Low-Overhead and High Coverage Run-Time Race Detection Through Selective Meta-Data Management

Ruirui Huang
Intel Corporation
Hillsboro, OR 97124, USA
ruirui.huang@intel.com

Erik Halberg, Andrew Ferraiuolo, G. Edward Suh
Cornell University
Ithaca, NY 14850, USA
{esh64,af433,gs272}@cornell.edu

Abstract

This paper presents an efficient hardware architecture that enables run-time data race detection with high coverage and minimal performance overhead. Run-time race detectors often rely on the happens-before vector clock algorithm for accuracy, yet suffer from either non-negligible performance overhead or low detection coverage due to a large amount of meta-data. Based on the observation that most of data races happen between close-by accesses, we introduce an optimization to selectively store meta-data only for recently shared memory locations and decouple meta-data storage from regular data storage such as caches. Experiments show that the proposed scheme enables run-time race detection with a minimal impact on performance (4.8% overhead on average) with very high detection coverage (over 99%). Furthermore, this architecture only adds a small amount of on-chip resources for race detection: a 13-KB buffer per core and a 1-bit tag per data cache block.

1. Introduction

Data race detection is widely used as a way to identify potential concurrency bugs in parallel programs due to unsynchronized memory accesses. Even though data races cannot detect all concurrency bugs, they provide a general condition to identify a broad range of bugs without application-specific knowledge. This paper presents an efficient algorithm and architecture that enable run-time data race detection with both high coverage and near-zero performance overhead. The proposed technique enables parallel programs to be continuously monitored for races even in production systems, which are extremely sensitive to run-time overhead.

Because checking data races purely in software can introduce substantial run-time overhead, several hardware-assisted techniques have been proposed [5, 18, 19, 22, 23, 30]. However, existing hardware techniques either show noticeable performance overhead or trade off detection coverage or scalability for lower overhead. For example, precise data race detection algorithms often depend on vector clocks [8, 27] to capture the happens-before relations [12] between memory accesses. While effective in accurately detecting data races, efficient and scalable hardware support for vector clocks is challenging because the size of vector clocks grows with the number of threads. For example, an early hardware vector clock scheme [23] could only support a small number of threads. The state-of-the-art vector clock scheme [5] provides good scalability with comprehensive detection coverage, but reports significant performance overhead at run-time (80% on average). Alternatively, a scheme based on scalar clocks was shown to have low overhead, but also a noticeably lower detection coverage of 77% [22].

This paper proposes a set of optimizations to selectively manage meta-data, which enable accurate race detection based on the happens-before relations in hardware with minimal performance overhead and without noticeably sacrificing the detection capability and scalability. The main optimization comes from the observation that only a small fraction of memory locations are accessed by multiple threads within a relatively short period where most data races happen. As a result, we found that storing meta-data only for those shared locations can greatly reduce the overhead with minimal impacts on coverage. While selectively maintaining vector clocks for statically shared memory locations has been proposed recently [5], we found that limiting the bookkeeping to locations that are *dynamically shared within a small window* is critical to achieve low overhead.

The proposed race detector only requires minor hardware changes with a small amount of state - a 13-KB buffer per core and a 1-bit tag per data cache block. Experimental results show that the selective bookkeeping does not significantly impact the race detection capability. In our experiments, the optimized detection scheme detected all 13 real-world data race bugs that we tested, and detected more than 99% of hundreds of data races that we injected to multi-threaded programs. Moreover, the experiments show that the proposed scheme has a minimal performance impact, with a 4.8% slowdown on average. In essence, the proposed scheme represents a new trade-off point between performance and coverage that was not possible before, making a deployment of continuous race detection in production systems feasible.

The rest of the paper is organized as follows. Section 2 presents a traditional (baseline) data race detection scheme based on vector clocks. Then, Section 3 describes how an accurate race detection can be efficiently realized in hardware

978-1-4799-3097-5/14/$31.00

with selective meta-data management along with architectural optimizations. Section 4 evaluates the effectiveness and overhead of the proposed race detector. Section 5 discusses related work, and Section 6 concludes the paper.

2. Data Race Detection Overview

While there are multiple approaches to detect data races, checking happens-before relations [12] is generally considered the most accurate technique in identifying data races. In this section, we provide an overview of data race detection based on happens-before relations, including the assumptions and intuitions behind the approach. We also describe a state-of-the-art race detection algorithm that uses vector clocks [8, 27] to capture the happens-before relations. This algorithm will be used as a baseline in this paper.

2.1. Assumptions

In this paper, we make a couple of assumptions that are common across many data race detection schemes, namely the shared memory programming model and identification of synchronization operations.

This work considers programs that are written under the shared memory programming model. Except for creating a thread and waiting for a termination, threads communicate through accesses to shared memory locations.

To distinguish data race bugs from legitimate synchronization operations, which often use races, we assume that synchronization operations can be explicitly identified. Programmers often rely on a library such as Pthreads to implement synchronization operations. In such cases, synchronization operations can be easily identified from the library calls. If a programmer uses custom synchronization primitives, our approach assumes that such primitives can be either explicitly marked or automatically identified. For example, previous studies showed that primitives such as spinlocks could be automatically detected [26, 28].

To be general, we describe synchronization operations using *release* and *acquire* instead of individual synchronization operations in the rest of the paper. While there exist many types of synchronization primitives, they can fundamentally be considered as acquiring and releasing tokens. For example, mutual exclusion requires for each thread to acquire a token (lock) before entering a critical section and releases a token after the critical section. Similarly, barrier synchronization can be realized by having each thread release its token after reaching a barrier and wait for acquiring tokens from all other threads before proceeding. In this paper, we refer to synchronization tokens as *synchronization objects*.

2.2. Data Races

A data race is defined as two conflicting memory accesses execute without any synchronization operation between them. Here, we define *conflicting accesses* as accesses from different threads to the same memory location, which include at least one write.

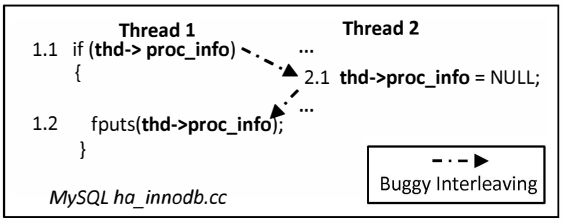


Figure 1. A data race bug in MySQL due to a missing critical section. The example is obtained from a previous study [13].

At run-time, data races can be accurately detected by checking if a pair of conflicting accesses is ordered by happens-before relations, which refer to an ordering between two events, in particular synchronization operations [12]. In other words, if a program is data race free, then every pair of conflicting accesses should be ordered by happens-before relations between synchronization operations [19].

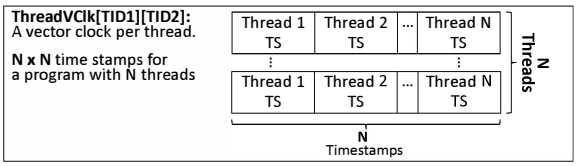
As an example, Figure 1 shows a data race in MySQL. In this example, none of the accesses to the shared pointer `thd->proc_info` is protected by synchronization. As a result, these accesses can execute in an arbitrary order, and potentially result in a fault if the pointer is set to be NULL by *2.1* between *1.1* and *1.2*. Here, there are two pairs of conflicting accesses, namely *1.1-2.1* and *1.2-2.1*. Data races can be detected as both conflicting access pairs are not ordered by happens-before relations. To fix the bug, both *1.1* and *1.2* need to be protected by a mutex lock, and *2.1* needs to be protected by the same lock to ensure an atomic execution.

2.3. Baseline Race Detection Algorithm

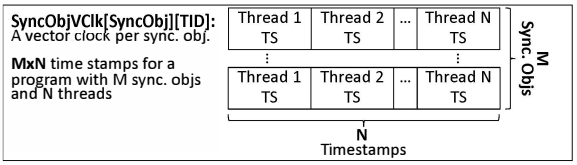
Here, we discuss a race detection algorithm based on vector clocks [8, 27]. We call this algorithm `RaceVC`, and use it as a baseline in the rest of this paper. Overall, `RaceVC` first identifies conflicting memory accesses, and checks if the conflicting accesses are ordered by happens before relations using vector clocks.

As shown in Figure 2, there are multiple vector clocks needed for the traditional `RaceVC` scheme. In the scheme, each thread is uniquely identified by a thread ID (`TID`). Vector clocks are used to encode the access history and happens-before relations among conflicting memory accesses and synchronization operations.

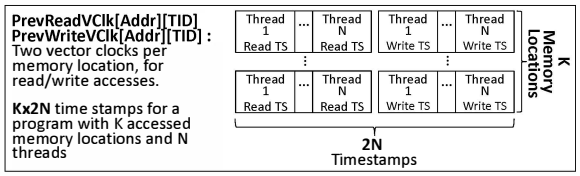
For a parallel program with `N` threads, each thread maintains a vector clock with `N` elements, as shown in Figure 2(a). Conceptually, elements in `ThreadVClk` encode the ordering constraint (i.e. happens-before relations) between two threads. For example, `ThreadVClk[i][j]` shows the earliest that a memory access from Thread `i` can be executed in terms of Thread `j`'s local time without violating the happens-before relations among synchronization operations. `ThreadVClk[i][i]` represents Thread `i`'s local clock that is incremented on each synchronization operation within the thread.



(a) Thread Vector Clock



(b) Sync. Obj. Vector Clock



(c) Memory Location Vector Clock

Figure 2. The meta-data required for baseline race detection algorithm (RaceVC). Each element in a vector clock records a time stamp (TS) for the associated thread.

The algorithm also maintains a vector clock for each synchronization object as shown in Figure 2(b). `SyncObjVClk` is used to encode the ordering constraints from each synchronization operation. On a release operation, the `SyncObjVClk` is updated with the `ThreadVClk` of the thread that performs the release (take the later timestamp for each element). The `SyncObjVClk` represents the earliest that the following acquire operation can happen in each thread's local time. On an acquire operation, a `ThreadVClk` is updated with the corresponding `SyncObjVClk`.

As shown in Figure 2(c), the algorithm uses `PrevReadVClk` and `PrevWriteVClk` to record timestamps for the most recent read and write from each thread to each memory location. The access timestamps are recorded based on each thread's local clock (i.e. `ThreadVClk[i][i]` for Thread i). If the vector clocks are properly maintained, one can check if the current memory access from Thread `i` and a previous access from Thread `j` are ordered by happens-before relations by comparing Thread `i`'s vector clock value `ThreadVClk[i][j]` with the timestamp of the previous access from Thread `j`. If the timestamp is greater or equal to `ThreadVClk[i][j]`, a data race is detected.

Figure 3 shows the detailed `RaceVC` algorithm. On a memory access, the algorithm first detects conflicting memory accesses, i.e. read-after-write, write-after-read, and write-after-write from multiple threads to the same memory location (`detect_conflict_access()`). Then, the algorithm determines if the conflicting access pair indicates

RaceVC Algorithm

Functions:

`detect_conflict_access(TID, Addr, Type, ThreadVClk[TID][TID])`

1. Check the most recent write in each thread:
   (a) For all valid threadID i $\neq$ `TID`, call
   `check_order(TID, i, PrevWriteVClk[Addr][i])`.
2. Check the most recent read in each thread:
   (a) If (`Type == Read`), skip Step 2.
   (b) For all valid threadID i $\neq$ `TID`, call
   `check_order(TID, i, PrevReadVClk[Addr][i])`.
3. Update the history for the memory location
   (a) If (`Type == Read`),
   `PrevReadVClk[Addr][TID] = ThreadVClk[TID][TID]`.
   (b) Otherwise,
   `PrevWriteVClk[Addr][TID] = ThreadVClk[TID][TID]`.

`check_order(TID, PrevTID, PrevTimeStamp)`

1. Check if the memory accesses can be re-orderd:
   If `ThreadVClk[TID][PrevTID]` $\leq$ `PrevTimeStamp`,
   report a data race.

`update_release(TID, SyncObj)`

1. `ThreadVClk[TID][TID]++`;
2. For each element in the vector clock, `SyncObjVClk[SyncObj][i] = MAX(ThreadVClk[TID][i], SyncObjVClk[SyncObj][i])`.

`update_acquire(TID, SyncObj)`

1. `ThreadVClk[TID][TID]++`;
2. For each element in the vector clock, `ThreadVClk[TID][i] = MAX(ThreadVClk[TID][i], SyncObjVClk[SyncObj][i])`.

Figure 3. RaceVC: Baseline algorithm.

a data race by checking whether the accesses are ordered by happens-before relations (`check_order()`). Lastly, the algorithm updates the associated memory location's vector clock based on each thread's local clock. On a synchronization *release* or *acquire* operation, `update_release()` or `update_acquire()` is called respectively to update vector clocks to encode the happens-before relations, and to increment the calling thread's local clock.

2.4. Challenges for Efficient HW Support

The main challenge in hardware support for data race detection lies in managing meta-data efficiently without significantly sacrificing scalability or detection coverage. A large amount of meta-data could result in large hardware structures or noticeable interference with regular program execution. On the other hand, reducing the amount of meta-data may limit the maximum number of threads that hardware can support or result in undetected races. In this context, traditional detection schemes based on vector clocks, such as `RaceVC`, are particularly challenging to support in hardware because they require vector clocks, whose size increases linearly with the number of threads, and for each memory location.

Specifically, as shown in Figure 2, `RaceVC` requires vector clocks for each thread, each synchronization object, and each memory location. The dominating portion of meta-data overhead comes from vector clocks for each memory location. This is because the number of accessed memory locations is typically significantly larger than the number of threads or the number of synchronization objects in a multithreaded program. Quite often, the size of vector clocks for threads and synchronization objects is negligible when

compared to the size of per-location vector clocks. Therefore, the main challenge is to efficiently manage meta-data for memory locations.

A recent algorithm, named FastTrack [9], showed that storing the last write per location, instead of a vector of writes (one from each thread), is enough for comprehensive data race detection. However, we note that even with a single clock for each write, the size of meta-data still increases linearly with the number of memory locations as we still need vector clocks for read operations. For the simplicity of presentation, we use vector clocks for both reads and writes in the `RaceVC` algorithm. `RaceVC` can be made to include the optimization by a simple change to only check the globally most recent write in Step 1 of Figure 3.

To reduce the overhead, previous proposals for happens-before data race detection in hardware store meta-data at a coarse granularity, often one or two vector clocks for each cache block [22, 23]. Also, these designs integrate the meta-data into data caches, adding storage for each cache block. Unfortunately, such integrated designs trade off flexibility and coverage for lower overheads. Ideally, the hardware support should have low overhead while allowing fine-grained bookkeeping to maintain high detection coverage.

3. HW-Assisted Race Detection

In this section, we describe an optimized race detection scheme, named `RaceSMM`, along with a hardware architecture support. The proposed optimizations are based on the insight that it is sufficient to maintain meta-data for a small number of recently shared memory locations. The design also decouples meta-data storage from caches and uses scalar meta-data to make the hardware scalable to a large number of threads.

3.1. Selective Bookkeeping

The main optimizations in our architecture design comes from the insight that the bookkeeping for race detection is only necessary for "shared" memory locations. Previous studies [10, 13, 15, 30] also observed that real-world race bugs typically manifest within a short window. Therefore, most real-world data races can be detected by maintaining meta-data for "shared" memory locations, which have conflicting accesses within a certain time period. Such shared memory locations are a fraction of the entire memory space, especially for a small window where most data races happen.

Table 1 shows the ratio of shared locations for various window sizes. Here, we define the window size by counting the total number of memory accesses (reads+writes) from all threads. The ratio is calculated by using the number of unique locations with conflicting accesses divided by the total number of unique locations accessed within a window.

For PARSEC and SPLASH2 benchmarks, less than 0.6% of memory locations have conflicting accesses that happen within a window of 100,000 memory accesses. In general,

Table 1. Percentage of shared locations in memory within various access window sizes. (P) - PARSEC, (S) - SPLASH2.

	1,000 Accesses	10,000 Accesses	100,000 Accesses	Entire Execution
Blackscholes(P)	0.000023%	0.00014%	0.00028%	40.27%
Bodytrack(P)	0.0030%	0.0059%	0.02%	69.79%
Fluidanimate(P)	0.0016%	0.014%	0.12%	26.70%
LU(S)	0.00021%	0.0030%	0.12%	99.28%
Ocean(S)	0.0020%	0.015%	0.11%	1.52%
Radix(S)	0.0023%	0.29%	0.60%	72.38%
Swaptions(P)	0.0012%	0.017%	0.22%	33.11%
Water-nsquare(S)	0.00013%	0.0040%	0.08%	42.53%
Water-spacial(S)	0.000087%	0.00079%	0.023%	57.43%
Geomean	0.00049%	0.0058%	0.05%	34.24%

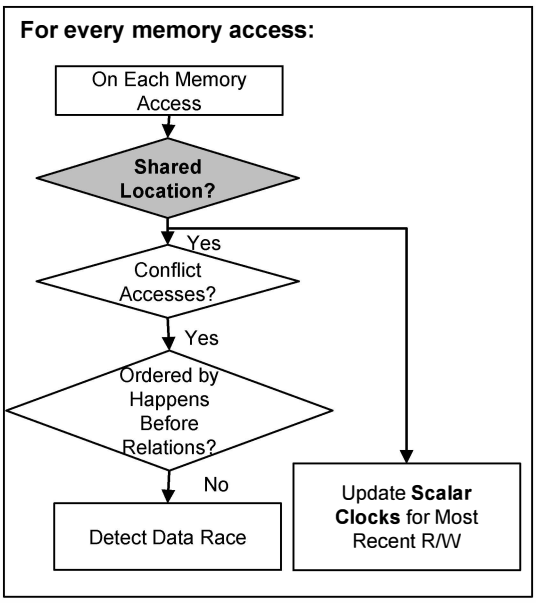


Figure 4. Flow chart for operations on memory accesses with the change (dark block) for selectively bookkeeping.

a 100,000 access window is big enough for the purpose of race detection as almost all data races happen within a much smaller window (1,000-10,000 window). This is true for all real-world experiments done in our evaluation and also concurs with previous studies [13, 15]. Recent studies [4, 11] have also confirmed that a significant percentage of the memory blocks are only accessed locally by one thread, even in parallel applications. Therefore, keeping meta-data such as timestamps and thread IDs (TIDs) for all memory locations is extremely wasteful. Instead, in our design, we decouple the detection of shared locations and the rest of bookkeeping so that most meta-data are stored only for memory locations with conflicting accesses.

The proposed design dynamically detects shared memory locations by augmenting each data cache block with a 1-bit tag, which indicates whether the block is shared between multiple threads or not, leveraging cache coherence events. The rest of the bookkeeping and detection are only performed

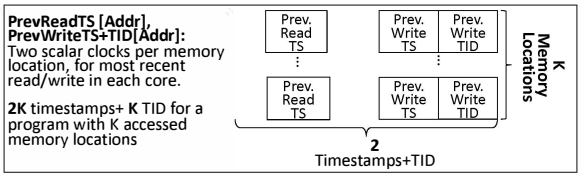


Figure 5. Per-location per-core meta-data for RaceSMM; only use scalar variables.

for those locations that are marked as shared (see Figure 4).

While RADISH [5] also discusses reducing meta-data by using a static analysis to identify memory locations that are never shared for the entire program execution, we found that limiting bookkeeping only to dynamically shared locations *within a time period* is critical to achieve low overhead. As shown in Table 1, the ratio of shared locations over the entire program execution is significant, several orders of magnitude higher than the one for a short period.

Based on the intuition that most data races happen within a relatively small window, we propose to selectively maintain meta-data only for memory locations that are dynamically detected to be shared while in on-chip caches As we will demonstrate in the evaluation section, the dynamic detection of shared locations and selective bookkeeping allow a much more efficient architecture while maintaining high detection coverage.

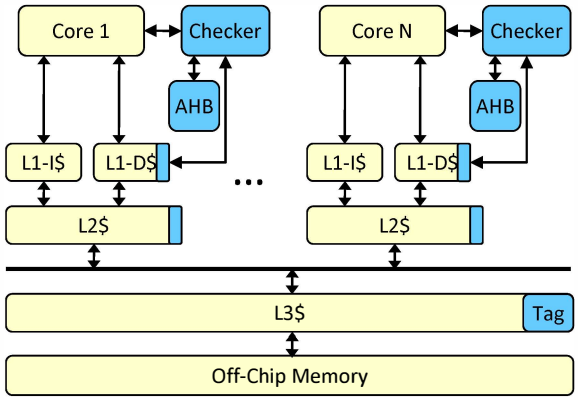
3.2. Distributed Scalar Clocks

Even with the selective bookkeeping, the vector clocks to track recent reads and writes, pose a significant challenge in building a scalable hardware-based race detector because their size increases linearly with the number of threads. While a previous work has shown that keeping information on only one write per location is sufficient [9], maintaining a vector clock per location for reads still poses a scalability challenge.

To address the challenge, `RaceSMM` stores scalar timestamps for writes and distributed scalar timestamps for reads for each memory location while using vector clocks for synchronization objects. The insight is that the read vector clocks can be maintained distributed across multiple cores so that only one scalar timestamp is stored in each core's meta-data buffer. Effectively, each core can keep a scalar timestamp for the most recent read access from the local thread and a scalar timestamp for the global most recent write access for each memory location.

As shown in Figure 5, in each core, `RaceSMM` only keeps track of timestamps (`PrevReadTS/PrevWriteTS`) and the write TID (`PrevWriteTID`) of the most recent read and write for each memory location. As each core keeps timestamps for reads for a local thread, we do not need to keep the read TID. Compared to `RaceVC`, the meta-data for each memory location no longer grows linearly with the number of threads.

On the other hand, in order to accurately capture the



Shared Location Detection: 1-bit shared tag for L1/2/3 cache block, set on L1/2 cache coherence downgrade event, propagate to L3.

Access History Buffer (AHB): Meta-data for most recent read/write to a shared location. Including PrevTIDs and PrevTSs.

Checker Module: Keeps active TID and ThreadVClk. On shared access from L1, records access to AHB, checks for data races. Bookkeeping ThreadVClk and SyncObjVClk on sync. operations.

Sync. Object Vector Clocks (not shown here): SyncObjVClks are stored and accessed through existing memory hierarchy.

Figure 6. A block diagram for the overall architecture. Blue (dark) blocks are additional hardware support needed for RaceSMM.

happens-before relations of synchronization objects and detect data races effectively, `RaceSMM` uses the same meta-data structures and bookkeeping operations as `RaceVC` for `ThreadVClk` and `SyncObjVClk`.

3.3. Architecture Support

Figure 6 shows the high-level block diagram for our architecture with support for data race detection. In the figure, the blue (dark) blocks indicate the new hardware components needed to support `RaceSMM`. The overall detection operations of our architecture closely follows the `RaceVC` algorithm, with the addition of using scalar timestamps in each core and selective bookkeeping for shared locations.

3.3.1. Extension for Shared Location Detection. In our architecture, each block in data caches has a 1-bit tag, which indicates whether the block is shared. The shared bit is set when a cache coherence event indicates that multiple cores access the same cache block with at least one write. More specifically, the shared bit is set when there is a downgrade request that changes the cache block to either shared or invalid coherence state. For example, in a MESI protocol, the shared bit is set with the following requests: `M->S, M->I, E->S, E->I, S->I, etc.` Greathouse et al. [10] have also found that the cache coherence events are indicative of data sharing between threads, and can be leveraged to drive

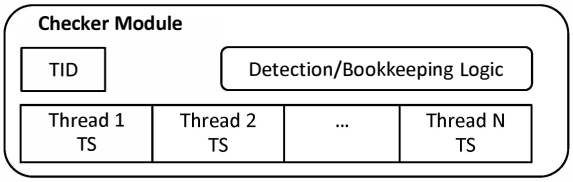


Figure 7. Checker: keeps current thread ID (TID), a thread vector clock (ThreadVClk), and detection/bookkeeping logic.

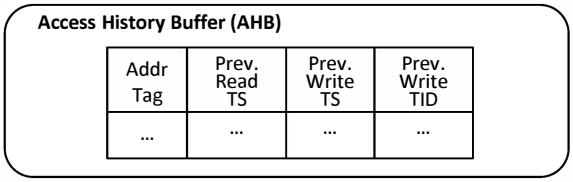


Figure 8. AHB stores meta-data for the most recent read/write to shared locations.

data race detection. This shared bit follows the data on-chip; a shared bit is written back to the lower cache levels, and the bit is read with data on a cache miss. However, the shared bit is cleared when a cache block is evicted to or read from off-chip memory. Effectively, this mechanism detects memory blocks that are *shared* within a time window, while the block exists in multiple private (L1/L2) caches, and keeps this history while the block is on-chip. While this design implies that we cannot detect shared locations with a large window between accesses from multiple threads, our experimental results suggest that this design, for cache sizes in modern processors, is sufficient to detect virtually all races tested.

The 1-bit tag may not detect locations that are shared by multiple threads on the same core or incorrectly identify a block as shared when a single thread moves from one core to another. However, we believe that the 1-bit tag is sufficient if each core runs one thread at a time with infrequent context switches or thread migrations. Our experiments show that even rather frequent context switches (every 1ms or even less) have negligible impact on detection coverage, and no false positives were encountered in our experiments.

3.3.2. Checker Module. In the proposed architecture, a checker module maintains per-thread state and performs most of the bookkeeping and checking operations at each core. As shown in Figure 7, for the active thread on the core, the checker keeps a thread ID (`TID`), and a thread vector clock (`ThreadVClk`). On a memory access from the core, the checker module uses the shared bit in an L1 data cache to determine if the access is to a "shared" block. If so, the checker records the access into the access history buffer (AHB), and examines whether there is a data race between the most recent read/write accesses and the current access.

On a local read/write access, the checker module checks if the access and the most recent write to the same location are ordered by comparing `ThreadVClk` with `PrevWriteTS` from the local AHB. To allow checking a race between remote read accesses and a local write, on a write to a shared location, the checker broadcasts the local thread's `ThreadVClk` to other checkers. Each remote checker then checks if the broadcasted write and the most recent read from its own thread are ordered by happens-before relations by comparing the broadcasted `ThreadVClk` with `PrevReadTS` from its own AHB (i.e. `check_order()`).

The access to local AHB can be done in parallel to local L1 accesses, and the `ThreadVClk` is kept locally within the checker module. Hence, the overhead of our detection mechanism is kept minimal on a read. On a write, however, broadcasting the write access and `ThreadVClk` can incur overheads. Fortunately, the broadcasting is needed only on writes to shared locations. For all benchmarks that we tested, writes to shared locations only account for 0.5% (3% worst case) of all memory accesses. Hence, the broadcasting overhead on writes is minimal in practice.

The checker module also coordinates with software layers through new instructions. The architecture provides two additional instructions to indicate a synchronization operation, one for *acquire* and the other for *release*. These instructions also convey the address of the vector clock for the corresponding synchronization object (`SyncObjVClk`). The vector clock for synchronization object is accessed and/or updated by the checker module on *acquire* and *release* operations through normal memory hierarchy.

3.3.3. Access History Buffer (AHB). The access history buffer (AHB) records information on the most recent read and write to a shared location that can be used to detect conflicting accesses and to check for data races. As shown in Figure 8, the AHB serves as a history table that saves `PrevReadTS`, `PrevWriteTS`, and `PrevWriteTID` for recently accessed shared locations. On a memory access to a shared location, the checker module records a thread ID (only for write accesses), a timestamp (TS), along with the memory address tag into the AHB.

As the AHB has a limited capacity, it works like a cache and only keeps the history of recently accessed shared memory locations. However, there is no backup hierarchy for the AHB. If an entry is evicted from an AHB, the information is simply thrown away. A miss to the AHB creates a new entry. While this design implies that we cannot detect conflicting accesses that are far apart, our experimental results suggest that an AHB with 1024 entries are sufficient for detecting virtually all races tested. Moreover, a miss to the AHB can only lead to potential false negatives, not false positives.

We note that the AHB can store an access history *per byte* because only accesses to shared memory locations are recorded. On the other hand, traditional designs that combine meta-data into the main cache often had to store information on a cache block granularity to keep overheads acceptable.

The AHB is kept coherent by a cache coherence protocol similar to other on-chip caches. We note that only the access history of a most recent write needs to be kept coherent. The access history of a most recent read is updated and accessed locally. We implemented the AHB coherence protocol separately from the main data cache for evaluation. As a future extension, the AHB coherence operations can be optimized by piggybacking on existing data cache coherence protocol, as we discuss later in the section.

3.3.4. Vector Clocks. Our proposed architecture uses two types of vector clocks: `ThreadVClk` and `SyncObjVClk`. For `ThreadVClk`, our architecture uses dedicated storage in each checker module for an active thread on the core. We found that `ThreadVClk` needs to be close to the checker because it is used in each `check_order()` operation. `ThreadVClk` needs to be treated as a part of thread state and managed by an operating system. For context switch or thread migration, the dormant thread's clock is saved in memory through OS support, and restored later. The size of `ThreadVClk` in each checker match the number of cores in a system.

On the other hand, `SyncObjVClk` is stored and accessed through the existing memory hierarchy. For each synchronization object, software allocates space for a vector clock in its memory space and passes the location using the instructions that indicate synchronization operations. On `update_release()` and `update_acquire()`, the `SyncObjVClk` is accessed and/or updated through the existing memory hierarchy. We note that `SyncObjVClk` needs to be accessed only on synchronization operations, which happen infrequently. As a result, `SyncObjVClk` accesses have a minimal impact on performance.

Hardware counters have a limited number of bits. As a result, the clock that each thread uses to represent its local time may overflow after many synchronization events. Fortunately, our experiments show that synchronization operations are rather infrequent and the thread clocks only increment slowly. In fact, we did not see any overflow for PARSEC and SPLASH2 benchmarks with 16-bit counters. Given that overflows are infrequent, our architecture handles them in a relatively slow but straightforward fashion instead of adding complex hardware. Upon detecting an overflow in its local clock, a checker raises an exception to an operating system, which in turn interrupts other cores that run other threads from the same program. Then, the operating system clears all clocks, and marks all AHB entries to be invalid in each core. In order to allow an operating system to clear vector clocks for synchronization objects, an application allocates them in separate pages that are known to the operating system.

3.4. AHB Optimizations

Our architecture includes a couple of optimizations for AHB, which are designed to improve detection coverage and reduce overhead. We also discuss how the AHB coherence operations can be optimized by piggybacking on existing data cache coherence protocol here.

3.4.1. Flexible Granularity. The AHB needs to store an access history *per byte* to avoid false positives from false sharing. However, for real-world applications, we noticed that only a very small fraction of memory accesses are at a byte granularity. Therefore, we implemented our AHB to support flexible granularity that defaults to word-granularity. For each AHB entry, there is a single-bit flag named word-granularity flag, which indicates if granularity is per byte (flag is false) or per word (flag is true). An additional 4-bits are appended to each entry to mark which bytes in a word are associate with the AHB entry. Accesses to all four bytes within a word are mapped to the same AHB entry; if the history is different for those bytes, only the most recent byte access history stays in the AHB. In this way, the architecture avoids using multiple AHB entries for a single word-granularity access while maintaining per-byte information to avoid false positive.

3.4.2. Prefetching and Selective Checking. Because shared memory locations are detected at a cache block granularity, one AHB access is likely to be followed by more AHB accesses to other bytes/words within the same cache block. We can exploit this locality and prefetch all corresponding AHB entries (write histories) when a cache line is loaded to a private data cache.

We can also reduce the number of write broadcasts exploiting the coherence state of a cache line. If a cache line stays in M state and the write TID for the local AHB entry matches the local thread ID, it implies that a write has been already broadcasted and there was no remote write to the cache line after that. In that case, broadcasts for additional writes to the same location can be avoided until there is a downgrade request for the cache line.

Similarly, checks on reads to a cache line that remains in S/E state can be avoided with a minimal impact on coverage if there is no valid AHB entry. The S/E state indicates that there was no remote write to the cache line after the corresponding AHB entries were fetched and checked on a cache miss. In that case, additional data races are unlikely until the cache line state is downgraded by a write from a remote thread. We note that there is a corner case for this optimization. For example, when there is write X from Thread 1 followed by read Y and read X from Thread 2, the race in X can be missed if X and Y both belong to the same cache line. However, our experiments suggest that this optimization has virtually no impact on the static race coverage in practice because multiple dynamic race occurrences will happen for each static race during a program execution, and our scheme always detect at least one dynamic instance of each static race in all tests.

Overall, the above optimizations are effective in reducing the number and the latency of AHB checks, which may

Table 2. Baseline architecture parameters.

Component	Parameters
Core	4 2-GHz in-order single-issue cores
Caches	L1 I/D (private, inclusive): 32KB/32KB 4-ways 3 cycles Latency L2 (private): 256KB, 4-ways 15 cycles latency L3 (shared): 8MB, 8-ways 40 cycles latency
Coh. protocol	MESI
DRAM	50ns Latency
Meta-data	8-bit thread IDs, 16-bit clocks
AHB	1024-entries, 8-way, 13KB, 3 cycles latency
On-chip Network	1GHz 32-bit bus, 2 bus cycles routing time 2 bus cycles per hop, 2 hops communication delay,

require sending write histories between AHBs. The optimizations are particularly useful if a memory read is delayed until the race check is done, which is the case for our evaluation.

3.4.3. Piggybacking AHB Coherence Messages. As a future extension, most of the AHB coherence operations can piggyback on existing data cache coherence protocol. Specifically, when a cache line is loaded into a private data cache on a write, the broadcasting of the local thread's `ThreadVClk` and `ThreadID` can piggyback on cache invalidation messages. Similarly on a read, the prefetching of remote AHB entries can piggyback on a bus upgrade request to and acknowledgments from remote caches. AHBs need to send additional messages for keeping the write histories coherent. However, we estimate that such AHB-only messages will be needed only for less than 1% of memory accesses after applying the optimizations in this subsection.

4. Evaluation

4.1. Evaluation Setup

Our infrastructure is built on the Pin binary instrumentation framework [16]. To evaluate the detection capability, our tool implements both the baseline `RaceVC` and `RaceSMM` algorithms by intercepting memory accesses and Pthread calls. To evaluate the performance overhead of `RaceSMM`, we implemented a typical memory hierarchy with bookkeeping structures in a Pin tool, and also added a timing model with an in-order core that runs 1 instruction per cycle, L1/L2/L3 caches, and a memory interface.

Table 2 summarizes the baseline architecture parameters. We model a processor with 4 cores, 64KB L1 and 256KB L2 private caches per core, and an 8MB shared L3 cache. We also model the MESI cache coherence protocol. For the additional bookkeeping, we model the AHB with an 8-bit thread ID and a 16-bit timestamp per read/write access. For cache and AHB coherence, we model an on-chip communication network with 1GHz 32-bit wide buses, 2 bus cycles routing time, 2 bus cycles per hop, and 2 hops for on-chip communication delay.

To evaluate the detection capability, we use two types of benchmarks, namely kernel bugs (KB) and real programs. The kernel bugs are created based on real-world application race bugs (MySQL, Apache, and Mozilla) from previous studies [13, 14, 29]. We also use three data race bugs from two large real-world server applications (Apache and MySQL). We use 30 threads for Apache and 10 threads for MySQL. For a further study on coverage, we also perform random race injections to benchmarks from SPLASH2 [3] and PARSEC [1]. The race injection is performed by randomly selecting a critical section and ignoring the critical section for the entire program execution. The SPLASH2 and PARSEC benchmarks were run using 4 threads with the default input size for SPLASH2 benchmarks and `simmedium` input size for PARSEC.

Table 3. Detection capabilities.

	RaceVC SW	RaceSMM SW	RaceSMM HW
Apache	**Yes**	**Yes**	**Yes**
MySQL-1	**Yes**	**Yes**	**Yes**
MySQL-2	**Yes**	**Yes**	**Yes**
KB1(MySQL)	**Yes**	**Yes**	**Yes**
KB2(MySQL)	**Yes**	**Yes**	**Yes**
KB3(MySQL)	**Yes**	**Yes**	**Yes**
KB4(Apache)	**Yes**	**Yes**	**Yes**
KB5(Mozilla)	**Yes**	**Yes**	**Yes**
KB6(Mozilla)	**Yes**	**Yes**	**Yes**
KB7(Mozilla)	**Yes**	**Yes**	**Yes**
KB8(Mozilla)	**Yes**	**Yes**	**Yes**
KB9(Mozilla)	**Yes**	**Yes**	**Yes**
KB10(Mozilla)	**Yes**	**Yes**	**Yes**

4.2. Race Detection Capability

We compare the race detection capabilities of two schemes: `RaceVC` and `RaceSMM`. `RaceVC` is implemented only in software, as we only need to use its detection capability as a baseline. `RaceSMM` is implemented in both software (`RaceSMM-SW`) and hardware (`RaceSMM-HW`). `RaceSMM-SW` keeps a vector clock for the most recent reads and only a scalar timestamp for the most recent write for each byte in memory, whereas `RaceSMM-HW` relies on caches and AHBs, both with limited capacities, for bookkeeping. We note that as suggested in previous study [9], `RaceSMM-SW` has the same coverage for static races as `RaceVC`. Hence, for the rest of this section, we only discuss results for `RaceVC` and `RaceSMM-HW`. We also use `RaceSMM` to denote `RaceSMM-HW` implementation in the rest of the paper.

4.2.1. Detection Coverage. Here, we present the static race detection coverage results after a single run of each program. We believe that the usefulness of a race detection scheme is mostly dependent on its static detection rate, as suggested in previous studies [22, 23]. This is because as long as one dynamic instance of a data race is detected, the data race is effectively exposed and can then be subsequently fixed if necessary. While not shown here due to space constraints, on average `RaceSMM` detects 60% of dynamic race occurrences.

Table 3 shows detection results for real-world data race bugs. The results show that both `RaceVC` and `RaceSMM` detect all races, indicating our hardware approach do not significantly affect detection coverage.

In theory, `RaceVC` may be able to detect data races that `RaceSMM` cannot. This is because the hardware implementation of `RaceSMM` relies on caches to detect shared locations

Table 4. Race injection study results. 50 races were randomly injected into the benchmarks.(P) - PARSEC, (S) - SPLASH2.

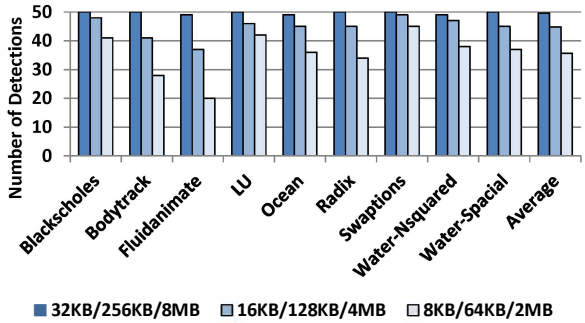
	RaceVC	RaceSMM
Blackscholes (P)	50	50
Bodytrack (P)	50	50
Fluidanimate (P)	50	49
LU (S)	50	50
Ocean (S)	50	49
Radix (S)	50	50
Swaptions (P)	50	50
Water-Nsquared (S)	50	49
Water-sSpacial (S)	50	50
Total	450/450	447/450

and AHBs to keep information on recent accesses. Because the caches and AHBs have limited capacities, relevant information may be evicted, causing potential false negatives. As shown in Table 4, `RaceSMM` did not detect 3 data races in our race injection study whereas `RaceVC` detected all.

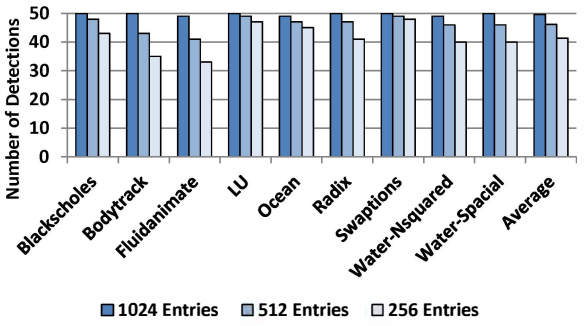
Overall, results for both real-world race bugs in Table 3 and injected races in Table 4 suggest that `RaceSMM`'s detection coverage is comparable to that of `RaceVC`. `RaceSMM` detected more than 99% of the injected races. Previous studies [13, 15] observed that real-world race bugs typically manifest within a short window. This explains why the hardware implementation with limited bookkeeping shows comparable detection coverage to the software implementation.

`RaceSMM` signals a race when two conflicting (same location, at least one write) memory accesses are not ordered by happens-before relations, which is precisely the definition of a data race. While it is possible to have false negatives due to limited bookkeeping, in theory, `RaceSMM` would never flag a pair of conflicting accesses when a data race does not exist. However, a false positive may happen in practice when aggressive thread switching happens. In such case, a memory location accessed by a single thread can be mistakenly identified as shared because a thread moves from one core to another. To eliminate the false positives, a thread ID can be added to the cache line in order to identify which thread each access comes from. The overhead would still be quite low as only one thread ID needs to be maintained per cache block. In our study, both the software and hardware implementations of `RaceSMM` report no false positives for Apache, MySQL, SPLASH2, and PARSEC applications.

We have also studied the impact of context switches on the detection coverage of our proposed scheme. Each core's AHB was periodically cleared to mimic the effect of a context switch on a core. Overall, the impact of context switches on detection coverage is negligible. For example, if a context switch occurs every 1ms on each core, our proposed architecture would still detect virtually all data races tested (coverage remains at 99%). In our experiments, the 1024-entry AHBs are filled within 0.01ms on average and 0.04ms in the worst case, which is a fraction of typical time quanta. Also, a data race is often detected many times over an execution; so



(a) Caches size study.



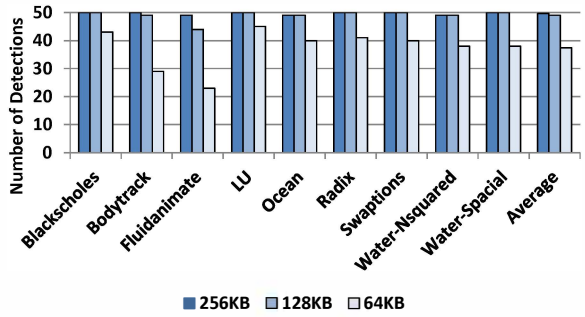
(b) AHB size study.

Figure 9. The impact of cache and AHB sizes on the detection capability of RaceSMM. We injected 50 races to each configuration. We reduced L1, L2, L3 and AHB sizes to 1/2 and 1/4 of the baseline configurations (32KB, 256KB, 8MB, 1024-entries).

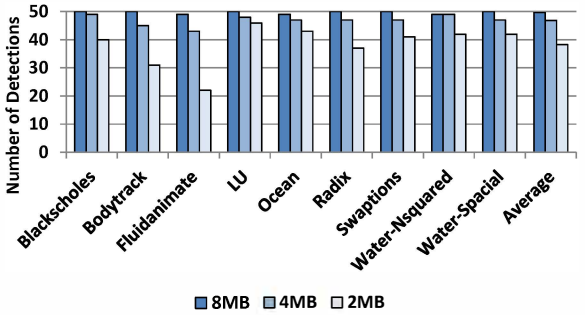
missing one dynamic instance does not necessarily lead to a lower static coverage.

4.2.2. Cache and AHB Size Analysis. The hardware-based `RaceSMM` scheme relies on caches and the AHB for bookkeeping. Therefore, the cache and AHB sizes directly affect the detection capability. The race injection study in Figure 9(a) shows the impact of reducing cache sizes on the detection coverage. Here, the L1, L2, and L3 caches are reduced to 1/2 and 1/4 of the baseline while keeping the AHB at the baseline size. As expected, the detection rate decreases as the cache sizes decrease. Similarly, Figure 9(b) shows the impact of reducing the AHB size. The coverage decreases as the AHB size decreases because smaller AHBs can only keep history for less memory locations. The exact impact of reduced cache and AHB sizes, however, depends on application characteristics. For example, memory intensive benchmarks such as `Fluidanimate` are more sensitive than others. Overall, the experiments indicate that our scheme needs a private (L2) cache of 128-KB, a last level cache (L3) of 4-MB, and an AHB with 512 entries in order to provide good coverage (around 90%).

We have also varied individual cache sizes and evaluated their effects on detection capability, as shown in Figure 10.



(a) L2 cache size study.



(b) L3 cache size study.

Figure 10. The impact of L2/L3 cache sizes on the detection capability of RaceSMM. We injected 50 races to each configuration. We reduced L2 and L3 cache sizes separately to 1/2 and 1/4 of their baseline configurations (256KB L2, 8MB L3).

We note that L1 cache is implemented to be inclusive, and hence its size does not impact race detection capability. Overall, we found that both L2 and L3 cache sizes have comparable impacts on detection capability, though for different underlying reasons. As we set the shared bits in the L2 cache on cache coherence downgrade requests, the L2 cache size impacts the probability of a shared location being detected. On the other hand, the L3 cache stores the shared bits while the shared cache blocks remain on-chip, and its size has a direct impact on how long a shared location's shared bit remains set.

We note that for all configurations shown in Figure 9 and 10, `RaceSMM` detected all real-world bugs in Table 3.

4.3. Performance and Power Overheads

The software implementation of `RaceSMM` incurs a 10-20X performance slowdown compared to the plain Pin execution of the programs. The overhead excludes the standard overhead of Pin, which has 12X overhead on average. The current software implementations are not optimized, and we expect the overheads to be lower with further optimization.

Figure 11 shows the normalized execution time for `RaceSMM`. The performance overhead is low at 4.8% on average. In the worst case, the overhead is 12.5% for

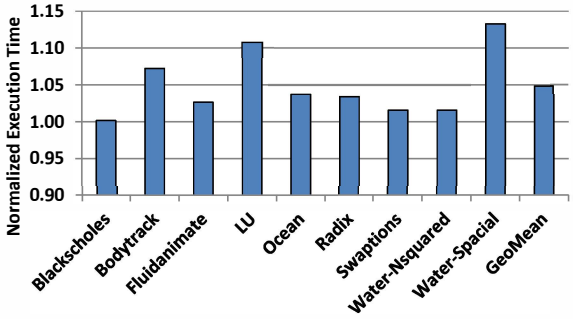


Figure 11. Performance overhead normalized to native execution.

`Water-Spacial`. Because the architecture mostly uses dedicated on-chip structures such as 1-bit cache tags and the AHB for bookkeeping, the only major source of performance overhead comes from communications between AHBs for keeping write history coherent, which is responsible for almost all of the performance overhead. We note that the current communication overhead is conservative as we modeled the communication network bus only with 32-bit width and at 1 GHz instead of the more aggressive configurations. The communication and check latencies can also be hidden if checks are performed in a background while the program execution continues instead of having each memory access wait for a check.

Another possible source of overhead comes from accessing vector clocks for synchronization objects through the normal memory hierarchy. However, the number of vector clock accesses are negligible when compared to the number of regular data accesses. The vector clocks only introduce 0.08% additional accesses for `Fluidanimate` in the worst case, and only 0.003% on average. Hence, the overall performance impact due to additional vector clock accesses is negligible. The average L1 cache miss rate only increases 0.03% compared to the baseline.

Counter overflows may also introduce performance overhead by requiring timestamps and vector clocks to reset. However, we have never encountered any overflow during experiments. In the worst case, in `Fluidanimate`, we observed the maximum timestamp value of 47,832 after 287,748,471 memory accesses when the benchmark finishes. In all other benchmarks, the timestamp values never exceeded 10,000 after the entire execution. Hence, we believe that the possible performance overhead introduced by timestamp overflows is negligible.

In terms of the area and power consumption, the AHBs are likely to be the main source of overhead. The dynamic power consumption of AHBs based on CACTI [17] is estimated to be 56mW on average.

Table 5. Comparison of HW-assisted data race detection schemes.

	ReEnact [23]	CORD [22]	RADISH [5]	RaceSMM
Scalable	No	Yes	Yes	**Yes**
Detection Coverage	High	77%	100%	**over 99%**
False Positive	Yes	No	No	**No**
Hardware Overhead	High	High	Low	**Low**
Average Performance Overhead	5.8%	0.4%	80%	**4.8%**
Worst Case Performance Overhead	14.7%	3%	200%	**12.5%**

4.4. Comparison to Related Schemes

The proposed `RaceSMM` scheme is most closely related to other hardware supported data race detection techniques based on happens-before relations. As shown in Table 5, we compare the characteristics of three other hardware assisted data race detection schemes with ours.

ReEnact [23] provides hardware support for logical vector clocks for cache lines. Due to its high hardware complexity and the use of vector clocks for cache lines, ReEnact suffers from noticeable performance overhead, and poor scalability for more than a few threads. While ReEnact provides high detection coverage (note that no quantitative detection coverage is available from [23]), it also suffers from false positives due to false sharing of vector clocks within a cache line. The hardware overhead for ReEnact is also high as it requires specialized registers for vector clock storage in each cache, and the register size increases linearly as the number of threads grows.

CORD [22] avoids the overheads and poor scalability issue of vector clocks by keeping four scalar timestamps per cache line, at the expense of lower detection coverage (77%). CORD has high hardware overhead as it requires on-chip state equal to 19% of cache capacity. In comparison, `RaceSMM` only requires 13-KB on-chip buffer space with 1-bit tag per cache line. Overall, CORD provides a scalable data race detection scheme with very low performance overhead and no false positive at the cost of lower detection coverage and high hardware overhead.

RADISH [5] proposes a hardware and software hybrid race detection scheme that uses a vector clock based race detection approach similar to FastTrack [9]. While it provides good scalability and comprehensive detection coverage, it still incurs significant performance overhead at run-time compared with our proposed scheme. RaceSMM shows a 4.8% average slowdown and RADISH reports an 80% average slowdown in execution. Comparing with RADISH, the performance improvement in RaceSMM comes from that it only maintains meta-data for *dynamically shared locations within a small window* compared to statically shared locations in RADISH. Also, RaceSMM only stores meta-data while in AHBs whereas RADISH keeps them as data in caches and memory, requiring extra transfers in the memory hierarchy.

Overall, `RaceSMM` provides a scalable race detection scheme with high detection coverage and no false positives, and requires low hardware overhead through separate on-chip only buffers (AHBs). In that sense, `RaceSMM` represents a new trade-off point between performance and detection coverage compared to existing race detection schemes.

5. Related Work

Data Race Detection: At a high-level, data race detection techniques can be categorized into static and dynamic approaches. Static race detection schemes such as RacerX [7] use static analysis to detect possible data races. However, static approaches are generally conservative without run-time information, resulting in false positives, and usually require source code.

Dynamic data race detection techniques fall into two main classes, namely lockset based and happens-before based. The lockset approach, such as Eraser [25], checks whether each shared variable is protected by at least one lock. The happens-before approach checks whether two memory accesses are explicitly synchronized [12]. There are many previous proposals that fall into the happens-before category, including RecPlay [24], ReEnact [23], CORD [22], FastTrack [9], Light64 [19], PACER [2], RADISH [5], and others. In general, the happens-before approaches are more accurate than the lockset approaches but often have higher overhead. Researchers have also investigated hybrid approaches in order to reduce the overhead of happens-before algorithms while maintaining a low false positive rate [6, 20, 21].

The proposed architecture can be considered as an extension of the happens-before approach to detect races at run-time. However, our hardware architecture shows that the happens-before race detection approach can be realized in hardware with minimal performance overhead and with minimal impact on detection coverage.

Hardware-Based Race Detection: While many race detection techniques can be enhanced with architecture support, this work is most closely related to happens-before race detection. In this context, ReEnact [23], CORD [22], and RADISH [5] are the most related detection schemes to our work, which are discussed in detail in Section 4.4.

As an alternative to the happens-before approach, researchers have also presented simple hardware support for race detection relying on other heuristics. For example, HARD [30] uses lockset and SigRace [18] uses hash signatures from Bloom filters to detect possible data races. These approaches enable reasonable race detection coverage with minimal hardware additions. However, generally, they trade off accuracy and coverage for the simplicity. In this work, we showed that accurate happens-before race detection can also be realized with relatively simple hardware support.

6. Conclusion

This paper proposes an efficient hardware architecture that enables run-time data race detection with high coverage and minimal performance overhead. In particular, The paper proposes an architectural optimization that decouples meta-data storage from regular caches so that expensive meta-data is only selectively stored for dynamically shared memory locations within a small window. Experimental results show that this hardware-assisted race detection scheme provides high detection coverage (over 99%) with no false positives, while keeping its overhead minimal. Overall, our paper provides an attractive way to detect data races at run-time.

7. Acknowledgment

This work was partially supported by the National Science Foundation under grants CNS-0746913 and CCF-0905208, the Air Force Office of Scientific Research under Grant FA9550-09-1-0131, the Office of Naval Research under grant N00014-11-1-0110, and an equipment donation from Intel Corporation. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of NSF, ONR, AFOSR, or Intel.

References

[1] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. Technical Report TR-811-08, Princeton University, January 2008.

[2] M. D. Bond, K. E. Coons, and K. S. McKinley. PACER: proportional detection of data races. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2010.

[3] CAPSL. The modified SPLASH-2. http://www.capsl.udel.edu/splash/, July 2007.

[4] B. A. Cuesta, A. Ros, M. E. Gómez, A. Robles, and J. F. Duato. Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks. In *Proceedings of the* 38^{th} *Annual International Symposium on Computer Architecture*, 2011.

[5] J. Deviett, B. P. Wood, K. Strauss, L. Ceze, D. Grossman, and S. Qadeer. RADISH: always-on sound and complete race detection in software and hardware. In *Proceedings of the* 39^{th} *Annual International Symposium on Computer Architecture*, 2012.

[6] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: a race and transaction-aware Java runtime. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007.

[7] D. Engler and K. Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *Proceedings of the* 19^{th} *ACM Symposium on Operating Systems Principles*, 2003.

[8] C. Fidge. Logical time in distributed computing systems. *IEEE Computer*, 24:28–33, August 1991.

[9] C. Flanagan and S. N. Freund. FastTrack: efficient and precise dynamic race detection. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[10] J. L. Greathouse, Z. Ma, M. I. Frank, R. Peri, and T. Austin. Demand-driven software race detection using hardware performance counters. In *Proceedings of the* 38^{th} *Annual International Symposium on Computer Architecture*, 2011.

[11] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Reactive NUCA: near-optimal block placement and replication in distributed caches. In *Proceedings of the* 36^{th} *Annual International Symposium on Computer Architecture*, 2009.

[12] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21:558–565, July 1978.

[13] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the* 13^{th} *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[14] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: detecting atomicity violations via access interleaving invariants. In *Proceedings of the* 12^{th} *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[15] B. Lucia, J. Devietti, K. Strauss, and L. Ceze. Atom-Aid: detecting and surviving atomicity violations. In *Proceedings of the* 35^{th} *Annual International Symposium on Computer Architecture*, 2008.

[16] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the* 2005 *Conference on Programming Language Design and Implementation International*, 2005.

[17] N. Muralimanohar and R. Balasubramonian. CACTI 5.3: A tool to understand large caches.

[18] A. Muzahid, D. Suárez, S. Qi, and J. Torrellas. SigRace: signature-based data race detection. In *Proceedings of the* 36^{th} *Annual International Symposium on Computer Architecture*, 2009.

[19] A. Nistor, D. Marinov, and J. Torrellas. Light64: lightweight hardware support for data race detection during systematic testing of parallel programs. In *Proceedings of the* 42^{nd} *Annual IEEE/ACM International Symposium on Microarchitecture*, 2009.

[20] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *Proceedings of the* 9^{th} *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2003.

[21] E. Pozniansky and A. Schuster. Efficient on-the-fly data race detection in multithreaded C++ programs. In *Proceedings of the* 9^{th} *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2003.

[22] M. Prvulovic. CORD: cost-effective (and nearly overhead-free) order-recording and data race detection. In *Proceedings of the* 12^{th} *Annual International Symposium on High-Performance Computer Architecture*, 2006.

[23] M. Prvulovic and J. Torrellas. ReEnact: using thread-level speculation mechanisms to debug data races in multithreaded codes. In *Proceedings of the* 30^{th} *Annual International Symposium on Computer Architecture*, 2003.

[24] M. Ronsse and K. De Bosschere. RecPlay: a fully integrated practical record/replay system. *ACM Transactions on Computer Systems*, 17:133–152, May 1999.

[25] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15:391–411, November 1997.

[26] C. Tian, V. Nagarajan, R. Gupta, and S. Tallam. Dynamic recognition of synchronization operations for improved data race detection. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, 2008.

[27] C. Valot. Characterizing the accuracy of distributed timestamps. In *Proceedings of the 1993 ACM/ONR Workshop on Parallel and Distributed Debugging*, 1993.

[28] W. Xiong, S. Park, J. Zhang, Y. Zhou, and Z. Ma. Ad hoc synchronization considered harmful. In *Proceedings of the* 9^{th} *USENIX Conference on Operating Systems Design and Implementation*, 2010.

[29] J. Yu and S. Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. In *Proceedings of the* 36^{th} *Annual International Symposium on Computer Architecture*, 2009.

[30] P. Zhou, R. Teodorescu, and Y. Zhou. HARD: hardware-assisted lockset-based race detection. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, 2007.