

# CORD: Cost-effective (and nearly overhead-free) Order-Recording and Data race detection \*

Milos Prvulovic

Georgia Institute of Technology  
<http://www.cc.gatech.edu/~milos>

## Abstract

Chip-multiprocessors are becoming the dominant vehicle for general-purpose processing, and parallel software will be needed to effectively utilize them. This parallel software is notoriously prone to synchronization bugs, which are often difficult to detect and repeat for debugging. While data race detection and order-recording for deterministic replay are useful in debugging such problems, only order-recording schemes are lightweight, whereas data race detection support scales poorly and degrades performance significantly.

This paper presents our CORD (Cost-effective Order-Recording and Data race detection) mechanism. It is similar in cost to prior order-recording mechanisms, but costs considerably less than prior schemes for data race detection. CORD also has a negligible performance overhead (0.4% on average) and detects most dynamic manifestations of synchronization problems (77% on average). Overall, CORD is fast enough to run always (even in performance-sensitive production runs) and provides the support programmers need to deal with the complexities of writing, debugging, and maintaining parallel software for future multi-threaded and multi-core machines.

## 1. Introduction

In the near future, multi-threaded and multi-core processors will become the dominant platform for general-purpose processing, and parallel software will be needed to effectively utilize them. Unfortunately, non-deterministic ordering of memory accesses across threads makes parallel software notoriously difficult to write and debug. This non-determinism can be intentional or unintentional. Intentional non-determinism increases parallelism, e.g. when threads are allowed to go through a critical section as they arrive, instead of forcing them to enter in a pre-determined order. Unintentional non-determinism is manifested through data races, typically due to missing or incorrect synchronization, and may cause a wide variety of incorrect program behavior. Both kinds of non-determinism make errors difficult to repeat for debugging. Additionally, unintentional non-

determinism can be manifested only intermittently and be very difficult to detect.

To repeat execution for debugging, the system must record outcomes of races (both intentional and unintentional) that determine the actual execution order at runtime. When a bug is detected, *deterministic replay* for debugging is achieved by replaying these race outcomes. For readability, *order-recording* will be used to refer to the activity of recording race outcomes for later deterministic replay. Note that order-recording proceeds together with normal application execution, while replay occurs only when a bug is found. Typical program execution (especially in production runs) is mostly bug-free, so recording efficiency is a priority.

A runtime manifestation of a synchronization problem can be detected by discovering at least one data race caused by that problem. Data races are conflicting memory accesses [22] whose order is not determined by synchronization [15]. Following this definition, a data race detector tracks synchronization-induced ordering during program execution, monitors conflicting data (non-synchronization) accesses, and finds a data race when conflicting data accesses are unordered according to the already observed synchronization order. Like order-recording, data race detection (DRD) proceeds together with normal application execution, so it should have a low performance overhead and avoid detection of false problems.

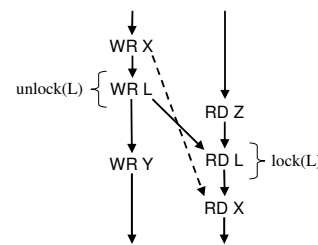


Figure 1. Example execution ordering.

The exact runtime ordering in an execution of a multi-threaded program defines a *happens-before* [11] relationship among memory access events in that execution. For two conflicting memory accesses  $A$  and  $B$ , either  $A$  happens before  $B$  ( $A \rightarrow B$ ) or  $B$  happens before  $A$  ( $B \rightarrow A$ ). Non-conflicting accesses can be concurrent (unordered), or they can be ordered transitively through program order and ordering of conflicting accesses. For example, a two-thread execution is shown in Figure 1. Vertical arrows represent program order within each

\*This work was supported, in part, by the National Science Foundation under Grant Number CCF-0447783. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

thread, while diagonal arrows represent ordering of cross-thread conflicts. In this example,  $WR\ X \rightarrow WR\ L$  due to program order,  $WR\ L \rightarrow RD\ L$  due to a race outcome, transitivity results in  $WR\ X \rightarrow RD\ L$ , and  $WR\ Y$  and  $RD\ X$  are concurrent. Note that  $WR\ X \rightarrow RD\ X$  is a conflict outcome, but this order is also due to transitivity.

For deterministic replay of this example, only  $WR\ L \rightarrow RD\ L$  needs to be recorded, because enforcement of that order also repeats the order of  $WR\ X$  and  $RD\ X$ . However, replay is correct if  $WR\ X \rightarrow RD\ X$  is also recorded. Recording either  $WR\ Y \rightarrow RD\ X$  or  $RD\ X \rightarrow WR\ Y$  (but not both) is also consistent with correct replay, although the accesses are actually concurrent. In fact, replay is correct even if we record  $WR\ L \rightarrow RD\ Z$  and then omit the actual race outcome  $WR\ L \rightarrow RD\ L$  as redundant. In general, an order-recording mechanism must record a consistent ordering (no cycles) that implies actual race outcomes, but need not precisely detect which accesses conflict or which conflicts are races – when in doubt, any pair of accesses can be treated as a race and its outcome recorded.

Data race detection (DRD) for execution in Figure 1 requires knowledge of which memory accesses are synchronization accesses and which are data accesses. Such knowledge can be obtained, for example, by marking synchronization load/store instructions in synchronization primitives (e.g. *lock*, *unlock*, *barrier* etc.). Using this knowledge, the detector records the ordering introduced by races among synchronization accesses and then looks for conflicting data accesses which are concurrent according to the previously observed synchronization-induced ordering. In Figure 1, when  $RD\ L$  executes the detector finds a conflict with  $WR\ L$  and records  $WR\ L \rightarrow RD\ L$  as synchronization-induced ordering. When  $RD\ X$  executes later, the detector finds the conflict with  $WR\ X$ , but also finds that their ordering is a transitive consequence of program order and synchronization. As a result, no data race is detected. It should be noted that a DRD mechanism must track ordering information much more precisely than an order-recorder. If, for example, the detector treats the conflict between  $WR\ X$  and  $RD\ X$  as a race, a problem is reported where in fact there is none. Similarly, a false problem is reported if the detector believes that  $WR\ Y$  and  $RD\ X$  conflict.

Both order-recording and data race detection can track access ordering and discover races using *logical time* [11]. Memory locations can be timestamped with logical timestamps of previous accesses, and each thread can maintain its current logical time. When a memory location is written to, its timestamps are compared to the current logical time of the writing thread. On a read access only the write timestamps are compared because at least one of the accesses in a conflict must be a write [22]. If the thread’s logical time is greater than the location’s timestamp, the two accesses are already ordered and no race is recorded or detected. If the location’s timestamp is greater than the thread’s logical time, or if they are unordered, a race is recorded and, in case of data accesses, a data race is found; the thread’s logical time is then updated to reflect this newly found ordering.

Tracking and detection of races for order-recording need not be very precise, so scalar logical clocks can be used and multiple memory locations can share a timestamp. Scalar clocks result in losing much concurrency-detection ability, while coarse-grain timestamps result in detection of false races between accesses to different locations that share a timestamp. However, such imprecision does not affect correctness of order-recording, so it can be implemented simply and with a very low performance overhead [26].

In contrast, it has been shown [24] that data race detection requires vector clocks [9] and per-word vector timestamps for all shared memory locations, to avoid detection of false data races and detect all real data races exposed by causality of an execution. Such clocks have been used in software tools [20], but often result in order-of-magnitude slowdowns which preclude always-on use. This has led to data race detection with hardware-implemented vector clocks in ReEnact [17]. However, ReEnact still suffers from significant performance overheads, scales poorly to more than a few threads, and has a considerable hardware complexity.

In this paper we find that, in most cases, a dynamic occurrence of a synchronization problem results in several data races, where detection of any one of these races points out the underlying synchronization problem. Based on this observation, we describe several key new insights that allow data race detection to be simplified and combined with an order-recording scheme. The resulting combined mechanism, which we call CORD (Cost-effective Order-Recording and Data race detection), is similar in complexity to order-recording schemes, but also provides practically useful data race detection support which reports no false positives and detects on average 77% of dynamic occurrences of synchronization problems.

To achieve the simplicity of CORD, we employ scalar clocks and timestamps, keep timestamps only for data in on-chip caches, and allow only two timestamps for each cached block. To avoid detection of numerous false data races due to out-of-order removal of timestamps from on-chip caches, CORD uses our novel main memory timestamp mechanism, which is fully distributed and has negligible cost and performance overheads. In contrast, ReEnact requires complex version combining and multi-version buffering mechanisms of thread-level speculation (TLS) for this purpose. In terms of timestamp storage, CORD uses on-chip state equal to 19% of cache capacity and supports any number of threads in the system. In contrast, vector timestamps used in prior work require the same amount of state to support only two threads, with state requirements growing in linear proportion to the number of supported threads.

To improve data race detection accuracy of CORD, we investigate typical data race scenarios which are not detected using basic scalar clocks. We find that many data races are missed because scalar clocks are often unable to capture the difference between clock updates due to synchronization and due to other events. To alleviate this problem, we propose a new clock update scheme that often succeeds in capturing this difference and, as a result, improves the problem detection rate of scalar clocks

by 62% without any significant increase in complexity or performance overheads.

In terms of performance, CORD has negligible overheads: 0.4% on average and 3% worst-case on Splash-2 benchmarks. These overheads are much lower than in prior work [17], mostly because CORD avoids the use of multi-version buffering and large vector timestamps and, as a result, uses on-chip cache space and bus bandwidth more efficiently.

The rest of this paper is organized as follows: Section 2 presents the CORD mechanism, Section 3 presents our experimental setup, Section 4 presents our experimental results, Section 5 discusses related work, and Section 6 summarizes our conclusions.

## 2 The CORD Mechanism

This section briefly reviews the background and definitions for order recording and data race detection. We then describe an ideal data race detection scheme and several considerations that limit the accuracy of practical schemes. Finally, we present the CORD mechanism, its main memory timestamp mechanism, its new clock update scheme, and some additional practical considerations.

### 2.1 Conflicts, Races, and Synchronization

Two accesses from different threads *conflict* if at least one of them is a write and they access the same memory location [22]. Conflicting accesses are *ordered* if their order follows from the order of previously executed conflicting accesses, otherwise they are *unordered*. An unordered conflict is also called a *race* [15], and all ordering across threads is determined by races outcomes. Because any memory access may potentially participate in one or more races, it is difficult to reason about parallel execution unless some programming convention is followed. To establish such a convention, *synchronization races* are distinguished from *data races*. Synchronization races occur between accesses *intended* to introduce ordering among threads. Accesses performed to lock/unlock a mutex are examples of synchronization accesses, and races between them are carefully crafted to introduce the intended ordering (e.g. mutual exclusion for a mutex). *Data races* occur between data (non-synchronization) accesses. An execution in which all data accesses are ordered by synchronization and program order is said to be *data-race-free*. Finally, if all possible executions of a program are data-race-free, the program itself is data-race-free or *properly labeled* [1].

Most parallel programs are intended to be properly labeled. However, this property is difficult to prove because all possible runs of a program must be taken into account. A more common approach is to test the program with a tool that detects data races at runtime [3, 4, 6, 8, 10, 16, 19, 20, 21]. If no data races are found in test runs, the program is assumed to be properly labeled. Unfortunately, a synchronization defect may be manifested through data races only under conditions that are not included in test runs. Such problems remain in deployed software and cause seemingly random occurrences of crashes

and erroneous results (*Heisenbugs*). Continuous data race detection (DRD) would eventually pinpoint these synchronization problems, and deterministic replay would allow these and other bugs to be debugged effectively.

### 2.2 Ideal Support for Order-Recording and Data Race Detection (DRD)

The ideal mechanism accurately identifies all races and correctly determines their type (synchronization or data). Race outcomes can then be recorded for deterministic replay and data races can be reported as bugs. This ideal race identification can be accomplished by timestamping each memory access. Upon a new access to a location, the thread's current logical time is compared to timestamps of previous conflicting accesses. To determine whether there is a conflict, the mode of each access (read or write) must also be recorded. Because the current access (e.g. a write) may conflict and have a race with many past accesses (e.g. reads) to the same variable, many timestamps per memory location must be kept to detect all races. In general, the number of timestamps per location and the number of locations for which timestamps must be kept can not be determined a priori. However, hardware support for unlimited access history buffering is impractical due to space limitations and due to performance overheads caused by numerous timestamp comparisons on each memory access.

The format of logical clocks and timestamps also poses a challenge for hardware implementation. Ideally, a comparison of two timestamps exactly indicates their current *happens-before* relationship, and a race is found when the comparison indicates that the accesses are concurrent. Logical vector clocks [9] are a classical scheme that accurately tracks the happens-before relationship. A vector clock or timestamp has one scalar component for each thread in the system, and it has been shown [24] that, if  $N$  threads may share memory, no clocking scheme with fewer than  $N$  scalar components can accurately capture the happens-before relationship. Unfortunately, hardware implementation of vector clocks is very problematic: fixed-size clocks and timestamps can be managed and cached relatively simply, but limit the number of parallel threads in the system. Conversely, variable-size clocks and timestamps would be very difficult to manage, store, and compare in hardware.

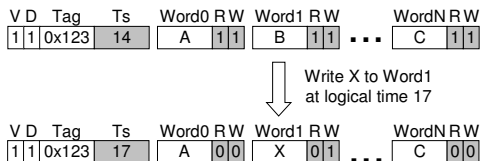
### 2.3 Realistic Access History Buffering

A practical hardware implementation of order-recording and DRD must have limited access history buffering requirements. To do this, we must allow false alarms, allow real races to sometimes be missed, or both. False alarms in production runs trigger costly debugging and error-reporting activities and inconvenience the user, so we need a scheme free of false alarms. As a result, our limited access history buffering will come at the cost of missing some real data races.

First, we limit the number of timestamps kept for each shared memory location. However, we do not simply keep a few most recently generated timestamps for each location, because then a thread that frequently accesses a shared location can "crowd out" other threads' timestamps. When that happens, a

synchronization problem will only be detected if accesses from the frequent-access thread are involved. As a result of this consideration, we keep the last read and the last write timestamp for each  $\langle \text{location}, \text{thread} \rangle$  pair. This follows existing work in debugging parallel programs, which indicates that beyond the first race for a manifestation of a problem, additional data races yield little additional insight into the problem [5, 14, 15, 17, 20]. With this per-thread approach, replacement decisions for access history entries are also simplified, because each thread’s processor can locally make its replacement decisions.

Our second simplification is to keep access histories only for locations that are present in the caches of the local processor. As discussed in Section 1, a timestamp should be associated with each word to prevent detection of false races. However, such per-word timestamps would result in a significant increase in cache complexity and affect cache hit times. For example, per-word vector timestamps, each with four 16-bit components, represent a 200% cache area overhead.



**Figure 2.** With only one timestamp per line, a timestamp change erases the line’s history (CORD state shown in gray).

We reduce the chip-area overhead by keeping a per-line timestamp. To avoid detection of false data races, we still associate the line’s timestamp with individual words by keeping two bits of per-word state. These bits indicate whether the word has been read and/or written with the line’s latest timestamp. This effectively provides per-word timestamps, but only for accesses that correspond to the line’s latest timestamp. However, a problematic situation is shown in Figure 2. It occurs when one word is accessed with a new timestamp (17 in our example). This timestamp replaces the old one and results in clearing all existing access bits. Note that before this latest access all words were effectively timestamped (in both read and write modes) with the old timestamp (14 in our example). This is a typical situation because spatial locality eventually results in setting access bits for most words. However, a timestamp change erases this history information and setting of access bits begins from scratch. To alleviate this problem, we keep two latest timestamps per line and two sets of per-word access bits, so the old timestamp and its access bits can provide access history for words that are not yet accessed with the newest timestamp. Our evaluation shows (Section 4.3) that with two timestamps per line we see little degradation in DRD ability, compared to an unlimited number of timestamps per word. Note that even a single timestamp, without per-word access bits, suffices for correct order-recording.

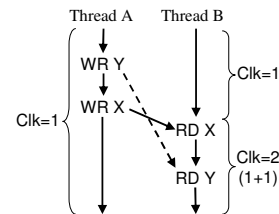
With 4x16-bit vector timestamps four threads can be supported, and with 64-byte cache lines the chip area overhead of timestamps and access bits is 38% of the cache’s data area. Next, we describe how 16-bit scalar clocks can be used to re-

duce this overhead to 19%, regardless of the number of threads supported.

## 2.4 Using Scalar Logical Time

Real systems may have many more threads than processors, so much larger (and more expensive) vector timestamps are needed. Since only data race detection (DRD) benefits from these vector timestamps, it is difficult to justify the on-chip area and the performance overhead associated with maintaining, updating, and comparing very large vector clocks.

To avoid these cost and scalability problems, CORD uses scalar timestamps (integer numbers) for both order-recording and DRD. We start with classical Lamport clocks [11], which are designed to provide total ordering among timestamped events. A Lamport clock consists of a sequence number and a thread ID. To determine the ordering of two logical times, their sequence numbers are compared first and the clock with the lower sequence number is assumed to happen before the one with the higher sequence number. Ties are broken using thread IDs<sup>1</sup>. When a thread accesses a memory location, the thread’s current logical clock is compared to the timestamps of conflicting accesses in the location’s access history. If a comparison indicates that the thread’s access happens after the history access, the conflict is assumed to be already ordered by transitivity. When the clock-timestamp comparison indicates that the thread’s access happens before the timestamp’s access, a race is found and the thread’s clock is updated to reflect the race outcome. This update consists of setting the clock’s sequence number to be equal to one plus the timestamp’s sequence number. Finally, each access has a unique timestamp because the thread’s clock is incremented after each access to a shared memory location.

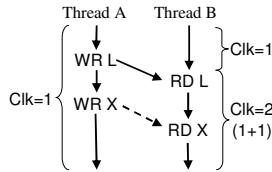


**Figure 3.** Data race on X results in an update of Thread B’s clock. This update prevents detection of data race on Y.

For order-recording and data race detection (DRD), total ordering of accesses is not needed – DRD actually works better with clocks that can express concurrency. Consequently, we can eliminate tie-breaking thread IDs and use a simple integer number as a logical clock. A race is now found when the thread’s current clock is less than *or equal to* the timestamp of a conflicting access in the access history. To reflect the new ordering, the thread’s logical clock is then incremented to become one plus the timestamp. For order-recording, these clock updates eliminate recording of redundant ordering. They also allow the simple recording format we use (Section 2.7.1). In DRD, we

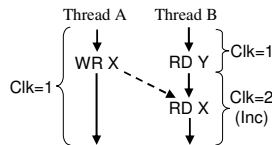
<sup>1</sup>If thread IDs are also equal, the two clocks belong to the same thread and program order defines their happens-before relationship

note that clock updates are needed on every synchronization race found, to prevent later detection of false data races. Updates on data races, however, may prevent detection of other data races because they can appear “synchronized” by a prior data race. Figure 3 show one such situation. The race on variable X is detected and the clock of Thread B is updated. As a result, the race on variable Y is not detected because Thread B’s clock, when it reads Y, is already larger than Y’s write timestamp. Fortunately, if synchronization bugs are not manifested often (a mostly data-race-free execution), such “overlapping” races are very likely caused by the same synchronization problem. The problem is detected by detecting any one of the races it causes, and when the problem is repaired all its races would be eliminated. As a result of these considerations, we choose to perform clock updates on all races.



**Figure 4.** Data race on X is missed if Thread A’s clock is not incremented after a write to synchronization variable L.

Lamport clocks are incremented on every “event” (in our case, every shared memory access). Clocks that advance so quickly would need to be large (e.g. 64-bit) to prevent frequent overflows, so we increment our clocks only on synchronization writes. These increments are needed by our DRD scheme to differentiate between pre-synchronization and post-synchronization accesses. Figure 4 shows an example in which Thread B reads the synchronization variable L and updates its clock to succeed L’s write timestamp of 1. If Thread A’s clock is not incremented after the synchronization write, its write to X is also timestamped with a logical time of 1, which is lower than Thread B’s clock of 2 when it reads X. As a result, the (real) data race on X is not detected. If Thread A’s clock was incremented after its write to synchronization variable L, the race on X would be detected. Due to this consideration, we increment a thread’s clock following each synchronization write. However, clock increments on reads and non-synchronization writes are unnecessary and would harm DRD (see Figure 5).

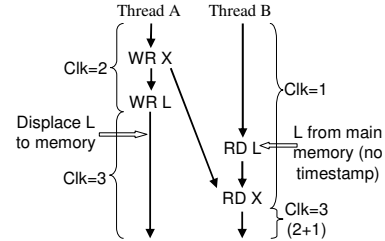


**Figure 5.** Data race on X is missed if Thread B’s clock is incremented after it reads Y.

## 2.5 Main Memory Timestamp

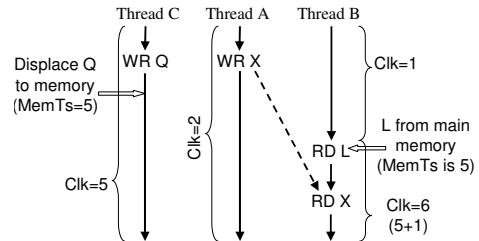
We want to avoid timestamping words or even blocks in main memory. However, a race can not be identified if it involves a location without any timestamps. Figure 6 shows an

example of this problem, where synchronization variable L is displaced from the cache after it is written in Thread A. As a result, when Thread B reads L it has no timestamp to compare against. If Thread B simply ignores this situation, replay is incorrect because the recorded ordering (which follows the thread’s clock) now indicates that Thread B reads L before Thread A writes it. Neglecting a synchronization race (Figure 6) also results in detection of a false data race on X.



**Figure 6.** Synchronization on L is missed without memory timestamps, resulting in incorrect order-recording and detection of false data race on X.

Our solution is to keep one read and one write timestamp for the entire main memory. Main memory timestamps are updated when a per-line timestamp and its access bits are removed to make room for a newer timestamp. If any of the access bits indicates a read access to any word, the line’s timestamp overwrites the memory read timestamp if the line’s timestamp is larger. The memory write timestamp is similarly updated if any per-word write access bit is set and the line’s timestamp is larger than the memory write timestamp. Effectively, the entire main memory becomes a very large block that shares a single timestamp, which allows correct order-recording.



**Figure 7.** Displacement of Q to memory in Thread C leads Thread B to update its clock when it reads lock L from memory, resulting in a missed data race on X.

However, the many-words granularity of main-memory timestamps prohibits precise data race detection. Figure 7 shows an example in which the main memory timestamp is updated on a write-back of Q in Thread C. Thread B then reads a synchronization variable L from memory, updates Thread B’s logical clock using the main memory timestamp, and consequently misses the data race on variable X. We note that Thread B must perform a clock update because it can not determine whether or not the memory write timestamp corresponds to a write-back of variable L. Because false positives are more damaging in production runs than undetected races, and because

order-recording also benefits from clock updates using memory timestamps, in CORD we choose to perform such clock updates even at the cost of missing some data races. Finally, if variable L in Figure 7 were not a synchronization variable, when Thread B reads it from memory and compares its clock against the main memory timestamp, the comparison would indicate that L is involved in a data race (which is not true). Fortunately, we can simply ignore (and not report) any data race detections that used a main memory timestamp. This may result in missing a real data race through memory, but avoids reporting false ones.

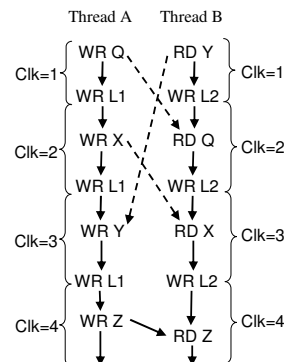
Maintaining the two main memory timestamps is relatively easy in a small-scale system with snooping - each cache can maintain its own copy of the two main memory timestamps. When a cache line or its oldest access history entry is displaced, its timestamp is compared to the local main memory timestamp. If this results in changing the local main memory timestamp, a special memory timestamp update transaction is broadcast on the bus (or piggy-backed to the write-back, if there is one). Other processors can then update their own memory timestamps. A straightforward extension of this protocol to a directory-based system is possible, but in this paper we focus on systems (CMPs and SMPs) with snooping cache coherence.

We note that ReEnact [17] performs data race detection (DRD) without timestamping main memory at all. However, ReEnact uses thread-level speculation (TLS) support with special version combining mechanisms to avoid detection of false races when synchronization through memory is missed. Also, like our new main memory timestamps, ReEnact’s approach misses all real data races through non-cached variables. Consequently, our new main memory timestamps provide the same order-recording and DRD capabilities without the complexity and performance overheads of TLS.

## 2.6 Sync-Read Clock Updates Revisited

As described in prior sections, our CORD mechanism correctly records execution order for deterministic replay, avoids detection of false data races, and detects some data races. Still, we find that many injected manifestations of synchronization problems were undetected, and identify three main reasons for missing them. First, some data races involved accesses that are too far apart to be detected using only in-cache timestamps with limited access histories. Any architectural mechanism that does not individually timestamp main memory locations and limits in-cache access histories would suffer from this problem, but to our knowledge this paper is the first to quantify it (Section 4.3). Second, some data races were missed because scalar clocks can not precisely capture the ordering across threads. This problem is quantified in Section 4.4. The third class of missed data races is described below, together with a new clock update mechanism that vastly improves their detection.

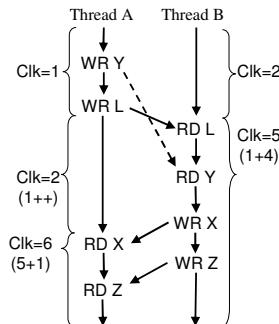
In some of the applications we study, synchronization writes occur at about the same rate in all threads. As a result, a thread’s current scalar clock tends to be larger than timestamps of variables accessed earlier by other threads. Figure 8 shows an example of this problem. In this example, only data races that involve nearly simultaneous pairs of accesses (e.g. the race on



**Figure 8.** Synchronization writes on L1 and L2 update thread’s clock, so they appear synchronized and data races on Q, X, and Y are missed. Only nearly-simultaneous data races (like the one on X) are detected.

Z) are detected. This problem would be eliminated if we eliminate clock increments on synchronization writes, but then many other data races would be missed (Section 2.4).

Our solution to this problem is to increase the “window of opportunity” for data race detection when logical clocks and timestamps are compared. Two accesses are normally considered synchronized if the logical clock of the second one is larger than the timestamp of the first one. However, in general we can require the timestamp of the second access to be larger by at least  $D$  for the two accesses to be considered synchronized. If  $D > 2$ , all data races in Figure 8 would be detected.



**Figure 9.** Sync-read on L creates a clock difference of  $D = 4$ , while other clock updates and clock increments use a difference of one. This allows correct treatment of the synchronized conflict on Y, detection of a data race on X, and even detection on a data race on Z.

Of course, synchronization reads must now update the thread’s clock to be at least  $D$  larger than the synchronization variable’s latest write timestamp. All other updates and clock increments continue to use an increment of one. An example of this behavior is shown in Figure 9, where the synchronization read in Thread B is updated to be  $D = 4$  larger than the lock’s write timestamp, but Thread A’s clock is incremented by only one following that write. When the race on X is found, Thread B’s clock is updated to be larger by only one than X’s

timestamp. This last update allows the order-recorder to avoid redundant recording of the race on Y, but this race is still detected by our data race detection (DRD) scheme. Note that the order-recorder can still omit races in which the second access has a clock greater than the timestamp of the first access. If such a difference is less than  $D$ , the accesses can still be treated as transitively ordered, but the DRD mechanism can detect a data race because it knows that this ordering is not through synchronization. The change in  $D$ , therefore, directly affects only the DRD scheme and allows it to detect more data races without detecting false positives.

## 2.7 Implementation Details

**2.7.1. Format for Order-Recording.** The processor chip contains two registers, which contain physical memory addresses of the next available location in the execution order log and of the end of the region reserved for the log. In our experiments we reserve sufficient log space for the entire application run, but if this space does run out, an exception is triggered to save the log or allocate more space.

When a thread's clock changes, it appends to the log an entry that contains the previous clock value, the thread ID and the number of instructions executed with that clock value. We use 16-bit thread IDs and clock values and 32-bit instruction counts, for a total of eight bytes per log entry. To prevent instruction count overflow, the thread's clock is simply incremented when its instruction count is about to overflow. This happens rarely, is compatible with correct order-recording, results in no detection of false data races, and has a negligible impact on detection of real races.

Our deterministic replay orders the log by logical time and then proceeds through log entries one by one. For each log entry, the thread with the recorded ID has its clock value set to the recorded clock value, and is then allowed to execute the recorded number of instructions. Note that during recording, if conflicting accesses have the same logical clock, we update the clock of one of the accesses. As a result, only non-conflicting fragments of execution from different threads can have equal logical clocks, in which case they can be replayed in any order. Optimizations are possible to allow some concurrency in replay, but this is a secondary concern because the replay occurs only in debugging when a problem is found.

**2.7.2. Clock Comparisons.** A memory access instruction results in an attempt to find the requested variable in the local cache. If the variable is not found, a bus request, tagged with the processor's current logical clock, is sent to fetch the block from memory or another cache. Snooping hits in other caches result in data race checks. Data responses are tagged with the data's timestamp and result in a clock update on the requesting processor. Memory responses use the main memory timestamps instead.

If the processor's access is a cache hit, both timestamps for the in-cache block can be lower than the current logical clock. In that case, the lower of the two timestamps and its access bits are removed (potentially causing a main memory timestamp update) and the new timestamp takes its place, with all access bits

reset to zero. An access that finds the corresponding access bit to be zero results in broadcasting a special *race check request* on the bus. This request is similar in effect to a cache miss, but without the actual data transfer. We also maintain two *check filter* bits per line that indicate when the entire line can be read and/or written without additional race check requests. As a result, each race check request actually results in the entire line being checked. A race on the particular word being accessed results in normal order-recording and data race detection activity. Absence of any potential conflicts for the entire line results in a permission to set one or both of the check filter bits. Note that our race check requests only use the less-utilized address and timestamp buses and cause no data bus contention.

Our clock comparisons are performed using simple integer subtractions (when  $D > 1$ ) and comparisons. A timestamp comparison for order-recording can use a simple comparator, while for data race detection with  $D > 1$  we need a subtractor and then a comparator. With two timestamps per cache block, each snooping hit and cache access results in two subtractions and four comparisons, which are easily handled by simple dedicated circuitry. We note that main memory itself is oblivious to our CORD scheme – our main memory timestamps are kept and compared on-chip.

**2.7.3. Synchronization.** Our scheme must distinguish between synchronization and data accesses, because they require different treatment: data race checks should only be done for data accesses, while synchronization accesses result in clock updates. To tell synchronization and data accesses apart, we rely on modified synchronization libraries that use special instructions to label some accesses as synchronization accesses. A similar approach is used in other data race detection work [6, 10, 17, 19, 20, 21].

**2.7.4. Thread Migration.** A practical data race detection (DRD) mechanism must allow a thread to migrate from one processor to another. However, such migration might cause false alarms in a DRD scheme because the timestamps of the thread remain on its previous processor and appear to belong to another thread. Since our logical clocks are not incremented on each memory access, an access by a migrated thread can find the data in the cache of its original processor, timestamped with the same timestamp, and with access bits that indicate a "conflict". As a result, a data race is falsely detected.

This problem is eliminated by simply incrementing the clock of a thread by  $D$  each time it begins running on a processor. With this simple modification, the self-race problem can no longer occur because new execution is "synchronized" with prior execution on the other processor. This can allow some data races to occur undetected, but thread migration occurs far less frequently than other clock updates. In our experiments, no data races are missed solely due to clock increments on thread migration. We note that a vector-clock scheme would suffer from the same thread migration problem, and that our "synchronize on migration" solution also applies to vector-clock schemes.

**2.7.5. Clock Overflow.** We use 16-bit logical clocks and timestamps, to reduce the cache area overhead. These clocks

can overflow and result in order-recording problems (missed races) or false positives in DRD. Fortunately, we find that we can use a sliding window approach with a window size of  $2^{15} - 1$ . This approach requires a slight modification in our comparator circuitry and requires elimination of very stale timestamps to prevent them from exiting the window. For this purpose, we use a cache-walker that uses idle cache ports to look up timestamps of in-cache blocks and trigger eviction of very old ones. During each pass, the walker also determines the minimum (oldest) timestamp still present in the cache. This minimum timestamp is kept by each cache and used to stall any clock update that would exceed the sliding window size. In our experiments no such stalls actually occur, because the cache walker is very effective in removing very old timestamps before they become a problem. Another concern is that removal of these old timestamps might prevent our DRD scheme from detecting some races. However, in our experiments we find that removed timestamps are much lower than current clock values in all threads, so for reasonable values of  $D$  no races can be detected with these old timestamps anyway.

**2.7.6. Recovery from Synchronization Problems.** CORD’s order-recording mechanism can be combined with a checkpointing mechanism to allow automated analysis and recovery from detected synchronization problems. Possible approaches for such recovery are to use pattern matching to identify and repair the dynamic instance of the problem [17], or to use conservative thread scheduling to serialize execution in the vicinity of the problem [27]. However, such recovery is beyond the scope of this paper, which focuses on cost-effective support for data race detection and order-recording.

## 3 Experimental Setup

### 3.1 Simulation Environment

We use a cycle-accurate execution-driven simulator of 4-processor CMP with private L1 and L2 caches. The 4-issue, out-of-order, 4GHz processor cores are modeled after Intel’s Pentium 4. They are connected using a 128-bit on-chip data bus operating at 1GHz, and the processor chip is connected to the main memory using a 200MHz quad-pumped 64-bit bus. Round-trip memory latency is 600 processor cycles, and the on-chip L2 cache-to-cache round-trip latency is 20 cycles. All other parameters, such as branch predictors, functional units, etc. represent an aggressive near-future design.

Because we use relatively small input sets for our applications, we use reduced-size caches to preserve realistic cache hit rates and bus traffic, using Woo et al. [25] as a reference. Each L2 cache is only 32KB and each L1 cache is 8KB in size. This reduced cache size also yields a realistic evaluation of data race detection accuracy, which would benefit from the numerous timestamps available in larger caches.

In our evaluation of performance overhead, the processor consumes data values on cache hits without waiting for a CORD timestamp comparison. Thus, we do not model any additional cache hit latency for CORD, but we do model bus and cache contention due to race check requests and the (rare) retirement

delay for each instruction whose CORD race check is still in progress when the instruction is otherwise ready to be retired.

### 3.2 Applications

To evaluate our CORD mechanism, we use Splash-2 [25] applications. These applications and the input sets we use are listed in Table 1. We note that some of these input sets are reduced from the default Splash-2 input sets. The reason for this is that, with our *Ideal* configuration, we keep (and check for data races) the entire history of accesses for every word the application accesses, and recycle a history entry only when it is guaranteed not to have any more races. This results in enormous memory space requirements - in some applications the simulation uses hundreds of megabytes even with the reduced input sets we use. In Radiosity, even with the test input set, simulation of *Ideal* runs exceeds the 2GB virtual address limitation.

App.	Input	App.	Input
barnes	n2048	cholesky	tk23.0
fft	m16	fmm	2048
lu	512x512	ocean	130x130
Radiosity	-test	radix	256K keys
raytrace	teapot	volrend	head-sd2
water-n2	216	water-sp	216

**Table 1.** Applications evaluated and their input sets.

### 3.3 Order-Recording

We performed numerous tests, with and without data race injections, to verify that the entire execution can be accurately replayed. Our order logs are very compact and in all applications require less than 1MB for the entire execution.

We note, however, that our system currently does not include checkpointing or I/O recording capabilities, so it can only replay an entire execution and requires manual replay of its inputs. This is not a problem in Splash-2 applications, in which the only inputs are files, but can be a problem in interactive applications. Checkpointing and I/O recording are largely orthogonal to the order-recording mechanism, and our CORD mechanism can be used with existing mechanisms for checkpointing and/or I/O replay, such as RecPlay [20], FDR [26], ReVive [18], BugNet [13], Flashback [23], etc. Our order-recording (and data race detection) results in negligible performance overheads (Section 4.1), so combined scheme’s overhead would mostly depend on the checkpointing and I/O recording scheme.

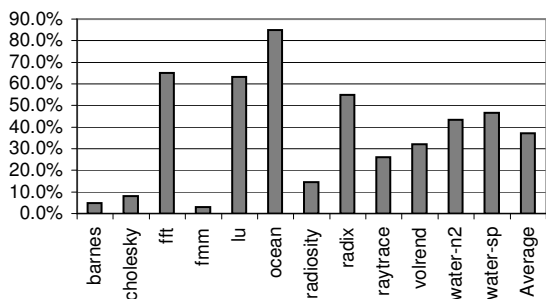
### 3.4 Data Races

Several Splash-2 applications already have data races that are discovered by CORD. Almost all are only potential portability problems, but at least one is an actual bug. All of these existing races can be removed by simple changes in the code and without much performance degradation, and most of the problems can be discovered even on very small input sets. Overall, these existing problems can be found by existing software tools and repaired before the software actually ships. In this paper, our goal is continuous testing in production runs. Therefore, we evaluate the effectiveness of each configuration in detecting

those elusive synchronization problems that escape testing, remain in production codes, are manifested sporadically, and offer only rare opportunities for detection.

We model this kind of error by injecting a single *dynamic instance* of missing synchronization into each run of the application. Injection is random with a uniform distribution, so each dynamic synchronization operation has an equal chance of being removed. When a synchronization instance to be removed is chosen, it is removed in a manner that depends on the type of synchronization. For mutex synchronization, we ignore a dynamic instance of a call to the “lock” primitive and the corresponding call to “unlock”. For `flag` (condition variable) synchronization, we ignore a single call to the `flag` “wait” primitive. Barrier synchronization uses a combination of mutex and `flag` operations in its implementation and each dynamic invocation of those mutex and `flag` primitives is treated as a separate instance of synchronization. This treatment of barriers results in more difficult detection than if an entire call to “barrier” is ignored – removing a single such call results in thousands of data races, one of which is virtually certain to be found. This would defeat our intent to model elusive, difficult to detect errors.

It is important to note that each of the error injections described above is performed in a separate simulation run to let us tell whether each injected error was detected. For each configuration we perform between 20 and 100 injections per application. The number of injections depends on the application because, somewhat surprisingly, in several applications most dynamic instances of synchronization are redundant - they result in no new cross-thread ordering and their removal creates no data races. Figure 10 shows the percentage of injections that actually resulted in an data races, as detected by the *Ideal* configuration which detects all dynamically occurring data races. We also check and confirm that program outputs are indeed correct in all injection runs for which *Ideal* finds no data races. Further analysis reveals that, in most of these injections, we removed a dynamic instance of a critical section protected by a lock that was previously held by the same thread, in which case the removed synchronization introduces no new cross-thread ordering. This finding further stresses the importance of always-on detection, because it indicates that synchronization defects in the code may become detectable only infrequently.



**Figure 10.** Percentage of injected dynamic instances of missing that resulted in at least one data race.

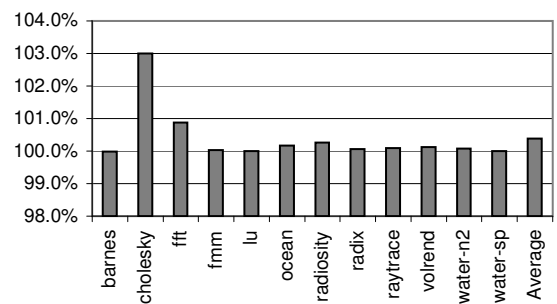
Since many injection runs result in no detectable data races, for some applications the number of runs with data races is too small to allow us to draw per-application conclusions, despite a large number of injection runs and thousands of processor-hours of simulation time. For example, we perform 100 injection runs per configuration in `fmm`, but get only 3 errors. However, our results still yield significant insight when comparing the configurations. This is especially true for averages across all applications, which are based on more than a hundred manifested errors per configuration.

## 4 Experimental Evaluation

In this section we evaluate our new CORD scheme, first its performance overhead and then its usefulness in detecting synchronization problems. Our order-recording is active in all experiments and we present no separate evaluation of its performance overhead because the overhead of the entire CORD mechanism is already negligible. Also, no separate evaluation is presented for the effectiveness of the order-recorder, because it always enables fully accurate replay.

### 4.1 Performance Overhead

Figure 11 shows the execution time with CORD, relative to a machine without any order-recording or data race detection support. We see that CORD imposes a very low performance overhead, 0.4% on average with a maximum of 3% for Cholesky. The increased overhead in Cholesky is primarily due to the increased on-chip address/timestamp bus contention. This bus is ordinarily less occupied than the data bus, so it runs at half the frequency of the data bus. Frequent synchronization in Cholesky results in many timestamp changes, which cause bursts of timestamp removals and race check requests on subsequent memory accesses. This, in turn, increases the contention for the address/timestamp bus.



**Figure 11.** Execution time with CORD relative to the Base-line which has no order-recording and no DRD support.

### 4.2 Data Race Detection

We use two criteria to compare effectiveness of data race detection (DRD) schemes. One criterion for DRD effectiveness is the actual *raw data race detection rate*, defined as the number of data races detected. The other criterion we call *problem detection rate*. The main purpose of a DRD mechanism is to find dynamic occurrences of synchronization problems. When

such an occurrence is found, it can be replayed, analyzed, and the problem repaired. As a result, the problem detection rate is defined as the number of runtime manifestations of synchronization errors for which at least one data race is detected. Our experience in debugging parallel software indicates that usefulness of a DRD mechanism is mostly dependent on its problem detection rate, so it serves at the main evaluation criterion for CORD’s effectiveness. Raw data race detection rates are also presented to provide additional insight.

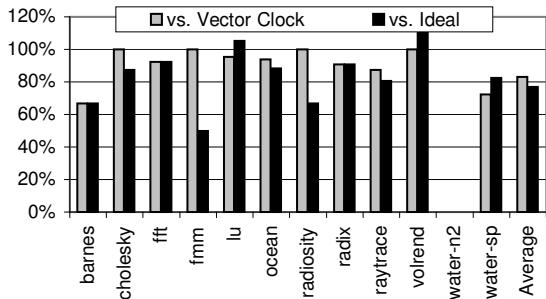


Figure 12. Problem detection rate.

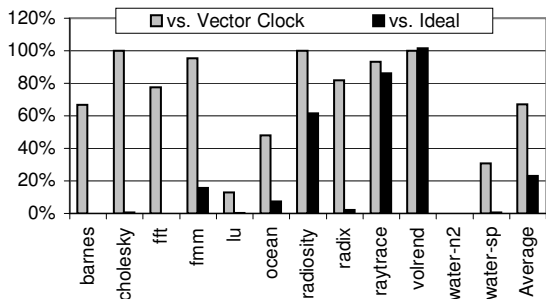


Figure 13. Raw data race detection rate.

Figure 12 shows CORD’s problem detection rate, relative to the problem detection rate of a CORD-like scheme that uses vector clocks. On average, our decision to use scalar clocks results in detecting only 83% of the problems that would be detected with vector clocks. We also note that in water-n2 CORD finds none of the problems, although several are found by the Ideal scheme and some are found by a vector-clock scheme. This stresses the fact that the choice between vector clocks and scalar clocks is a tradeoff and not a trivial decision. In our case, we target a cost-effective hardware implementation, in which case our scalar clocks offer an excellent tradeoff between cost, performance, and problem detection rates. Figure 12 also compares CORD’s problem detection to that of the *Ideal* configuration, which uses unlimited caches, unlimited number of history entries per cache line, and vector clocks. We see that our simple, implementable, and high-performance CORD scheme still detects the majority (77% on average) of all problems detected by an ideal data race detection scheme.

In Figure 12, CORD finds 18 problems for Volrend, two more than the ideal scheme. This seemingly contradictory result is due to the non-deterministic nature of parallel execution.

Our fault injector randomly generates a number  $N$  and then injects a fault into  $N$ -th dynamic instance of synchronization for each configuration. However, fault injections for *CORD* and for *Ideal* are performed in different runs, so the  $N$ th instance of synchronization in one run may be different from the  $N$ th instance in the other run. Execution order in different configurations changes mainly due to different cache hit/miss scenarios (Ideal’s L2 cache is infinite and always hits).

Figure 13 shows CORD’s effectiveness in detecting raw data races. Again, the results are relative to the vector-clock scheme and the Ideal scheme. An interesting trend can be observed by comparing Figures 12 and 13. We see that CORD’s ability to detect raw data races is significantly diminished (only 20% of Ideal). However, there seems to be little clustering of data races that CORD does and doesn’t find. As a result, for a problem that causes several data races there is a large probability that one of those races will be detected. This finding indicates that the choices we made when simplifying CORD’s hardware mechanisms (limited access histories and buffering, scalar clocks) have largely resulted in sacrificing the less valuable raw data race detection capability, but still retain much of the more useful problem detection capability.

### 4.3 Impact of Limited Access Histories on Data Race Detection (DRD)

To gain better insight into tradeoffs of designing a DRD mechanism, we now evaluate how access history limitations alone affect the effectiveness of a DRD scheme. The *Ideal* configuration uses vector clocks, unlimited caches, and an unlimited number of access history entries per cache block. The *InfCache* configuration still uses vector clocks and unlimited caches but limits the number of timestamps per block to two. The *L2Cache* configuration uses vector clocks, but with our default L2 cache size and two timestamps (with their per-word access bits) per line, and *L1Cache* maintains two vector timestamps and their access bits per line only in our L1 caches. Problem detection rates of these schemes are shown in Figure 14. Few problems are missed due to a limited number of timestamps per cache line, and limiting access histories to the L2 cache results in a small additional loss of problem detection. A severe limitation (*L1Cache*) degrades problem detection significantly, although most problems are detected even then.

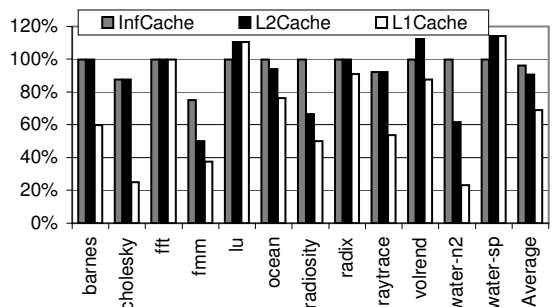
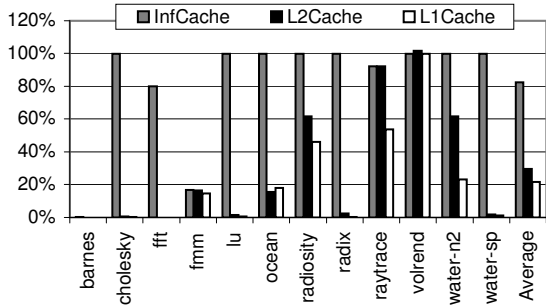


Figure 14. Problem detection rate with limited access histories and buffering limitations.



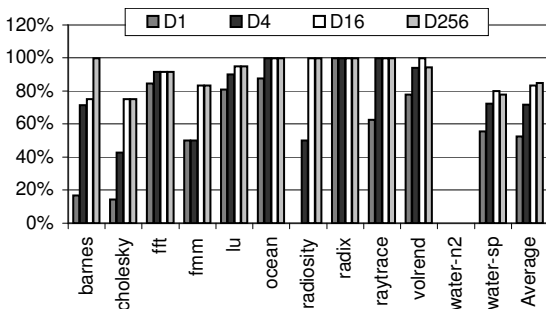
**Figure 15.** Raw data race detection rate with limited access histories and buffering limitations.

Figure 15 shows raw data race detection rates for the same four configurations. We see that most races are missed in *L2Cache* and *L1Cache*, and even an unlimited cache with only two timestamps per line misses 18% of all data races.

Overall, these results indicate that it is possible to effectively detect synchronization problems with a limited number of available timestamps and timestamped memory locations, although raw data race detection rates diminish when the number of timestamped locations is limited. We also find that a severely constrained system (*L1Cache*) has considerably worse problem detection rate than a moderately constrained one.

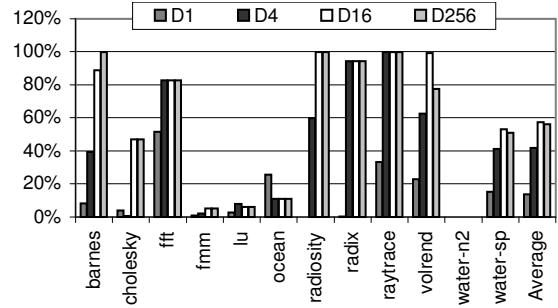
#### 4.4 Impact of Clock Limitations

In this section we evaluate the impact of clocking limitations on the effectiveness of data race detection. All configurations used in this evaluation use two timestamps per line in L1 and L2 caches. The results are all relative to a scheme that uses vector clocks (the *L2Cache* configuration in Section 4.3). We show a CORD configuration without additional sync-read clock updates described in Section 2.6 ( $D = 1$ ), and CORD configurations with sync-read clock updates of  $D = 4$ ,  $D = 16$ , and  $D = 256$ .



**Figure 16.** Synchronization problem detection with our scalar clocks.

Figures 16 and 17 show the results for problem detection and raw data race instruction. We observe that, compared to vector clocks, scalar clocks with  $D = 1$  result in a major loss of raw data race detection ability, but only a moderate loss of problem detection capability. We also find significant improvement in



**Figure 17.** Raw data race detection with different logical clock mechanisms.

both problem detection and raw data race detection as  $D$  increases up to 16. For values of  $D$  beyond 16, we see additional improvement only in barnes.

## 5 Related Work

Much work has been done in software tools and methods for order-recording and detection of data races or serializability violations such as RecPlay [20], Eraser [21], ParaScope [6], and others [3, 4, 8, 10, 16, 19, 27]. All of these schemes, especially those that detect data races or serializability violations, suffer from significant performance overheads. Low-overhead order-recording hardware has been proposed by Xu et al. [26], but without DRD support. Hardware has also been proposed for detection of races that violate a consistency model [2], or for specifically structured programs [3, 12]. A more recently proposed hardware mechanism, called ReEnact [17], supports both order-recording and DRD but uses vector clocks, thread-level speculation support, and multi-version caches. The complexity of this mechanism is considerable and performance overhead are non-negligible, mainly due to multi-version buffering which reduces the effective cache capacity by storing different versions of the same block in the same cache, and also due to large vector timestamps that increase bus traffic. However, vector clocks used by ReEnact allow detection of 22% more instances of synchronization problems than scalar clocks used in our CORD mechanism.

Finally, different clocking schemes are discussed and compared in prior work [7, 24], but mostly in the context of distributed systems and without considerations related to hardware constraints. Scalar clocks used there are not limited in size, are updated and compared differently than in our CORD scheme, and do not employ our  $D > 1$  sync-read clock increments.

## 6 Conclusions

This paper presents our new Cost-effective Order-Recording and Data race detection (CORD) mechanism. This mechanism is comparable in cost to existing order-recording schemes and considerably less complex than prior hardware support for data race detection. CORD also has a *negligible performance overhead* of 0.4% on average and 3% worst-case across Splash-2 benchmarks. Still, CORD records the execution order for *ac-*

curate deterministic replay, and detects most dynamic manifestations of synchronization problems (77% on average) in our problem-injection experiments.

We also quantitatively determine the effect of various limitations and enhancements in our CORD mechanism. Realistic buffering limitations result in missing 9% of synchronization problems. Note that these limitations are present in prior architectural schemes for data race, detection but their effect was not quantified. The additional transition from classical vector clocks to our scalar clocks, which are needed to reduce complexity and scale to more than a few threads, results in missing an additional 17% of synchronization problems. Finally, we find that our optimizations of the clock update scheme result in finding 62% more problems than a naive scalar clock update scheme.

We conclude that, with a careful quantitative approach, hardware for detection of most (77% in our experiments) synchronization problems can be achieved at a very low cost, without false alarms, and with a negligible performance overhead (0.4% on average in our experiments). Furthermore, the same hardware also records execution ordering for deterministic replay. This combined CORD mechanism is simple enough to be implementable, is fast enough to run always (even in performance-sensitive production runs), and provides the support programmers need to deal with the complexities of designing and supporting parallel software for future chip-multiprocessors and multi-threaded machines.

## References

- [1] S. V. Adve and M. D. Hill. Weak ordering - a new definition. In *17th Intl. Symp. on Computer Architecture*, pages 2–14, 1990.
- [2] S. V. Adve, M. D. Hill, B. P. Miller, and R. H. B. Netzer. Detecting Data Races on Weak Memory Systems. In *18th Intl. Symp. on Computer Architecture*, pages 234–243, 1991.
- [3] K. Audenaert and L. Levrrouw. Space efficient data race detection for parallel programs with series-parallel task graphs. In *3rd Euro-micro Workshop on Parallel and Distributed Processing*, page 508, 1995.
- [4] J.-D. Choi, K. Lee, A. Loginov, R. O’Callahan, V. Sarkar, and M. Sridharan. Efficient and Precise Datarace Detection for Multithreaded Object-Oriented Programs. In *ACM SIGPLAN 2002 Conf. on Prog. Lang. Design and Implementation*, pages 258–269, 2002.
- [5] J.-D. Choi and S. L. Min. Race Frontier: Reproducing Data Races in Parallel-Program Debugging. In *3rd ACM SIGPLAN Symp. on Principles & Practice of Parallel Programming*, pages 145–154, 1991.
- [6] K. D. Cooper, M. W. Hall, R. T. Hood, K. Kennedy, K. S. McKinley, J. M. Mellor-Crummey, L. Torczon, and S. K. Warren. The ParaScope Parallel Programming Environment. *Proc. of the IEEE*, 81(2):244–263, 1993.
- [7] K. De Bosschere and M. Ronsse. Clock snooping and its application in on-the-fly data race detection. In *third Intl. Symp. on Parallel Architectures, Algorithms, and Networks (I-SPAN)*, pages 324–330, 1997.
- [8] A. Dinning and E. Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *second ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 1–10, 1990.
- [9] C. Fidge. Logical Time in Distributed Computing Systems. *IEEE Computer*, 24(8):23–33, 1991.
- [10] R. Hood, K. Kennedy, and J. Mellor-Crummey. Parallel program debugging with on-the-fly anomaly detection. In *1990 Conf. on Supercomputing*, pages 74–81, 1990.
- [11] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978.
- [12] S. L. Min and J.-D. Choi. An Efficient Cache-Based Access Anomaly Detection Scheme. In *4th Intl. Conf. on Arch. Support for Prog. Lang. and Operating Sys.*, pages 235–244, 1991.
- [13] S. Narayanasamy, G. Pokam, and B. Calder. BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging. In *32nd Intl. Symp. on Computer Architecture*, 2005.
- [14] R. H. B. Netzer and B. P. Miller. Improving the accuracy of data race detection. In *third ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 133–144, 1991.
- [15] R. H. B. Netzer and B. P. Miller. What are race conditions?: Some issues and formalizations. *ACM Letters on Prog. Lang. and Systems*, 1(1):74–88, 1992.
- [16] D. Perkovic and P. J. Keleher. A Protocol-Centric Approach to On-the-Fly Race Detection. *IEEE Trans. on Parallel and Distributed Systems*, 11(10):1058–1072, 2000.
- [17] M. Prvulovic and J. Torrellas. ReEnact: Using Thread-Level Speculation Mechanisms to Debug Data Races in Multithreaded Codes. In *30th Intl. Symp. on Computer Architecture*, pages 110–121, 2003.
- [18] M. Prvulovic, Z. Zhang, and J. Torrellas. ReVive: Cost-Effective Architectural Support for Rollback Recovery in Shared-Memory Multiprocessors. In *29th Intl. Symp. on Computer Architecture*, pages 111–122, 2002.
- [19] B. Richards and J. R. Larus. Protocol-based data-race detection. In *SIGMETRICS Symp. on Parallel and Distributed Tools*, pages 40–47, 1998.
- [20] M. Ronsse and K. De Bosschere. RecPlay: A Fully Integrated Practical Record/Replay System. *ACM Trans. on Computer Systems*, 17(2):133–152, 1999.
- [21] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multi-Threaded Programs. *ACM Trans. on Computer Systems*, 15(4):391–411, 1997.
- [22] D. Shasha and M. Snir. Efficient and Correct Execution of Parallel Programs that Share Memory. *ACM Trans. on Prog. Lang. and Systems*, 10(2):282–312, 1988.
- [23] S. Srinivasan, C. Andrews, S. Kandula, and Y. Zhou. Flashback: A Light-weight Extension for Rollback and Deterministic Replay for Software Debugging. In *Usenix technical conference (USENIX)*, 2004.
- [24] C. Valot. Characterizing the Accuracy of Distributed Timestamps. In *1993 ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 43–52, 1993.
- [25] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *22nd Intl. Symp. on Computer Architecture*, pages 24–38, 1995.
- [26] M. Xu, R. Bodik, and M. D. Hill. A “Flight Data Recorder” for Enabling Full-System Multiprocessor Deterministic Replay. In *30th Intl. Symp. on Computer Architecture*, pages 122–135, 2003.
- [27] M. Xu, R. Bodk, and M. D. Hill. A Serializability Violation Detector for Shared-Memory Server Programs. In *ACM SIGPLAN 2005 Conf. on Prog. Lang. Design and Implementation (PLDI)*, 2005.