Timetraveler: Exploiting Acyclic Races for Optimizing Memory Race Recording

Gwendolyn Voskuilen*, Faraz Ahmad* and T. N. Vijaykumar (* both primary student authors)
School of Electrical and Computer Engineering, Purdue University
{geinfeld,fahmad,vijay}@ecn.purdue.edu

ABSTRACT

As chip multiprocessors emerge as the prevalent microprocessor architecture, support for debugging shared-memory parallel programs becomes important. A key difficulty is the programs' non-deterministic semantics due to which replay runs of a buggy program may not reproduce the bug. The non-determinism stems from memory races where accesses from two threads, at least one of which is a write, go to the same memory location. Previous hardware schemes for memory race recording log the predecessor-successor thread ordering at memory races and enforce the same orderings in the replay run to achieve deterministic replay. To reduce the log size, the schemes exploit transitivity in the orderings to avoid recording redundant orderings. To reduce the log size further while requiring minimal hardware, we propose *Timetraveler* which for the first time exploits acyclicity of races based on the key observation that an acyclic race need not be recorded even if the race is not covered already by transitivity. Timetraveler employs a novel and elegant mechanism called *post-dating* which both ensures that acyclic races, including those through the L2, are eventually ordered correctly, and identifies cyclic races. To address false cycles through the L2, Timetraveler employs another novel mechanism called *time-delay buffer* which delays the advancement of the L2 banks' timestamps and thereby reduces the false cycles. Using simulations, we show that Timetraveler reduces the log size for commercial workloads by 88% over the best previous approach while using only a 696-byte time-delay buffer.

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

Race Recording, Debugging, Determinism, Replay.

1. INTRODUCTION

Chip multiprocessors (CMPs) are emerging as a better alternative to uniprocessors in terms of power dissipation and performance. However, CMPs run shared-memory parallel programs which are significantly harder to debug than sequential programs due to the programs' non-deterministic semantics. That is, multiple runs of a buggy shared-memory program may result in the bug not manifesting. The reason for the non-deterministic semantics is that *memory races* among threads via shared memory — i.e., accesses to a memory location at least one of which must be a write — change based on the interleaving of the threads. If the interleaving changes due to variations in thread scheduling or access latencies (e.g., due to differences in cache behavior induced by changes in thread-core mappings), then the memory races change resulting in a different program outcome (e.g., lack of bug manifestation).

One approach to address this problem is to provide system support for deterministic replay which achieves the same memory race order, and hence produces the same (buggy) execution as the original run. System support for such replay handles two aspects: (1) a checkpoint state of the program from which a replay may start (to aid in debugging, this checkpoint should capture the state before the bug); and (2) additional state to capture the *predecessor-successor ordering* of the threads involved in each memory race (i.e., a load from one thread occurs before or after a store from another thread). While the former aspect is well-studied (e.g., [4,25,28]), the latter aspect has recently been receiving attention. Because such memory races always involve a write to shared memory and because cache coherence generates global events for such memory races (i.e., write invalidations and read misses), hardware support leverages cache coherence to capture the ordering. Because the hardware treats alike all memory races — synchronization accesses and data races, such schemes are applicable to programs with and without data races.

While previous work has proposed both centralized [16,18] and distributed schemes [30,31], we explore a distributed scheme due to its scalability (see Section 2). In distributed schemes, the hardware records the predecessor-successor ordering among the racing accesses by implementing Lamport's clocks [13]. A per-thread *log* in the program's memory records the races (i.e., no special hardware structure). The schemes incur two significant overheads: (1) large log sizes and (2) hardware overhead of the per-block instruction count in the L1 cache. Reducing the log size enables longer debugging traces and reducing hardware overhead facilitates widespread adoption.

Targeting the first overhead, previous work [20,30] exploits transitivity in the ordering among threads to avoid logging redundant orderings (e.g., if thread *A* is already ordered after thread *B* by an earlier recorded race, then another memory race that implies the same ordering need not be recorded). Other transitivity-based optimizations have also been proposed [31]. Targeting the second overhead, a recent distributed scheme, *Rerun* [12], builds on Strata [18] to eliminate the per-block instruction count in the L1 caches. Rerun partitions a thread's dynamic instruction stream into *episodes*, each of which is an atomic sequence of instructions that ends at a *current-race*. We call a memory race a *past-race* if the race is between previously-ended predecessor episode(s) and a currently-live successor episode, and a *current-race* if the race is among currently-live predecessor and successor episodes. Rerun exploits the fact that past-races are covered by transitivity and need not be recorded, whereas current-races are not covered. Accordingly, Rerun's episodes may include several past-races. While previous schemes use a full-blown timestamp per L1 block to establish thread ordering, Rerun needs only to distinguish whether the block was accessed in a previously-ended episode (for a past-race) or in a currently-live episode (for a current-race), and need not know the exact access time of the block. Rerun combines this distinction with a single timestamp per thread to order the episodes. Rerun's *binary* distinction requires significantly less hardware than the previous schemes (e.g., read/write bits per block or Bloom filters).

We propose *Timetraveler* to achieve smaller logs than Rerun while requiring minimal hardware by exploiting, for the first time, *acyclicity* of races. Bringing this completely new property to bear on the problem of memory race recording is the key intellectual novelty of this paper. Timetraveler addresses three key limitations of Rerun via two novel mechanisms. The first limitation is that the predecessor episode ends *immediately* upon a current-race to address the potential threat that the predecessor's timestamp will later advance past the successor's timestamp. Such ending prevents longer episodes and hence increases the log size. Instead, we make the key observation that episodes may include multiple current-races as long as (1) the predecessor and successor episodes involved in every such race can *eventually* be ordered properly, and (2) the predecessor-successor ordering imposed by the current-races on currently-live episodes is not cyclic (e.g., episode *B* is ordered after episode *A,* and *A* needs to be ordered after *B*). Clearly, cyclicity would make predecessor-successor order ill-defined. Accordingly, Timetraveler partitions a thread's dynamic instruction stream into *chapters,* each of which ends only when the threads appear to be involved in a cyclic, current-race. Thus, a chapter is an atomic sequence of instructions that may include several acyclic, current-races. While previous schemes benefit from transitivity because current-races are rarer than past-races, we observe that cyclic current-races are even rarer, enabling Timetraveler to achieve longer chapters, and hence smaller log.

Timetraveler's first novel and elegant mechanism, called *post-dating*, satisfies both the constraints of our observation. For the first constraint, the predecessor chapter provides a *post-dated* timestamp to the successor chapter with the guarantee that in the future the predecessor chapter's timestamp will not advance beyond the post-dated timestamp; essentially, post-dating creates some "breathing room" for the predecessor chapter. The successor chapter advances its own timestamp beyond the predecessor's post-dated timestamp, guaranteeing that the successor is ordered after the predecessor and neither needs to end its chapter. Cyclic ordering would imply that a chapter has to advance beyond its own post-dated timestamp. Thus, post-dating easily detects when the second constraint is about to be violated and forces the offending chapter to end. Post-dating's elegance stems from achieving the above functionality while adding merely one post-dated timestamp register per core and not any complicated cycle-detection hardware as one might expect. While post-dating handles memory races via L1-resident blocks, the chapters would be shortened by Rerun's second limitation of ending an episode upon the eviction of an L1 block accessed by the episode. Rerun conservatively ends the episode because current-races via the block may go undetected in the shared L2 due to lack of coherence (if the L2 employs coherence then this problem would occur at memory). We again employ post-dating to overcome this limitation. Upon evicting a current-block, a chapter does not end and instead sends its post-dated timestamp to the L2 bank, guaranteeing that any successor is ordered after the predecessor (evicting thread).

Although post-dating handles races via both L1-resident blocks and L1-evicted blocks, L1 evictions cause another problem which is Rerun's third limitation. To ensure correct ordering of races via the L2 while avoiding the overhead of per-block timestamps, Rerun employs a single timestamp for each L2 bank which inherits the latest timestamp among the in-coming, evicted L1 blocks. If Timetraveler were to employ this scheme, a chapter that hits in an L2 bank would advance its timestamp beyond the bank's timestamp to ensure proper ordering with whichever previous chapter had last accessed the block. Unfortunately, L1 eviction of a single recently-accessed block would force the *entire* L2 bank to advance to the evicted block's timestamp, even if all the other blocks in the bank were accessed at much older timestamps. Such advancement would often cause the bank's timestamp to exceed a chapter's post-dated timestamp, inducing false cycles upon L2 hits and forcing the chapter to end. We make the key observation that if L1 evictions are delayed from advancing the L2 bank's timestamp, the current chapters' post-dated timestamps would not be exceeded and the false cycles would be prevented. To this end, we propose a per-L2-bank *time-delay buffer*, Timetraveler's second novel mechanism, to delay the advancement of the L2 bank's timestamp by holding a few recent L1 evictions.

Timetraveler's key contributions are:

- While previous schemes exploit transitivity to reduce the log size, Timetraveler is the first to exploit acyclicity of races.
- Timetraveler proposes two novel and elegant mechanisms, post-dating and delay buffers, which achieve reduction in the log size while incurring *minimal* hardware overhead;
- Using simulations, we show that Timetraveler reduces the log size compared to Rerun by 88% and 99% for commercial and scientific benchmarks, respectively, while adding over Rerun only a 696-byte time-delay buffer and two 32-bit registers for post-dated timestamp and past timestamp per core.

The rest of the paper is organized as follows. We discuss related work in Section 2. We describe Timetraveler and its mechanisms, post-dating and time-delay buffer, in Section 3. Section 4 describes our experimental methodology. We show our results in Section 5, and conclude in Section 6.

2. RELATED WORK

Bacon and Goldstein [6] pioneered the idea of recording memory races in hardware. They record races by logging coherence messages on the snoopy bus. Recent schemes exploit Netzer's transitivity [20] to reduce the log size by avoiding recording memory races whose ordering is implied by transitivity. These schemes include centralized and distributed implementations. Strata [18] proposes the key idea of exploiting race-free instruction sequences which are captured in *strata*, each of which contains the memory-access count of *all* the threads between the last and the current-race. Strata uses a centralized monitor to log each stratum. Unfortunately, Strata's log size is large and does not scale well with thread count [12]. DeLorean [16] proposes a novel chunk-based scheme where a *chunk* is a memory-race-free sequence of instructions in a thread (similar to episodes in Rerun). DeLorean's basic version records the chunk size and the ordering of the chunks in a race, achieving log sizes similar to Rerun. DeLorean employs two optimizations, *OrderOnly* and *PicoLog*, where, respectively, the chunk size is pre-defined but the thread ordering is logged, and both the chunk size and thread ordering are pre-defined. In OrderOnly, any memory race before the pre-defined chunk size is reached, causes one of the threads to roll back. In PicoLog, because even the thread ordering in a race is pre-defined, any race involving a different order causes a rollback. By not recording chunk sizes and thread ordering, these optimizations produce much smaller logs than Rerun. However, DeLorean commits the chunks via a centralized arbiter which cannot scale to a large number of threads, and the rollbacks require substantial hardware support for memory speculation (via BulkSC [8]) which does not exist in conventional CMPs.

The distributed schemes, on the other hand, are more scalable as each thread locally logs the thread ordering for its memory races using Lamport clocks [13]. However, to determine the ordering, FDR-1 [30] and FDR-2 [31] incur significant overhead in the L1 caches. Each cache block has an instruction count of the time of the thread's last access to the block. FDR-1 and FDR-2 log the following upon a memory race at a block: (a) the predecessor thread's instruction count at the time of its access to the block as recorded in its L1 cache, (b) the predecessor thread's identifier, and (c) the current instruction count of the successor thread. This information is sufficient to enforce race ordering in a replay run. Rerun [12] improves upon FDR-1 and FDR-2 to produce comparable log sizes while significantly reducing the hardware overhead — i.e., the per-block instruction counts in the L1 cache. We contrast to Rerun throughout the paper.

BugNet [19] is a software development tool to replay user code and shared libraries and uses FDR-1's hardware. CORD [21], a race recording and detection scheme, uses the concept of scalar clocks to improve performance and race detection accuracy over previous schemes. ReEnact [22], a memory race detector, leverages thread level speculation to rollback and reexecute upon data race detection. ReEnact explores the interesting idea where subsequent re-executions use a library of race signatures to characterize or even eliminate the detected race. Unfortunately, in addition to needing considerable hardware support, this approach requires programmers to identify data races, and incurs performance overhead even during race-free execution.

3. TIMETRAVELER

Recall from Section 1 that Timetraveler exploits, for the first time, acyclicity of races to address Rerun's three key limitations and reduce the log size. To this end, Timetraveler introduces *chapters*, each of which is an atomic instruction sequence that ends at a *cyclic*, *current* memory race. We call a memory race a *past-race* if the race is between previously-ended predecessor chapters and a currently-live successor chapter, and a *current-race* if the race is among currently-live predecessor and successor chapters. Current-races are called *cyclic* if the predecessor-successor ordering imposed by the races on currently-live chapters is cyclic (e.g., chapter *B* is ordered after chapter *A,* and *A* needs to be ordered after *B*). While Timetraveler's chapters do not end on past-races similar to Rerun's episodes, a chapter may include several, acyclic current-races, unlike Rerun's episodes which end at a current-race.

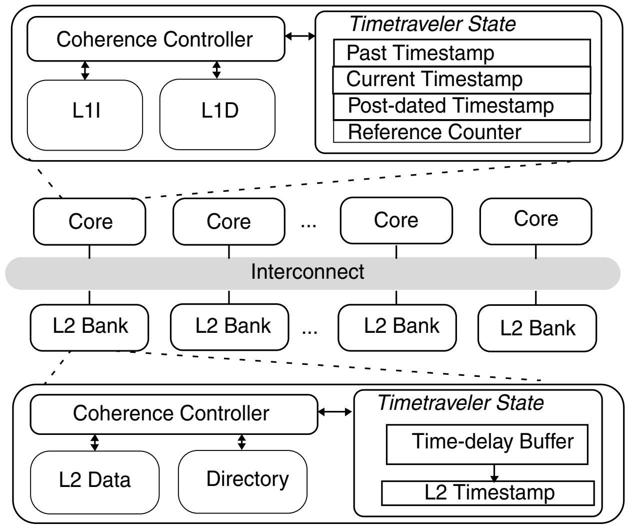

Figure 1. Timetraveler architecture

We describe Timetraveler's implementation as an extension of Rerun's implementation while pinpointing the similarities and differences. We emphasize that Timetraveler is amenable to be implemented as a simple extension of Rerun even though Timetraveler exploits the completely new property of acyclicity of races, illustrating the elegance of Timetraveler's key mechanisms, post-dating and time-delay buffer.

Like Rerun, each thread has a timestamp which is incremented upon the end of a chapter; a chapter's timestamp may also advance as the chapter proceeds, as we see below. Timetraveler logs the chapter's length and final timestamp which represents the chapter's start time in a replay run so that at replay, a chapter starts after the chapters with lower timestamps from *all* the threads have ended. Each log entry corresponds to a chapter and is of the form <*TS, REFS*> where *TS* is the chapter's timestamp and *REFS* encodes the chapter's length as the memory-reference count in the chapter. Because non-memory instructions are purely functional in nature (i.e., deterministically repeat the same behavior in any execution as long as their inputs are the same), non-memory instructions need not be counted in the chapter length. This observation reduces *REFS* and hence the log size.

Figure 1 presents Timetraveler's architecture which we describe via its two novel mechanisms: post-dating and time-delay buffer. Our discussions assume sequential consistency where all memory accesses occur atomically and in program order, so that replay runs also make accesses in the same order. We leave applying Timetraveler to weaker consistency models for future work.

3.1 Post-dating

There are three kinds of races relevant to Timetraveler: past, acyclically current, and cyclically current. Because Timetraveler ends its chapters upon only the third kind and not the first two kinds, we need a way to distinguish among them. As Rerun also does not end its episodes upon past-races, we borrow Rerun's method of distinguishing between past- and current-races.

3.1.1 Post-dating: Overview

Rerun deems a race via a block to be a past-race or current-race depending on whether the *predecessor thread* (that accessed the block first) accessed the block, respectively, in a *previous* episode or in the *current* episode. In the former case of a past-race, because the predecessor thread's previous episode has *already* ended, the successor thread's episode can ensure proper ordering by advancing its timestamp beyond the predecessor thread's *current* timestamp. Because the predecessor thread's current timestamp is guaranteed to be ordered after *its own* previous episode, transitivity guarantees that the successor thread's episode is ordered after the predecessor thread's previous episode. Not ending the successor threads' episodes in such cases enables Rerun to achieve long episodes, and hence small logs. We take the same approach for past-races with one small modification: The successor advances its timestamp beyond the predecessor's *previous* chapter's timestamp instead of the predecessor's current timestamp. Because the above transitivity guarantee still holds, the correct ordering is captured. To this end, Timetraveler keeps two timestamps per thread, a *current timestamp* for the current chapter's timestamp and a *past timestamp* for the last completed chapter's timestamp.

In the latter case of current-races, Rerun ends the predecessor episode (the first limitation discussed in Section 1) so that the predecessor's timestamp does not advance beyond the successor's timestamp, ensuring proper predecessor-successor ordering. Not ending the predecessor would allow the predecessor's timestamp to advance due to other memory races which may violate the ordering. Timetraveler does not end a chapter on a current-race and addresses this problem via *post-dating* based on the key observation that chapters may include multiple current-races as long as (1) the predecessor and the successor chapters involved in every such race can *eventually* be ordered properly, and (2) the predecessor-successor ordering imposed by the current-races on the currently-live chapters is not cyclic.

For the first constraint, the predecessor chapter provides a *post-dated* timestamp to the successor chapter with the guarantee that in the future the predecessor chapter's timestamp will not advance beyond the post-dated timestamp. The post-dated timestamp is a fixed offset, called the *post-dating offset*, from the chapter's current timestamp (there is a corner case where the offsetting is different for optimization purposes, as we explain in Section 3.1.3). The successor chapter advances its own timestamp beyond the predecessor's post-dated timestamp, guaranteeing that it will always be ordered after the predecessor chapter. Cyclic ordering would require the chapter to advance beyond its own post-dated timestamp. Post-dating easily detects this condition and forces the offending chapter to end. All threads which do not have any post-dated timestamp (threads with no memory race and successor-only threads) do not have any constraints on their timestamps and they can continue execution without ending their chapters. In case a memory race requires a successor to increase its timestamp beyond the post-dated timestamp, the successor will end its current chapter, log it, clear the post-dating and start a new chapter. Because cyclic current-races are rarer than acyclic current-races, Timetraveler achieves longer chapters than Rerun, resulting in reduced log size.

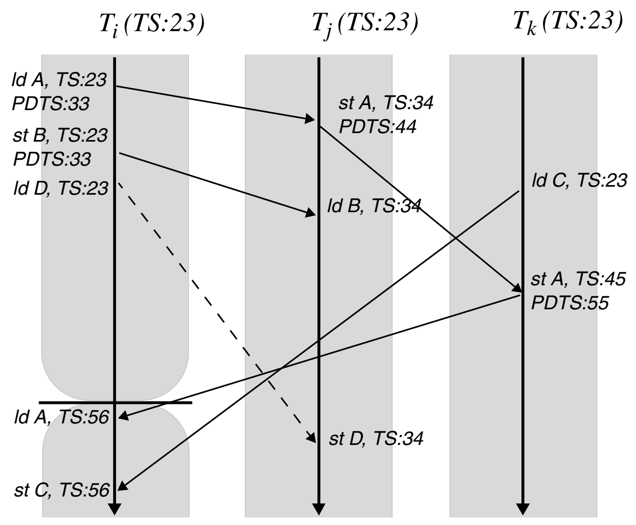

Figure 2. Timetraveler's chapters

To distinguish between past- and current-races, Rerun tracks whether a block has been accessed by a currently-live predecessor episode (i.e., a *current-block)*, or by a previously-ended predecessor episode (i.e., a *past-block*). This binary information requires significantly less hardware (e.g., bit per block or Bloom filters), compared to the exact instruction count of the access in the L1 as done in [30,31]. Rerun employs a pair of Bloom filters per thread to track the current-blocks as the set of blocks read and written by the thread's current episode. While races always involve a write, tracking reads and writes together in one set would misidentify multiple episodes reading the same block as being involved in a race. Therefore, reads and writes are tracked separately. Instead of Bloom filters, Timetraveler uses read/write bits in the cache tags (Section 3.1.4).

3.1.2 Post-dating: Example

To illustrate post-dating, Figure 2 shows an example of Timetraveler chapters for three threads T_i, T_j and T_k, each with the initial timestamp of *TS* = 23. Time progresses downward; the arrows represent races and the ordering between the racing threads. At the time of each racing reference, we show the local timestamp. We omit the memory references that do not cause races (e.g., L1 cache hits). Each grey box represents a chapter and a horizontal line marks the end of a chapter. For the current-race between T_i and T_j via current-block *A*, T_i provides a post-dated timestamp (*PDTS*) of 33 to T_j (the *post-dating offset* is 10). As a result, T_j advances its timestamp (*TS*) to 34 and provides a *PDTS* of 44 to T_k for the current-race via current-block *A*. Accordingly, T_k advances its *TS* to 45. Another current-race between T_i and T_j via current-block *B* causes T_i to provide its already-established *PDTS* of 33 to T_j. However, because T_j's *TS* is already 34 (i.e., T_i and T_j are ordered correctly already), this race does not change T_j's *TS*. For the current-

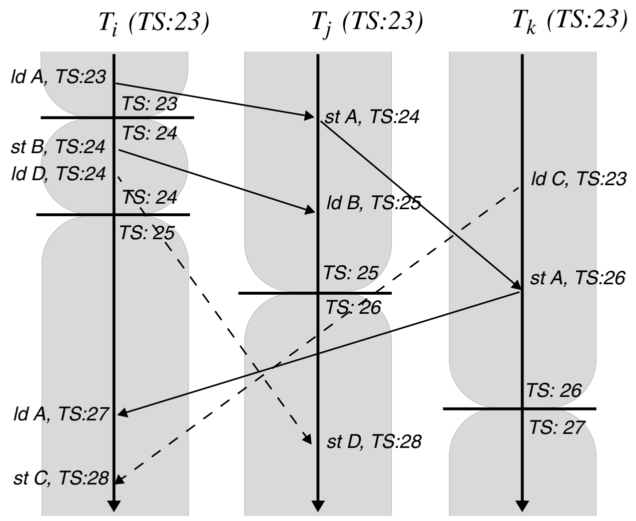

Figure 3. Rerun's episodes

race between T_i and T_k via current-block *A*, T_k provides a PDTS of 55 to T_i. Advancing T_i's *TS* to 56 would cause T_i to exceed its PDTS of 33, indicating a cyclic race. Consequently, T_i ends its chapter before the offending instruction, and starts a new chapter with a *TS* of 56. For the current-race between T_i and T_k via block *C*, T_k provides a PDTS of 55 to T_i whose *TS* is already 56, and hence results in no change. Finally, the race between T_i and T_j via block D is a past-race because the thread T_i has ended the chapter in which it last accessed the block. The thread T_j does not change its timestamp because the thread T_i's past timestamp is lower than the thread T_j' current timestamp. At the time of replay, the chapters are ordered as follows: T_i's first chapter (*TS* = 23),T_j's chapter (34 <= *TS* <= 44), T_k's chapter (45 <= *TS* <= 55), and T_i's second chapter (*TS* >= 56). There is a corner case where the predecessor's PDTS is different than the value determined by the post-dating offset, as we explain in Section 3.1.3.

In the case of Rerun, this example shows two past-races and four current-races, resulting in four episodes as shown in Figure 3. T_i's first episode ends at *TS* = 23 because it is the predecessor in the current-race with T_j via block *A*. T_i's second episode starts at *TS* = 24 and ends at *TS* = 24 because it is the predecessor in the current-race with T_j via block *B*. For each of these races, the figure shows how T_j's *TS* advances from 23 to 24 and then to 25. The dotted arrows show past-races, e.g., block *C* is a past-block for T_k at the time of T_i's access because T_k accesses *C* in its first episode which ends before T_i's access. Being a past-race, this race does not cause T_k's second episode to end and simply causes T_i's *TS* to advance to 28. We see that there are a total of four current-races which result in four episodes. At the time of replay. the episodes would be ordered as follows: T_i first episode (*TS* = 23), T_i second episode (*TS* = 24), T_j first episode (*TS* = 25), and T_k first episode (*TS* = 26).

3.1.3 Post-dating: Operation

To track the memory blocks accessed in a current chapter, a thread increments its REFS counter for the current chapter and sets the read/write bits for the accessed blocks upon commit of a memory operation (see Figure 4, lines 1-3). In the absence of cache misses, a thread executes without ending its current chapter and its timestamp remains unchanged. In the case of a miss, races are detected

```
[1]  On commit of a mem op
[2]  REFS++
[3]  set read/write bit based on load/store

[4]  On sending request for block A
[5]  REQUEST (A, PDTS)

[6]  At predecessor (On data/ack send for block A)
[7]  If the access is a memory race
[8]    If A is a past-block
[9]      SEND (A, PTS)
[10]   else                // current-block
[11]     if rec_PDTS is unset,
[12]         if my_PDTS unset then my_PDTS = CTS + D_PDTS
[13]     else if $CTS \geq rec\_PDTS$ // successor ends, cycle detected
[14]         if my_PDTS unset then my_PDTS = CTS + D_PDTS
[15]     else // successor also has PDTS, try half-way
[16]         my_PDTS = min (my_PDTS, (CTS + (rec_PDTS - CTS) /2))
[17]     SEND (A, my_PDTS)
[18] If the access is not a memory race
[19]   If A is a past-block
[20]     SEND (A, PTS)
[21]   else
[22]     SEND (A, CTS)

[23] At successor (On data receive, RECEIVE (A, rec_TS))
[24]  If $rec\_TS \geq my\_PDTS$ // limit reached, end chapter
[25]          LOG (REFS, CTS)
[26]          clear read/write bits and REFS
[27]          clear PDTS
[28]          PTS = CTS
[29]          CTS = rec_TS + 1
[30] else // increment my timestamp without ending chapter
[31]          CTS = max (rec_TS + 1, CTS )
```

Abbreviations:	
PDTS = Post-dated timestamp	*REFS = Memory references count*
CTS = Current timestamp	*rec_PDTS = Received timestamp*
PTS = Past timestamp	*rec_TS = Received timestamp*
D_PDTS = Post-dating offset	

Figure 4. Post-dating algorithm

through coherence actions as done in previous schemes — e.g., a successor thread's read miss satisfied by a remote predecessor's dirty block. The successor thread sends its post-dated timestamp along with the coherence request message for a block (see Figure 4, lines 4-5). The predecessor thread(s) check the read/write bits in the L1 to determine whether a race exists and if so, whether it is a past- or a current-race (i.e., the requested block is a past-block or a current-block for the predecessor respectively). We discuss the cases of a past-race and a current-race first and then the case of the absence of a race.

In the case of a past-race (via a past-block), the predecessor threads send their last-completed chapters' timestamps piggy-backed on the coherence reply payload (data or acknowledgement, as appropriate) to the successor thread (see Figure 4, lines 8-9).

This action is similar to Rerun's action (Section 3.1.1). The successor thread advances its timestamp one more than the largest of the predecessors' past timestamps. Neither the predecessor nor the successor threads end their chapters.

In the case of a current-race (via a current-block), Timetraveler's actions are different from Rerun's (Section 3.1.1). There are three cases based on the successor's PDTS, which is sent in the coherence request payload. (i) If the successor's PDTS is not set, then the predecessor sends its PDTS (if the predecessor's PDTS was not set, then it is set to be the current timestamp added to the post-dating offset) (see Figure 4, lines 11-12). (ii) If the predecessor's current timestamp already exceeds the successor's PDTS, the predecessor sends its PDTS and the successor ends its chapter, clears its read/write bits, REFS, and PDTS, and starts a new chapter which is ordered correctly after the predecessor's PDTS (see Figure 4, lines 13-14, 24-29). (iii) If the successor's PDTS is set, the successor's *TS* is already limited by its *PDTS*; therefore, if the predecessor blindly sends its PDTS to the successor, then this computed value may exceed the successor's PDTS causing the successor's chapter to end. Instead, the predecessor *tries* to change its own PDTS to half-way between the successor's PDTS and the predecessor's current timestamp, so that the successor has some room and its chapter need not end (see Figure 5). If this half-way point exceeds the predecessor's current PDTS, the predecessor's PDTS does not change (because a predecessor cannot increment its PDTS once assigned) and the successor chapter ends like the second case. If the half-way point does not exceed the predecessor's current PDTS then the predecessor's PDTS would retreat in time so that the predecessor's other successors are still ordered correctly. Therefore this change occurs only if the halfway point does not exceed the predecessor's current PDTS (see Figure 4, lines 15-16). The half-way computation is approximate and done by simply dropping the least-order bit instead of expensive, accurate dividing. This third case is the corner case mentioned in Section 3.1.2.

For both past- and current-races, the successor always advances its current timestamp by incrementing beyond the received timestamp if the received timestamp is greater than its timestamp (see Figure 4, lines 29,31). If there are multiple predecessors, then all the predecessors take the above actions and the successor advances beyond the largest received timestamp.

So far, we have covered the cases when a predecessor detects a past- or a current-race. Now we discuss the case of a predecessor detecting the absence of a race (i.e., a read miss encounters a clean, shared block). The read miss must still be ordered after the last write to the block. The issue is that the block may have changed coherence state after the last write due to either reads from other cores (e.g., Modified to Shared downgrade) or evictions (we cover L1 evictions later in Section 3.1.4). As such, the state changes hide the fact that a write has occurred. However, ordering the read miss after the entity that has the block (another thread or L2) would ensure that the read is ordered after the write (that entity would have been ordered after the write). Accordingly, the reader advances its own timestamp to one more than the received timestamp. If the sending entity is another thread, then depending upon whether the block is past or current with respect to the thread's current chapter, the thread sends its last-completed chapter's timestamp or the current timestamp (see Figure 4, lines 18-22).

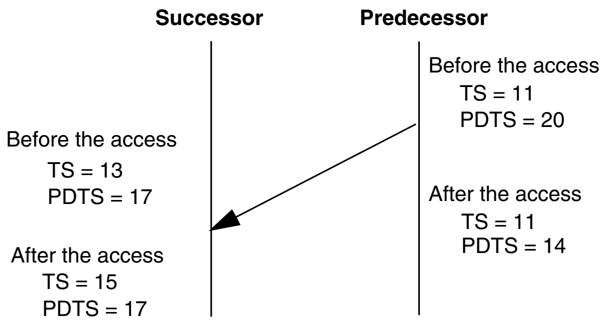

Figure 5. Half-way example

Following Rerun, we send the timestamps as part of coherence message payloads, incurring minimal latency and bandwidth overheads (we report our overhead in Section 5.1). In addition to the obvious impact of the overheads on performance and hence adoption of the scheme, support for deterministic replay should not perturb execution such that turning on the support introduces nondeterminism in the observability of bugs. Because the higher the overheads the higher the perturbance, achieving minimal latency/ bandwidth overhead is important. We note that the additional harwdare needed for post-dating is minimal: one post-date timestamp register per core.

3.1.4 L1 evictions and L2

The remaining issues with post-dating are L1 evictions and modified-to-shared downgrades. If a block accessed by a currently-live chapter (i.e., a *current-block*, as defined in Section 3.1.1) is evicted or downgraded from the L1 then a current-race via the block may go undetected in the shared L2. To avoid this problem, Rerun conservatively ends the episode upon a current-block eviction or downgrade (the second limitation discussed in Section 1), essentially converting potential future current-races into past-races by forcing all evicted or downgraded blocks to be past-blocks. As in the case of current-races via L1-resident blocks, not ending the predecessor would allow the predecessor's timestamp to advance which may violate the predecessor-successor ordering.

Even though the episode is ended upon current-block evictions and downgrades, the ended episode's timestamp is needed to advance the successor's timestamp and properly order the successor episode. We first explain how Rerun addresses this problem before explaining how Timetraveler addresses L1 evictions and downgrades. In Rerun, L1 evictions and downgrades send their timestamps to L2 which maintains a coarse-grain, per-bank timestamp to avoid the overhead of per-block timestamps (because races through the L2 are less frequent than those through the L1). The per-bank timestamp is updated whenever the L1 timestamp is larger than the per-bank timestamp. Silent replacements (i.e., replacements of a clean, shared copy), a key optimization of directory coherence, also end the episodes like non-silent replacements on current-block evictions but do not generate any traffic to the directory and hence do not update the L2 timestamp. However, the directory forwards requests to the L1 cache for silently-replaced blocks due to lack of knowledge of the replacements. The L1 replies to the requests with the current timestamp, so that the successor is guaranteed to be ordered after the current timestamp, and hence by transitivity, after the episode-ending silent L1 eviction, which occurred in the past.

Because ending the chapters upon L1 evictions and downgrades would shorten the chapters, Timetraveler instead employs post-dating to address this limitation. Upon evicting or downgrading a current-block, a chapter sends its post-dated timestamp to the L2 bank, without ending the chapter. The post-dated timestamp guarantees that any successor accessing the block in the L2 is ordered after the predecessor (evicting thread). For past-block evictions or downgrades, a chapter sends its past timestamp to the L2 bank. While we employ post-dating for non-silent replacements, we perform an additional optimization for silent replacements. Blindly post-dating upon silent replacements of current-blocks would unnecessarily shorten chapters, which cannot advance past the post-dated timestamp in the common case where there are no future requests for the silently-replaced current-block. Instead, we employ a single bit per cache set indicating that a current-block in the set has been replaced silently. Later coherence requests to such silently-replaced blocks are guaranteed to be forwarded to the L1 by the directory. Such requests would miss in the cache but would map to a set with the bit turned on. Accordingly, we post-date the chapter under the conservative assumption that a current-race is about to occur via the replaced block. This bit per set is similar to FDR-2's per-set timestamp for replaced blocks.

Finally, Rerun's Bloom filters decouple the timestamp state from the cache so that the cache is not complicated by the timestamp state, and the filters and the cache can be sized independently. While such decoupling was proposed first in FDR-2 which has full-blown timestamps per block, Rerun has only two bits per block which is not so large as to complicate the cache. Further, though 512-bit and 1024-bit filters suffice so that the state held by the filters is small (64-128 bytes) [12], each bit in the 512-bit filter needs to be addressed requiring a 9-to-512 decoder which incurs a large area overhead [24]. Reducing the area overhead via smaller filters causes false positives which force Rerun's episodes to end. Consequently, we advocate using read/write bits in the cache tags like many speculative memory schemes [9,10,26] and hardware transactional memory schemes [11,17]. Unlike many of these schemes, Timetraveler's bits do not interact with or change coherence actions in any way (i.e., no nacks or wait states which may cause deadlocks or livelocks), and hence do not add any complexity to coherence. Though one may think that updating the read bits on a read access requires an extra write upon a read access, reads already update replacement information (e.g., LRU) which could be extended to update the Timetraveler read bits.

3.2 Time-delay Buffer

In Rerun, the L2 bank inherits the evicted or downgraded blocks' normal or post-dated timestamps based on whether the block is past or current. However, eviction or downgrade of a single recently-accessed block forces the *entire* L2 bank to advance to the block's (normal or post-dated) timestamp, even if all the other blocks in the bank were accessed at much older timestamps (the third limitation discussed in Section 1). If Timetraveler were to employ this inheritance, a current chapter that hits in an L2 bank would advance its timestamp beyond the bank's timestamp and hence ensure correct ordering with respect to the chapters that last accessed the block. If the bank's timestamp were to exceed the chapter's post-dated timestamp upon an L2 hit, the requester would have to end its chapter even if the access is made to an unrelated L2 block — that is, even if it is a false cycle. To address this problem, we make the key observation that if L1 evictions are delayed from advancing the L2 bank's timestamp, the current chapters' post-dated timestamps would not be exceeded and false cycles would be prevented. To this end, we propose a per-L2-bank *time-delay buffer*, Timetraveler's second novel mechanism, which holds the timestamps of a few recent evictions (Figure 1).

If an L1 writeback's timestamp is greater than the L2 bank's timestamp, the L1 writeback checks the buffer for address matches. In the absence of a match, the writeback address and timestamp are placed in the buffer — the writeback data is sent to the L2 and not held in the buffer; upon a match, the matching entry's timestamp is updated if the writeback timestamp is larger. L1 misses also check the buffer for address matches and upon a match, the matching entry's timestamp is returned. In the case of an exclusive access, the entry is removed from the buffer. This removal occurs because a matching L1 miss implies that the block has now moved into an L1 with a more recent timestamp than that in the entry, so that races via the block would be caught by coherence and hence the entry in the buffer is obsolete.

The buffer is organized as a FIFO which empties its entries into the L2 bank. Each dequeued entry from the buffer updates the L2 bank's timestamp if the entry's timestamp is larger. Emptying the buffer as per timestamp order instead of arrival order does not reduce the impact of the L2 bank's timestamp on chapter ending enough to justify complex timestamp-ordering circuitry.

We optimize the time-delay buffer to avoid false self-cycles. Blocks that are written back by a chapter may be accessed later by the same chapter and the chapter may have to end because of its own post-dating. To avoid this problem, we include the core identifier in the buffer entry so that the entry's timestamp is ignored upon a core identifier match (i.e., the chapter's timestamp is not changed when a block is re-accessed). One complication that arises is that when the block is moved back into the L1 from the buffer after a previous L1 eviction within the same chapter, the block loses the read/write bits from before the eviction. If the block was written prior to the eviction and read after the eviction, then the block read/write bits would reflect only the read and not the write. Thus, Timetraveler may miss some conflicts via such blocks. Thus we keep read/write bits in the buffer as well to propagate the block state from the delay-buffer to L1. If the block is dequeued from the buffer and sent to the L2 before being re-accessed, the re-access would result in a self-race causing the chapter to end (this case may shorten chapters but occurs only when the buffer capacity is exceeded). We observe that this case is infrequent.

We summarize the hardware requirements for Rerun and Timetraveler in Table 1. While Timetraveler's time-delay buffer is extra, the buffer holds only addresses and timestamps, but not data. We find that a modest-sized buffer suffices (e.g., 8 entries per bank).

For OS events such as context switches and thread migrations, we end chapters, like Rerun. There is one minor interaction between thread migration and the core identifiers in the time-delay buffer entries. The buffer may incorrectly consider a newly-migrated thread on a core, with the same core identifier as the previous thread on the core, to be the previous evictor of a block, assume a self-cycle, and ignore the block's timestamp. Such ignoring may cause Timetraveler to miss some current-races between the previous and newly-migrated threads. To address this problem, a newly-

Table 1. Hardware requirement

Hardware	Rerun	Timetraveler
Per Core	-Episode length counter -One timestamp register -Read and write Bloom filters	-Chapter length counter -Three timestamp registers for current, past, and post-dated timestamps -One read and one write bit per L1 cache block and one bit per L1 set
Per L2 Bank	-One timestamp register	-One timestamp register -Time-delay buffer where each entry holds a block address, R/W bits, core id, and a timestamp.

migrated thread's timestamp starts by advancing one more than the larger of (1) the thread's starting timestamp for its next chapter and (2) the previous thread's post-dated timestamp. This advancing ensures that the previous thread's blocks held in the time-delay buffer are in the past of the newly-migrated thread and hence cannot cause current-races.

3.3 Discussion

While most often chapters end due to cyclic, current-races, Timetraveler may prematurely end a chapter due to the following false reasons: (1) an acyclic race where the successor's post-dated timestamp happens, by chance, to be smaller than the predecessor's timestamp (Section 3.1.3); (2) exceeding the post-dated timestamp due to past-races (Section 3.1.3), (3) exhaustion of the post-dated timestamp due to half-way division (Section 3.1.3), (4) unnecessary post-dating due to L1-evicted current-blocks that are not accessed by any other thread; (5) false cycles through the L2 upon exceeding the Time-delay buffer's capacity (Section 3.2); and (6) exhaustion of the memory reference counter (i.e., REFS in Section 3). We found that in practice these cases are uncommon.

Finally, because post-dating advances the timestamps by the post-dating offset, Timetraveler's timestamps may advance much faster than Rerun's timestamps and may exhaust the timestamp space sooner (e.g., 32 bits). Upon such exhaustion, all the threads' chapters end, and the time-delay buffer and L2 banks' timestamps are reset. However, because a thread's successive chapters' timestamps differ only by small amounts (e.g., 256), we can employ well-known log compression [23] to record only the differences between successive timestamps instead of the entire timestamp. This compression scheme allows a larger timestamp space without increasing the log size; only the timestamp registers in the core and the time-delay buffer need to be wider.

4. METHODOLOGY

We simulate Timetraveler using Wisconsin GEMS-2.1 [15] built on top of Simics, a full-system simulator[14]. We simulate a SPARC-based multicore system running Solaris 10. The hardware parameters are given in Table 2. We use three commercial workloads and three scientific applications from Splash-2 suite [29], as described in Table 3. To account for statistical variations, we use enough randomly-perturbed runs to achieve 95% confidence [5]. We compare Timetraveler with Rerun, the best previous distributed

Table 2. Base system configuration

Cores	8, in-order
L1 Caches	Split I&D, Private, 32K 4-way set associative, write-back, 64B cache block, LRU replacement, 3 cycle hit
L2 Cache	Unified, Shared, Inclusive, 8M 8-way set associative, write-back, 8 banks, LRU replacement, 37 cycle hit
Directory	Full bit vector in the L2
Memory	8 GB, 300 cycles
Coherence	MESI Directory, silent replacements

scheme. We simulate Rerun using Bloom filters of the same size as in [12] — i.e., 256 bits for the write filter and 1024 bits for the read filter. Timetraveler uses two read/write bits per L1 cache block, one bit per L1 set for silent replacements (Section 3.1.4), post-dating offset of 10, and a 696-byte, 8-entry time-delay buffer per L2 bank, where each entry holds an address, a timestamp, core identifier, and R/W bits. For both Timetraveler and Rerun, we assume 32-bit timestamps and a 16-bit memory reference counter. While log compression via timestamp differences is possible (as mentioned in Section 3.3), we assume uncompressed logs for both Timetraveler and Rerun.

5. EXPERIMENTAL RESULTS

We start by comparing the log sizes of Timetraveler against those of Rerun, followed by a dissection of the chapters and an analysis of the various reasons due to which the chapters end (either due to true or false cycles, as discussed in Section 3.3). Next, we isolate

Table 3. Benchmarks

Commercial (com)
Apache is a web server. We use Apache 2.2.11 [27] and SURGE v1.3 [7] with http 1.1 capability to generate web requests from a repository of 20,000 files (~500MB). We simulate 3200 clients, each with 25ms think time between requests, and warm up for ~1,500,000 transactions before measuring 600 transactions. **Online Transaction Processing (OLTP)** models the database for a wholesale supplier, with many users performing concurrent transactions. We use PostgreSQL v8.3.7 [2] database server and Open Source Development Labs Test Suite DBT-2 v0.40[1] for modeling users based on TPC-C specifications. We use a 5 GB database with 25,000 warehouses. We simulate 128 users with 0 think time, and warm up the database for ~100,000 transactions before taking measurements for 200 transactions. **SPECjbb2005** is a java-based server workload v1.07[3] for online transaction processing in middle ware. We use Sun J2SE v1.5.0 JVM. We simulate 1.5 warehouses per processor with 0 think time, warm up for 350,000 transactions and take measurement for 10,000 transactions.
Scientific (sci)
SPLASH-2: We use *ocean*, *raytrace*, and *water-nsquared* from SPLASH-2[29]. We run these applications to completion using default parameters. For raytrace, we render the "car" object as provided by the SPLASH-2 suite.

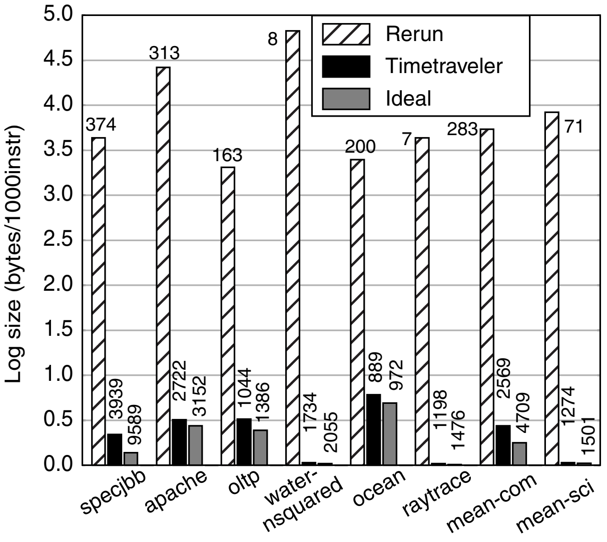

Figure 6. Log size

the contributions of post-dating and time-delay buffer. Then, we present Timetraveler's sensitivity to its various parameters (e.g., time-delay buffer size). We conclude this section by showing the scalability of Timetraveler with the number of cores.

5.1 Timetraveler's Log Size

We compare Timetraveler with Rerun in terms of the log size. The *Ideal* bar represents true cyclic, current-races which we identify using a race table by recording in a graph the races among the threads' live chapters via all (L1-resident and L1-evicted) blocks, and detecting true cycles using a depth-first-search of the graph without using post-dating. Figure 6 shows the log size of the schemes in bytes per thousand instructions executed (Y-axis, lower is better) for each benchmark (X-axis). The two rightmost clusters show the mean for the commercial benchmarks (*mean-com*) and scientific benchmarks (*mean-sci*). Because the commercial and scientific benchmarks behave quite differently, we separate their means. The numbers on top of the bars are the average lengths in number of memory references of Timetraveler's chapters and Rerun's episodes. Rerun's log sizes for the commercial benchmarks as obtained by us are in line with the original paper (which does not show scientific benchmarks).

Timetraveler achieves lower log growth rate (i.e., smaller log) than Rerun for all benchmarks providing 91%, 89%, and 84% reductions for *specjbb*, *apache*, and *oltp*, respectively, and 99%, 77%, and 99% reductions for *water-nsquared*, *ocean*, and *raytrace*, respectively, with mean reductions of 88% and 99% for the commercial and scientific benchmarks, respectively. Timetraveler's log size on average is 0.44 and 0.03 bytes per 1000 instructions for the commercial and scientific benchmarks, respectively, which are close to Ideal's log size of 0.25 and 0.02 bytes per 1000 instructions. Because the scientific benchmarks have much less data sharing and synchronization, and hence significantly fewer cyclic races than the commercial benchmarks, Timetraveler achieves greater log size reduction over Rerun in the scientific benchmarks than the commercial benchmarks.

To put these reductions in perspective, for an 8-core multicore using aggressive 2-GHz, 1-cycle-per-instruction cores, Timetraveler's log grows as 950 KB/s and 59 KB/s for the commercial and scientific benchmarks, respectively, as compared to Rerun's rate of 8 MB/s and 7.3MB/s. Timetraveler's mean chapter lengths are 2569 and 1274 memory references for the commercial and scientific benchmarks, respectively, as compared to Rerun's episode length of 283 and 71 memory references, and Ideal's chapter lengths of 4709 and 1501 memory references. Due to rampant replacements which force Rerun to end its episodes, *water-nsquared* and *raytrace* have unusually short episodes in Rerun. The log reductions show the benefit of Timetraveler's approach of exploiting both past- and acyclic, current-races over Rerun's approach of exploiting only past-races. This improvement requires Timetraveler's time-delay buffer which is extra hardware but the overhead is only 696 bytes for an 8-core CMP. The overhead due to piggybacking timestamps on coherence messages and writing the log increases Timetraveler's total chip bandwidth by nearly 11% which is close to Rerun's 10% increase.

We now turn to DeLorean [16], the latest centralized scheme. In addition to being more scalable and using less hardware, Timetraveler achieves a log size of 0.05 bits per core/1000 instructions compared to DeLorean's 0.3 bits per core/1000 instruction for PicoLog without compression, as reported in [16]. We present this comparison with the caveat that the numbers are from different systems using different hardware parameters and benchmarks.

5.2 Chapter Analysis

In this subsection, we first present a dissection of the chapters and later we analyze the reasons contributing to the ending of a chapter

5.2.1 Chapter Dissection

Table 4 shows the events occurring inside a chapter. *Current-races* (A) are the total number of current-races per chapter. *Current-block replacements* are divided into two categories; those that would have resulted in current-races had they not been replaced and had the chapter not ended (B), and those that do not lead to races (C). Column (D) shows the total number of current-races within a chapter which is the sum of (A) and (B). Columns (A), (B), and (C) represent the cases where Rerun ends its episodes. However, the numbers in the table do not directly relate to Timetraveler's improvements over Rerun because Timetraveler's chapters are longer than Rerun's episodes, and hence may have more current-block replacements compared to Rerun's episodes. Also, some of the past-races in Rerun may turn into current-races for Timetraveler, and vice versa. From the table, we see that a chapter covers 6.8 and 6.4 current-races, on average, for the commercial and scientific benchmarks, respectively. *Timestamp sent from buffer* (E) shows the number of times the time-delay buffer, and not the L2, sends the timestamp for an L1 miss. This count shows the number of times a chapter may avoid potential false cycles through the L2.

5.2.2 Reasons for Ending Chapters

We analyze the reasons — false ones listed below and the true one of cyclic, current-races — for Timetraveler to end a chapter and show that the majority of chapters end on true cycles. In Figure 7, the Y-axis represents the fraction of chapters ended due to each of the reasons (normalized to 100%), and the X-axis shows the benchmarks. *True cycles* represent cyclic, current-races as mentioned in Section 5.1. We list the false reasons as follows

Table 4. Chapter dissection

Benchmarks	Current-races (A)	Current-block replacements		Total current-races/ chapter (D)	Timestamp sent from buffer (E)
		Current-races (B)	Non-races (C)		
specjbb	0.6	1.1	21.0	1.7	0.79
apache	1.5	8.0	26.1	9.5	3.02
oltp	3.4	5.8	12.2	9.3	1.95
water-nsquared	2.3	6.4	228.2	8.7	6.41
ocean	1.8	2.4	5.1	4.1	1.47
raytrace	2.4	3.9	197.8	6.3	3.26
mean-com	1.8	4.9	19.8	6.8	1.92
mean-sci	2.1	4.2	143.7	6.4	3.71

(Section 3.3). *False cycles through L1* represents the chapters ending due to acyclic races via L1-resident blocks where the successor's post-dated timestamp happens, by chance, to be greater than the predecessor's timestamp. *False cycles through L2* represents the chapters ending due to one of two cases: (1) L1 misses which also miss in the time-delay buffer and the L2 bank's timestamp is greater than the requesting thread's post-dated timestamp though the L2 block's real timestamp would be smaller than the post-dated timestamp. (2) unnecessary post-dating for L1-evicted current-blocks that are not accessed by any other thread. *Past-races* represent the chapters ending on past-races when a successor exceeds its post-dated timestamp. Other false reasons discussed in Section 3.3 — half-way division and memory reference counter exhaustion — contribute less than 1% and hence are not shown.

From the graph, we see that the false reasons contribute to nearly 28% of chapter endings most of which are false cycles through the L1 and L2. While false cycles through the L1 occur by chance and may be hard to remove, false cycles through the L2 can be reduced by increasing the time-delay buffer's size (see Section 5.4). Because *specjbb* exerts higher capacity pressure on the L1 than the other workloads, *specjbb* has many more false cycles through the L2. As we show in Section 5.4, *specjbb*'s false cycles can be reduced by increasing the time-delay buffer size.

5.3 Impact of Timetraveler's Mechanisms

We isolate the contributions of Timetraveler's mechanisms — bit-per-set optimization for silent replacements (Section 3.1.4), time-delay buffer and post-dating — by showing three variants of Timetraveler. The first variant, called *Timetraveler-no-bit-per-set* is Timetraveler without the per-cache-set bits. The second variant, called *Timetraveler-no-tdb,* is Timetraveler without the time-delay buffer, where all replacements directly update the L2 bank's timestamp. We also present Rerun, which is equivalent to Timetraveler excluding the bit-per-set optimization, time delay buffer, and post-dating**.**

Figure 8 shows the average chapter length in terms of total number of memory references on the Y-axis for each variant (higher is better). The mean chapter length of Timetraveler at 1921 is significantly better than that of *Timetraveler-no-bit-per-set* at 1698, and *Timetraveler-no-tdb* at 1488, showing the effectiveness of employing, respectively, bit-per-set optimization, and the time-delay buffer which delays advancement of the L2 banks' timestamps. *Timetraveler-no-tdb*'s mean chapter length is 1488 as compared to Rerun's 238. This difference highlights the impact of post-dating for current-races and current-block replacements. Overall, we see that all of Timetraveler's mechanisms have significant impact on the log size.

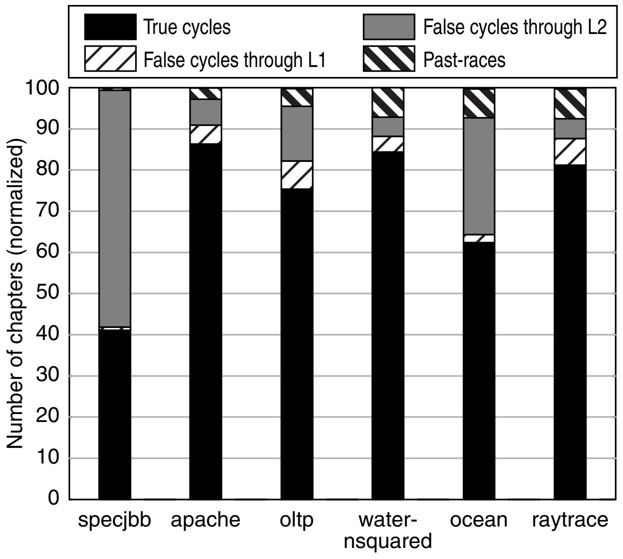

Figure 7. Reasons for ending chapters

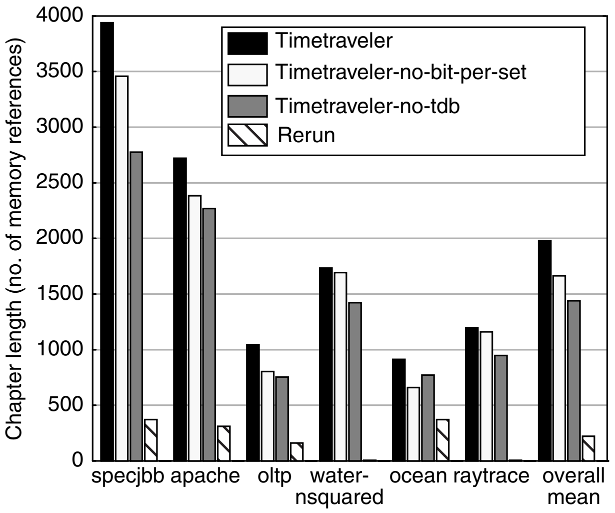

Figure 8. Impact of Timetraveler's mechanisms

5.4 Sensitivity to Post-dating Offset and Time-delay Buffer Size

Figure 9 shows Timetraveler's sensitivity to two of its design parameters — post-dating offset and time-delay buffer size. The Y-axis shows chapter length in number of memory references. *Tt-default* represents Timetraveler with the default parameters (post-dating offset of 10 and per-bank time-delay buffer of 8 entries). For all the other bars in the figure, one specific parameter is changed

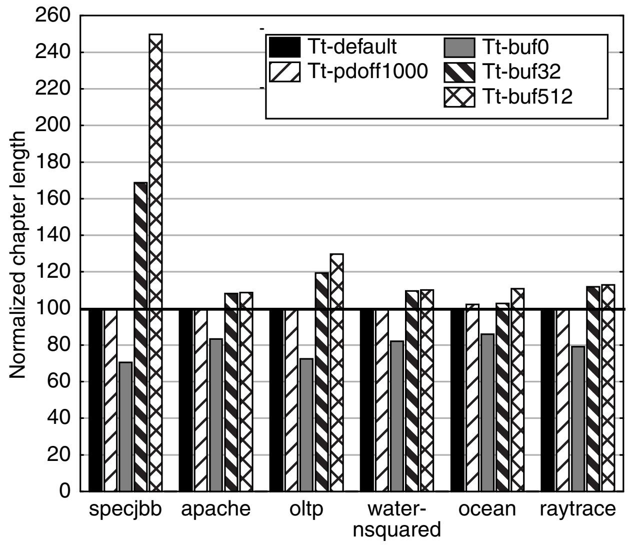

Figure 9. Timetraveler sensitivity

and the rest are kept the same as that of *Tt-default*. *Tt-pdoff1000* represents Timetraveler with a post-dating offset of 1000. We observe that the chapter length does not vary much as we increase the post-dating offset from 10 (*Tt-default)* to 1000. This lack of variation is because most of the chapters do not exceed the post-dated timestamp before encountering a current cycle which causes the chapters to end naturally (i.e., cases such as the repeated half-way division, which increase the chances of exceeding the post-dated timestamps, are rare)**.** We also observed that decreasing the post-dating offset below 10 (default) deteriorates the chapter length (not shown). Though post-dating offsets of 10 and 1000 produce similar results, we choose 10 as our default because a smaller offset causes timestamps to advance less rapidly and the timestamp space to be exhausted less often.

The next set of bars vary the buffer size. *Tt-buf0* does not have a time-delay buffer and is the same as *Timetraveler-no-tdb* (Section 5.3). *Tt-buf32* and *Tt-buf512* use 32- and 512-entry time-delay buffers per L2 bank, respectively. The graph shows that the chapter length increases as we increase the buffer size. *specjbb,* with its higher capacity pressure, is more sensitive to buffer size as compared to the other benchmarks. This is because *specjbb* has a lot more capacity misses than the other two workloads which require larger buffer size to accommodate the misses. Consequently, the time-delay buffer's fine-grained timestamps have a higher impact on *specjbb* than on the others.

5.5 Scaling the Number of Cores

In Figure 10, we show the scaling trend of Timetraveler's log sizes as we increase the number of cores. The Y-axis shows the log size in bytes per thousand instructions whereas the X-axis represents our workloads. From the graph, we see that the log growth rate scales well as we add more cores though the trend across the benchmarks is not consistent.

6. CONCLUSION

Our goals were to achieve smaller logs for recording memory races than currently achieved by exploiting transitivity of races and to require minimal hardware to do so. To these ends, we proposed *Timetraveler* which for the first time exploits acyclicity of races based on the key observation that an acyclic race need not be recorded even if the race is not covered already by transitivity. Timetraveler employs a novel and elelgant mechanism called *post-dating* which both ensures that acyclic races, including those through the L2, are eventually ordered correctly, and identifies cyclic races. To address the problem of false cycles through the L2, Timetraveler employs another novel mechanism called *time-delay buffers* which delay the advancement of the L2 banks' timestamps and thereby reduce the false cycles. Using simulations, we showed that Timetraveler reduces the log size by 88% and 99% over Rerun for commercial and scientific benchmarks, respectively, while using only a 696-byte time-delay buffer.

The elegance of post-dating and time-delay buffers ensures that Timetraveler requires minimal additional hardware over a conventional chip multiprocessor (CMP) despite exploiting the completely new property of acyclicity of races. By achieving significant reduction in log size while requiring minimal hardware, Timetraveler lowers the barrier for adoption of hardware support for memory-race recording in commercial CMPs. While we assumed sequential consistency in this paper, we will extend Timetraveler to other consistency models in our future work.

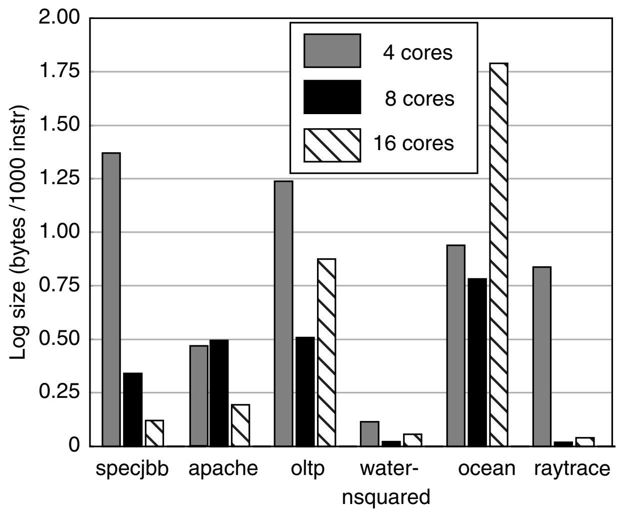

Figure 10. Scaling with number of cores

REFERENCES

[1] Open Source Development Labs Database Test Suite 2 v0.40 http://osdldbt.sourceforge.net/.

[2] PostgreSQL. v8.3.7. http://www.postgresql.org/.

[3] The standard performance evaluation corporation. SPECjbb2005 suite. http://www.spec.org/jbb2005/.

[4] R. Ahmed, R. Frazier, and P. Marinos. Cache-aided rollback error recovery (CARER) algorithm for shared-memory multiprocessor systems. In *20th International Symposium on Fault-Tolerant Computing, 1990 (FTCS-20) Digest of Papers.,* pages 82–88, 1990.

[5] A. R. Alameldeen and D. A. Wood. Variability in architectural simulations of multi-threaded workloads. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, page 7, 2003.

[6] D. F. Bacon and S. C. Goldstein. Hardware-assisted replay

of multiprocessor programs. In *Proceedings of the 1991 ACM/ONR workshop on Parallel and Distributed Debugging (PADD)*, 1991.

[7] P. Barford and M. Crovella. Generating representative web workloads for network and server performance evaluation. In *Proceedings of ACM Sigmetrics International Conference on Measurement and Modeling of Computer Systems*, 1998.

[8] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. BulkSC: Bulk enforcement of sequential consistency. In *ISCA '07: Proceedings of the 34th Annual International Symposium on Computer Architecture*, pages 278–289, 2007.

[9] M. Cintra, J. F. Martinez, and J. Torrellas. Architectural support for scalable speculative parallelization in shared-memory multiprocessors. In *ISCA '00: Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 13–24, 2000.

[10] S. Gopal, T. Vijaykumar, J. Smith, and G. Sohi. Speculative versioning cache. In *Fourth International Symposium on High-Performance Computer Architecture,* page 195, 1998.

[11] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, page 102, 2004.

[12] D. R. Hower and M. D. Hill. Rerun: Exploiting episodes for lightweight memory race recording. In *ISCA '08: Proceedings of the 35th International Symposium on Computer Architecture*, pages 265–276, 2008.

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

[14] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, v. 35 pages 50-58, 2002.

[15] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, 2005.

[16] P. Montesinos, L. Ceze, and J. Torrellas. Delorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In *ISCA '08: Proceedings of the 35th International Symposium on Computer Architecture*, pages 289–300, 2008.

[17] K. Moore, J. Bobba, M. Moravan, M. Hill, and D. Wood. LogTM: Log-based transactional memory. In *The Twelfth International Symposium on High-Performance Computer Architecture*, pages 254–265, 2006.

[18] S. Narayanasamy, C. Pereira, and B. Calder. Recording shared memory dependencies using strata. In *ASPLOS-XII: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 229–240, 2006.

[19] S. Narayanasamy, G. Pokam, and B. Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *ISCA '05: Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 284–295, 2005.

[20] R. H. B. Netzer. Optimal tracing and replay for debugging shared-memory parallel programs. In *PADD '93: Proceedings of the 1993 ACM/ONR workshop on Parallel and Distributed Debugging*, 1993.

[21] M. Prvulovic. Cord: cost-effective (and nearly overhead-free) order-recording and data race detection. In *The Twelfth International Symposium on High-Performance Computer Architecture*, pages 232–243, 2006.

[22] M. Prvulovic and J. Torrellas. ReEnact: using thread-level speculation mechanisms to debug data races in multithreaded codes. In *ISCA '03: Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 110–121, 2003.

[23] M. Ronsse, L. Levrouw, and K. Bastiaens. Efficient coding of execution-traces of parallel programs. In *ProRISC & IEEE-Benelux workshop on Circuits, Systems, and Signal Processing*, pages 251–258, 1995.

[24] D. Sanchez, L. Yen, M. D. Hill, and K. Sankaralingam. Implementing signatures for transactional memory. In *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 123–133, 2007.

[25] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood. SafetyNet: improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *ISCA '02: Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 123–134, 2002.

[26] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A scalable approach to thread-level speculation. In *ISCA '00: Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 1–12, 2000.

[27] Apache http Server. v2.2.11. The apache software foundation. http://httpd.apache.org/.

[28] Y.-M. Wang, Y. Huang, K.-P. Vo, P.-Y. Chung, and C. Kintala. Checkpointing and its applications, In *Proceedings of the 25th International Symposium on Fault-tolerant Computing,* page 22, 1995.

[29] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: characterization and methodological considerations. In *ISCA '95: Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, 1995.

[30] M. Xu, R. Bodik, and M. D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *ISCA '03: Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 122–135, 2003.

[31] M. Xu, M. D. Hill, and R. Bodik. A regulated transitive reduction (RTR) for longer memory race recording. In *ASPLOS-XII: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 49–60, 2006.