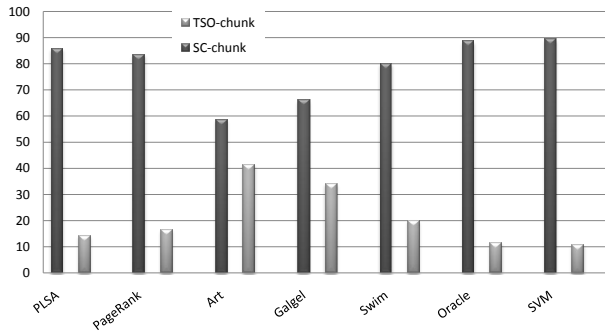


(a) Recording time overhead normalized to execution w/o CoreRacer

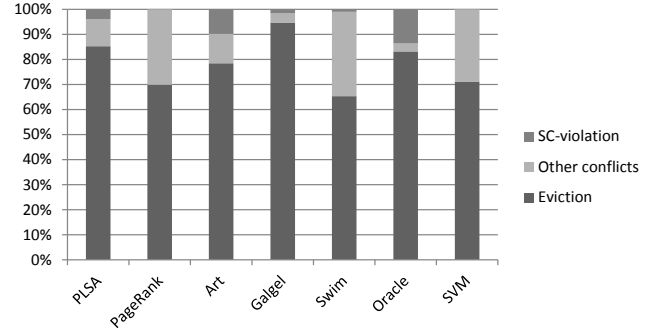


(b) Bandwidth

Figure 6: CoreRacer Overhead



(a) TSO-chunks and SC-chunks distribution

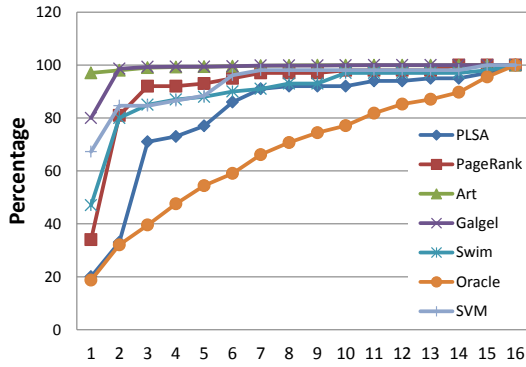


(b) Reasons for logging TSO-chunks

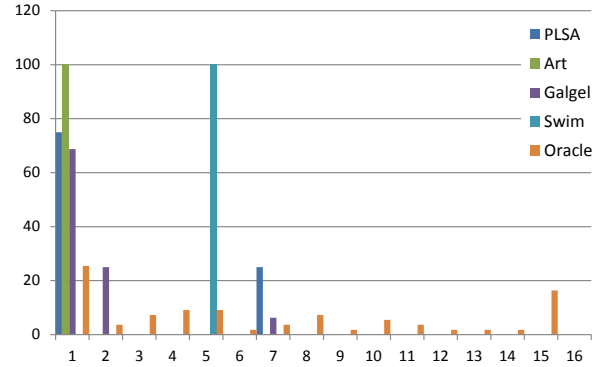
Figure 7: Chunk Analysis

already requires a major overhaul of the cache and coherence protocol. RTR adds more to it by detecting SC violations dynamically. The idea is to monitor the cache line of a load that is reordered before a prior store and log the value of that load if the line is invalidated by a remote store, before the store that was passed by the load commits to memory. This mechanism requires hardware for monitoring the cache lines and for buffering the load values and their addresses. For instance, if a monitored line is evicted from the cache before the store that was passed by the load commits to memory, then the address of the load and its value must be buffered

so that they can be compared against a remote store for detecting a potential SC violation. In general, since SC violations are likely to occur less frequently than other events – in our experiments, TSO-chunks terminated due to an SC violation represent respectively 4%, 0.5%, 0.2%, 1.85%, and 0.55% of the chunks in Art, Galgel, Swim, Oracle and PLSA –, we are of the opinion that such an overhaul of the cache subsystem and coherence protocol is not warranted for adoption by commercial processors. Our approach removes this complexity by extending a chunk entry with the RSW of the last load to retire in the chunk. We showed how a replayed



(a) Cumulative distribution of RSW size



(b) RSW size histogram for TSO-chunks due to SC violation

Figure 8: Reordered Store Window (RSW)

execution uses the RSW to reconstruct an execution that conforms to the TSO execution. We showed that this approach does not change the cache or the coherence protocol. In the same category as RTR is LReplay [7], which proposes to log SC violating loads using a large CAM structure.

Rerun [10], Intel MRR [21], Karma [3] and DeLorean [14] are additional techniques that use chunks for detecting the interleaving of shared memory operations. Rerun and Intel MRR apply to SC, while Karma leverages RTR to detect SC violations dynamically. Our approach can complement them in many ways. First, our approach is a straightforward extension of Rerun, Intel MRR and Karma for TSO. There is no requirement for changing the core. Second, our approach improves Rerun, Intel MRR and Karma by using the TSC, which removes the constraint of changing the cache coherence protocol in these approaches. DeLorean can record a TSO execution because it assumes the Bulk architecture [6], which advocates a complete overhaul of the hardware. In Bulk, there is a central arbiter that implements the recording by logging the commit ordering of chunks. Our approach is geared toward influencing commercial x86 processors, which do not implement Bulk.

A recent work by Lee et al. [13] proposed an approach similar to ours in the sense that they also deal with TSO offline. Their approach is layered on top of a load-value based mechanism where the address and value of cache misses are logged in order to replay each thread individually. This is done by augmenting the cache with some status bits. Given the traces of addresses and load values for each thread, an offline symbolic analysis can then be specified ordering constraints so that a valid causal order of a TSO execution is unveiled offline. To reduce the search space of the symbolic analysis, the mechanism augments the log record with additional information such as pending stores in the store buffer or committed memory operations. Our approach is different because we do not assume any changes to the cache and we do not log any load address or value. In addition, our log can immediately lead to a valid TSO execution order, without resorting to symbolic analysis.

8. CONCLUSION

In this paper, we described CoreRacer, a memory race recorder architecture for multi-core x86 TSO processors. We presented a detailed design of CoreRacer and explained why

its design is appealing for integration into commercial processors. CoreRacer’s key advantages over other memory race recorder systems are that it does not require any changes to existing cache or coherence protocols, yet, it integrates seamlessly into the x86 TSO memory model.

CoreRacer handles TSO by recording RSW, which is the number of pending stores in the store buffer when a chunk terminates, and by simulating the store buffer during replay using RSW. We showed that this information is enough to reproduce a TSO execution. CoreRacer also leverages a specific x86 feature, the invariant timestamp, to maintain ordering between chunks without piggybacking on coherence messages. Because the components of CoreRacer are confined within a single block module, it is convenient to integrate in a high-end processing environment, such as a desktop, or a low-end processing environment, such as a SoC.

9. ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their valuable comments and suggestions for improving this paper.

10. REFERENCES

- [1] W. Ahn, S. Qi, M. Nicolaidis, J. Torrellas, J.-W. Lee, X. Fang, S. Midkiff, and D. Wong. BulkCompiler: High-performance Sequential Consistency Through Cooperative Compiler and Hardware Support. In *Proceedings of the International Symposium on Microarchitecture*, 2009.
- [2] G. Altekar and I. Stoica. ODR: Output-deterministic Replay for Multicore Debugging. In *Symposium on Operating System Principles*, 2009.
- [3] A. Basu, J. Bobba, and M. D. Hill. Karma: Scalable Deterministic Record-Replay. In *International Conference on Supercomputing*, 2011.
- [4] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. *ACM Trans. Comput. Syst.*, 14:80–107, February 1996.
- [5] R. H. Carver and K.-C. Tai. Replay and testing for concurrent programs. *IEEE Softw.*, 8:66–74, March 1991.
- [6] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval. Bulk Disambiguation of Speculative Threads in

- Multiprocessors. In *Proceedings of the International Symposium on Computer Architecture*, 2006.
- [7] Y. Chen, W. Hu, T. Chen, and R. Wu. LReplay: a pending period based deterministic replay scheme. In *Proceedings of the International Symposium on Computer Architecture*, 2010.
- [8] G. Dunlap, S. King, S. Cinar, M. Basrai, and P. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In *OSDI*, 2002.
- [9] G. Dunlap, D. Lucchetti, M. Fetterman, and P. Chen. Execution replay of multiprocessor virtual machines. In *International Conference on Virtual Execution Environments*, 2008.
- [10] D. Hower and M. Hill. Rerun: Exploiting episodes for lightweight memory race recording. In *Proceedings of the International Symposium on Computer Architecture*, 2008.
- [11] Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Chapter 10, 2010.
- [12] S. T. King, G. W. Dunlap, and P. M. Chen. Operating System Support for Virtual Machines. In *Proceedings of the USENIX Annual Technical Conference*, 2003.
- [13] D. Lee, M. Said, S. Narayanasamy, and Z. Yang. Offline symbolic analysis to infer total store order. In *Proceedings of the International Symposium on High-Performance Computer Architecture*, 2011.
- [14] P. Montesinos, L. Ceze, and J. Torrellas. DeLorean: Recording and Deterministically Replaying Shared-Memory Multiprocessor Execution Efficiently. In *Proceedings of the International Symposium on Computer Architecture*, 2008.
- [15] P. Montesinos, M. Hicks, S. King, and J. Torrellas. Capo: A Software-Hardware Interface for Practical Deterministic Multiprocessor Replay. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2009.
- [16] S. Narayanasamy, C. Pereira, and B. Calder. Recording shared memory dependencies using strata. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.
- [17] S. Narayanasamy, G. Pokam, and B. Calder. BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging. In *Proceedings of the International Symposium on Computer Architecture*, 2005.
- [18] 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 *Symposium on Operating System Principles*, 2009.
- [19] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie. PinPlay: a Framework for Deterministic Replay and Reproducible Analysis of Parallel Programs. In *CGO*, 2010.
- [20] C. Pereira, G. Pokam, K. Danne, R. Devarajan, and A.-R. Adl-Tabatabai. Virtues and obstacles of hardware-assisted multi-processor execution replay. In *HotPAR*, 2010.
- [21] G. Pokam, C. Pereira, K. Danne, R. Kassa, and A.-R. Adl-Tabatabai. Architecting a Chunk-based Memory Race Recorder in Modern CMPs. In *Proceedings of the International Symposium on Microarchitecture*, 2009.
- [22] G. Pokam, C. Pereira, K. Danne, L. Yang, S. King, and J. Torrellas. Hardware and software approaches for deterministic multi-processor replay of concurrent programs. *Intel Technology Journal*, 13(4), Dec. 2009.
- [23] D. Sanchez, L. Yen, M. D. Hill, and K. Sankaralingam. Implementing signatures for transactional memory. In *Proceedings of the International Symposium on Microarchitecture*, 2007.
- [24] J. P. Shen and M. H. Lipasti. *Modern Processor Design: Fundamentals of Superscalar Processors*. McGraw-Hill Higher Education, 2005.
- [25] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.
- [26] S. Srinivasan, S. Kandula, C. Andrews, and Y. Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *Proceedings of the USENIX Annual Technical Conference*, 2004.
- [27] VMWare. vsphere availability guide.
- [28] M. Xu, R. Bodik, and M. Hill. A flight data recorder for enabling full-system multiprocessor deterministic replay. In *Proceedings of the International Symposium on Computer Architecture*, 2003.
- [29] M. Xu, R. Bodik, and M. D. Hill. A Regulated Transitive Reduction (RTR) for Longer Memory Race Recording. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.