SigRace: Signature-Based Data Race Detection

Abdullah Muzahid
University of Illinois at
Urbana-Champaign, USA
muzahid2@illinois.edu

Darío Suárez
Universidad de Zaragoza
Zaragoza, Spain
dario@unizar.es

Shanxiang Qi
University of Illinois at
Urbana-Champaign, USA
sqi2@illinois.edu

Josep Torrellas
University of Illinois at
Urbana-Champaign, USA
torrella@illinois.edu

ABSTRACT

Detecting data races in parallel programs is important for both software development and production-run diagnosis. Recently, there have been several proposals for hardware-assisted data race detection. Such proposals typically modify the L1 cache and cache coherence protocol messages, and largely lose their capability when lines get displaced or invalidated from the cache. To avoid these shortcomings, this paper proposes a novel approach to hardware-assisted data race detection. The approach, called *SigRace*, relies on hardware address signatures. As a processor runs, the addresses of the data that it accesses are automatically encoded in signatures. At certain times, the signatures are automatically passed to a hardware module that intersects them with those of other processors. If the intersection is not null, a data race may have occurred.

This paper presents the architecture of SigRace, an implementation, and its software interface. With SigRace, caches and coherence protocol messages are unmodified. Moreover, cache lines can be displaced and invalidated with no effect. Our experiments show that SigRace is significantly more effective than a state-of-the-art conventional hardware-assisted race detector. SigRace finds on average 29% more static races and 107% more dynamic races. Moreover, if we inject data races, SigRace finds 150% more static races than the conventional scheme.

Categories and Subject Descriptors

B [**Hardware**]: B.3 Memory Structures,B.3.2 Design Styles. **Subjects:** Shared memory; B.3.4 [**Reliability, Testing, and Fault-Tolerance**]: Error checking.

General Terms

Design, Measurement, Reliability.

Keywords

SigRace, Signature, Timestamp, Data Race, Concurrency Defect, Happened-Before.

ISCA'09, June 20–24, 2009, Austin, Texas, USA.
Copyright 2009 ACM 978-1-60558-526-0/09/06 ...$5.00.

1. INTRODUCTION

With the widespread use of multicore hardware, parallel programming is likely to become more prevalent. At the same time, concurrency bugs are likely to take on a higher profile and become a very costly problem. Consequently, it is crucial to continue developing more effective techniques to detect and fix them.

An important type of concurrency bug is a data race. A data race occurs when two threads access the same variable without an intervening synchronization and at least one of the accesses is a write. The erroneous program behavior caused by the race may only appear under certain access interleavings, making debugging data races notoriously hard.

For this reason, data race detection has been the subject of much work (e.g., [5, 8, 12, 14, 15, 16, 17, 18, 19, 22, 24, 26, 27, 29, 30]), including the development of commercial software tools for race debugging (e.g., [8, 26]) and even the proposal of special hardware structures in the machine (e.g., [12, 18, 19, 30]). In general, there are two approaches to finding data races, namely the lockset approach, as in Eraser [24], and the happened-before one, as in Thread Checker [8]. The lockset approach is based on the idea that all accesses to a given shared variable should be protected by a common set of locks. Consequently, it tracks the set of locks held while accessing each variable. It reports a violation when the currently-held set of locks (lockset) at two different accesses to the same variable have a null intersection.

The happened-before approach relies on epochs. An epoch is a thread's execution between two consecutive synchronization operations. Each processor has a logical clock, which identifies the epoch that the processor is currently executing. In addition, each variable has a timestamp, which records at which epoch the processor accessed it. When another processor accesses the variable, it compares the variable's timestamp to its own clock, to determine the relationship between the two corresponding epochs: either one logically happened before the other, or the two logically overlap. In the latter case, we have a race.

Race detectors that use these algorithms in software typically induce about 10–50x slowdowns on programs [8, 14, 22, 24]. Such slowdowns can distort the timing of races identified in production runs, and make them hard to find. For this reason, there have been several recent proposals for race detectors with hardware assists [12, 18, 19, 30]. Such schemes should be effective at debugging races in production runs. However, they detect races by augmenting the cache state and the coherence protocol. Specifically, they tag each cache line with a timestamp [12, 18, 19] or a lockset [30], perform additional operations on local/external access to the cache, and piggyback information on cache coherence protocol messages. L1 caches and coherence protocol units are key hardware structures, either time-critical or complicated. In addition, if a line is displaced or invalidated from the cache, these systems typically lose the ability to detect races involving the line.

This paper proposes a novel approach to hardware-assisted data race detection that overcomes these limitations. Our approach, called *SigRace*, relies on hardware address signatures. As a processor runs, the addresses of the data that it accesses are automatically encoded in signatures. At certain times, the signatures are automatically passed to a hardware module that intersects them to those of other processors. If the intersection is not null, a data race may have occurred. With SigRace, there are no changes to the cache or the cache coherence protocol messages, and there are no critical-path operations performed on local/external access to the cache. Moreover, lines can be displaced or invalidated from caches without affecting SigRace's ability to detect data races.

This paper presents the architecture of SigRace, an implementation, and its software interface. Application code is unmodified. Our experiments show that SigRace is significantly more effective than a state-of-the-art conventional hardware-assisted race detector. SigRace finds, on average, 29% more static races and 107% more dynamic races. Moreover, if we inject data races, SigRace finds 150% more static races than the conventional scheme.

This paper is organized as follows: Section 2 gives a background; Sections 3 and 4 describe the SigRace architecture and implementation; Section 5 evaluates SigRace; and Section 6 concludes.

2. BACKGROUND

2.1 Logical Timestamps for Happened-Before

Lamport's happened-before relation [9] in a multithreaded environment states that an event α happened before another β if (i) both are performed by the same thread and α precedes β in program order, or (ii) α is a release and β is an acquire on the same object, or (iii) for some other event γ, α happened before γ and γ happened before β. If α happened before β or vice-versa, the two events are ordered; otherwise, they are concurrent or unordered. The happened-before algorithm for race detection finds out whether two memory accesses to the same location that are performed by different threads are unordered and at least one is a write. This algorithm only detects races that actually occur during execution.

In a typical implementation, each thread maintains a logical vector clock, which has as many components as number of threads [7]. If thread *t* has a vector clock $vc_t[.]$, then the element $vc_t[t]$ contains the time of the thread itself and, given another thread *u*, $vc_t[u]$ contains the latest time of *u* "known" to *t*. When *t* performs a synchronization operation, it starts a new *Epoch* and increments $vc_t[t]$. Suppose that, after *t* performed a release on object *S*, *u* acquires *S*. In this case, *u* increments $vc_u[u]$ and, in addition, updates the rest of $vc_u[.]$ as follows: $vc_u[i] = max(vc_u[i], vc_t[i])$ for every $i \neq u$. Here, $vc_t[.]$ is the vector clock of thread *t after* the release operation. We refer to the value of a thread's vector clock during an epoch as the epoch's *Timestamp*. Figure 1(a) shows an example execution with epoch timestamps.

We determine whether there is a happened-before relation between two epochs by comparing their timestamps. Specifically, if epoch *f* of thread *t* has timestamp $vc_t^f[.]$ and epoch *g* of thread *u* has timestamp $vc_u^g[.]$, then *f* happened before *g* if and only if $vc_t^f[t] < vc_u^g[t]$ and $vc_t^f[u] < vc_u^g[u]$. For example, in Figure 1(a), the epoch after the acquire in Thread 2 happened before the epoch after the second acquire in Thread 0.

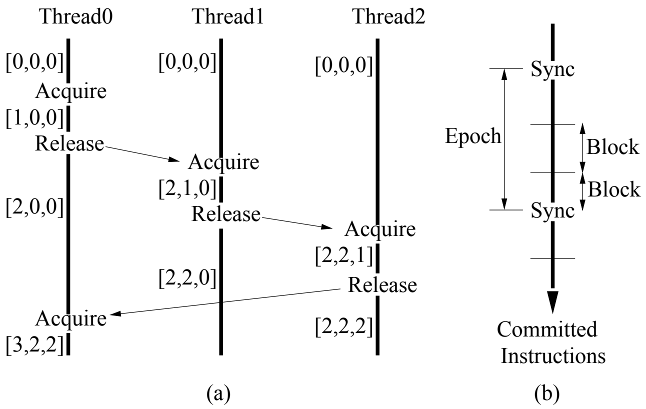


Figure 1: Example of execution of three threads with epoch timestamps in brackets (a), and definitions in a thread's execution (b).

2.2 Hardware Schemes for Race Detection

There are at least four proposals for hardware-assisted data-race detectors, namely Min and Choi's [12], ReEnact [19], CORD [18] and HARD [30]. They all detect races by tagging the state in the caches as it is being accessed, and then piggybacking the tags on cache coherence protocol messages between processors so that they can be compared.

ReEnact and CORD use the happened-before approach. They tag each cache line with timestamp information, and send and compare timestamps at least at every coherence action (invalidation of cached line or external read of a dirty cached line). In ReEnact, the tag is an index into a table of vector-clock timestamps. In CORD, the tag is four scalar timestamps (two for read and two for write), and two sets of read-write bits per word. HARD uses the lockset approach and, therefore, only handles locks properly. It tags each cache line with two special state bits, and a bit vector that represents the lockset for the line. These bits are checked at every access to the line, and are kept coherent by the coherence protocol as if they were data. Finally, Min and Choi use the happened-before approach for only nested doall loops. They tag each cache line with a set of read and write bits for each doall nesting level, and perform tag checking at every cache access.

In all these schemes, the hardware can easily detect an address and an instruction involved in a race on the fly. Then, to reveal the other (or several other) instructions involved in the same race to the programmer, it is necessary to roll back and re-execute the code section. For example, ReEnact [19] executes under thread-level speculation. If a race is detected, it rolls back execution to the most recent checkpoint, places a watchpoint on the racing address, and re-executes. The machine then captures all the accesses to the racing address. In addition, re-execution is also necessary to discard false-positive races. They occur because some of these hardware schemes tag the cache at line-size granularity. Consequently, accesses from different processors to different words of the same line (false sharing) may appear as races. Re-execution disambiguates this case.

Overall, these schemes have two shortcomings. First, they modify the L1 cache, the operations performed on some local/external accesses to L1, and the cache coherence protocol messages. These are key hardware structures, either time-critical or complicated to design and debug. Second, when a line is displaced or invalidated from the cache, the system loses its ability to detect a data race for that line. An exception is CORD, which keeps some timestamp information in memory. We would like a design that decouples cache and coherence protocol from race detection, and has a longer detection window than that provided by cache residence.

2.3 Hardware Address Signatures

A hardware address signature is a long register (e.g., 2Kbits long) where the memory addresses accessed by the processor are automatically hash-encoded and accumulated using a Bloom filter [2]. Signatures have been used in the Bulk system [4] and several subsequent multiprocessor designs (e.g., [3, 13, 28]) to detect data dependences between threads in thread-level speculation and transactional memory. Signatures are efficiently operated on in hardware using simple logic (e.g., bit-wise AND of signatures to find common addresses). From a signature, it is only possible to obtain a superset of the addresses that were originally encoded in the signature. Consequently, operations on signatures may produce false positives, although not false negatives.

In this paper, we use signatures to detect data races. While HARD [30] used a Bloom filter to encode locksets for efficient manipulation, this is the first paper that uses address signatures for happened-before race detection.

3. SIGNATURE-BASED RACE DETECTION

3.1 Overview of the Idea

The idea of SigRace is to automatically record the set of addresses accessed by the processor in a code section in hardware signatures. At appropriate intervals, the signatures and the epoch timestamp are automatically passed to an on-chip hardware module called *Race Detection Module* (RDM). The RDM keeps the signatures and the timestamp in an in-order queue assigned to the initiating processor, and compares them to the entries of queues assigned to other processors using very efficient signature operations. The comparison quickly determines whether there has been a potential data race.

SigRace addresses the two shortcomings of existing hardware-assisted schemes. First, there are no L1 cache modifications, no critical-path operations performed on local/external accesses to L1, and no cache coherence protocol message changes. Signature generation, storage, and comparison are decoupled from caches and coherence protocol. Second, lines can be displaced or invalidated from caches without SigRace losing the ability to detect data races. In practice, the RDM necessarily has limited storage capacity, and old signatures are discarded when room is needed, also limiting the race detection window. We will see, however, that SigRace's race detection capability is higher than that of cache-based systems.

Like all of the currently-proposed hardware schemes (Section 2.2), SigRace needs to rely on rollback and re-execution to provide the full set of racing instructions to the programmer, and to disambiguate false-positive races. However, using signatures introduces two differences. First, since SigRace detects races lazily when signatures are compared, SigRace without re-execution cannot provide any of the racing instructions. In contrast, since the currently-proposed schemes detect the race eagerly, they can plausibly detect one of the racing instructions without re-execution.

The second difference is the source of false-positive races. Unlike currently-proposed schemes, SigRace does not suffer false positives due to false sharing. This is because SigRace encodes *fine-grain* (e.g., word) addresses in signatures. Accesses to different words of the same line do not induce a data race report. However, address aliasing in signatures may induce false positives in SigRace. This is because signatures represent a superset of the addresses that were encoded [4]. False negatives are not possible.

For simplicity, we want SigRace to support the rollback and re-execution largely in software. Consequently, SigRace does not use thread-level speculative execution. Reads and writes commit as usual. We use the ReVive checkpointing/rollback mechanism proposed by Prvulovic *et al.* [20]. After rollback and re-execution to the race, an analysis phase takes place. We envision rollback, re-execution, and analysis to be transparent to the user, who should at worst notice a slight slowdown when many false data races are detected.

Address collection into signatures is disabled and enabled in software at kernel entries and exits, respectively, and, optionally, at library entries and exits. This typically improves race detection. Moreover, the programmer can disable address collection during the execution of certain code sections. Finally, signatures are assigned to software threads rather than to hardware contexts.

In the following, we describe SigRace's operation under three stages: normal execution, re-execution, and race analysis. For simplicity of presentation, this section assumes one thread per processor and no thread migration. The implementation of SigRace is left for Section 4.

3.2 Normal Execution under SigRace

The execution of a thread is logically divided into epochs, which are the dynamic instructions committed between synchronization operations (Figure 1(b)). The latter include, e.g., acquiring a lock, releasing it, waiting on a flag, setting a flag, or crossing a barrier. Under SigRace, each processor keeps the timestamp of the current epoch, which is encoded and updated as per Section 2.1. In addition, the processor has a Read (R) and a Write (W) Signature. When a load or a store commits, a hardware Bloom filter as in [4] automatically hash-encodes and accumulates the address loaded from or stored to, respectively, into the correct signature.

Ideally, a processor can keep its timestamp and R and W signatures to itself until the end of the epoch. At that point, they are made visible to all other processors, to check for data races. In practice, long epochs would cause the signatures to accumulate so much state that any operation on them would likely induce many false positives due to aliasing [4]. Consequently, when the processor has committed a certain number of dynamic instructions that we call a *Block* without finding a synchronization operation, the hardware automatically passes the timestamp and signatures to the RDM. Figure 1(b) shows the resulting execution: a block finishes when either a certain number of dynamic instructions have been committed or a synchronization operation is found.

The exact actions taken when a block in processor *i* finishes for either reason are as follows (Figure 2). First, the hardware automatically dumps the timestamp and R and W signatures into a memory-mapped FIFO queue of registers in the RDM called BlockHistoryQueue[i] (Step *1* in the figure). To save network bandwidth, the data is transferred in compressed format. The R and W signatures are then cleared. Finally, if the block finished because of a synchronization operation, library software updates the epoch timestamp and then saves it in a log in memory to keep a trail of timestamp changes — which is useful if we need to roll back execution.

At the RDM, simple hardware automatically compares the incoming data to entries in all the other BlockHistoryQueue[.] (Step *2* in the figure). Specifically, for a given BlockHistoryQueue[j], the incoming timestamp TS_{i0} gets compared to TS_{j0}, TS_{j1}, etc — in sequence order starting from the latest one available. Such comparisons stop as soon as one of the *j* timestamps is found to precede the incoming timestamp — in this case, due to transitivity, all earlier *j* timestamps will also precede the incoming one. Then, for all timestamp pairs found to be unordered (e.g., TS_{i0} and TS_{jN}), simple

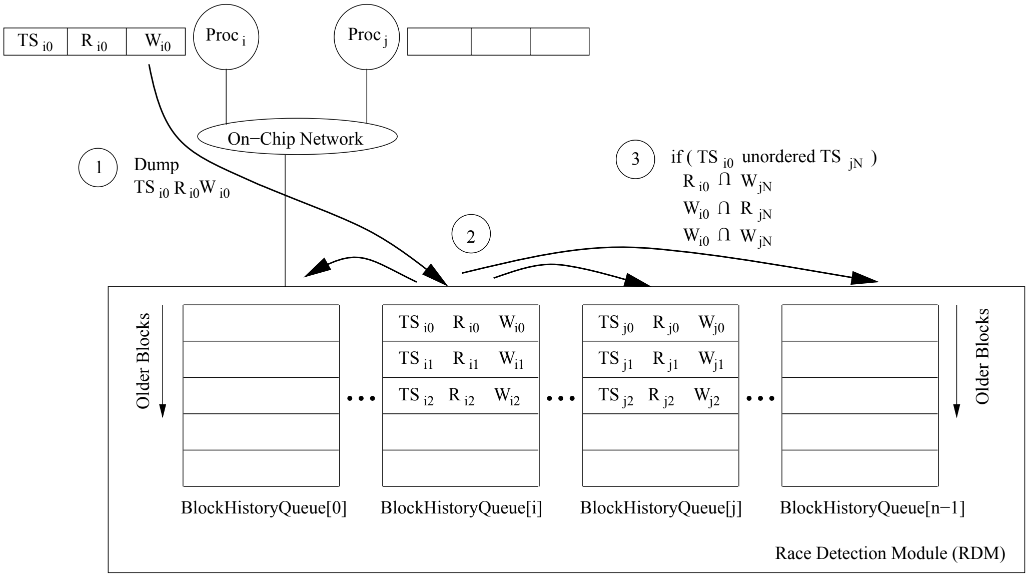


Figure 2: Operations when a block finishes. In the figure, *TS*, *R*, and *W* refer to timestamp and read and write signature, respectively. In any BlockHistoryQueue[k], entries for older blocks have higher subscripts.

signature functional units compute $R_{i0} \cap W_{jN}$, $W_{i0} \cap R_{jN}$, and $W_{i0} \cap W_{jN}$ (Step *3* in the figure). If any of these is not null, the two blocks have accessed the same location(s) without synchronization and at least one wrote. We have detected a data race — or a false positive. We call these two blocks and their corresponding threads the *Conflicting Blocks* and *Threads*.

A BlockHistoryQueue[k] is a FIFO queue. When it overflows, information on the displaced blocks is lost. We have lost the ability to detect data races in those blocks. We accept this limitation to keep overheads to a minimum.

3.3 Re-Execution under SigRace

When a pair of Conflicting Blocks is found, we want to identify for the user the exact instructions and address(es) involved in the race(s), and to weed out any false positive transparently to the user. In our design, an exception forces all processors to roll back to the previous checkpoint and enter the *Re-execution* mode. In this section, we describe the checkpointing support and the re-execution process.

3.3.1 Checkpointing Support

The SigRace design that we present needs a low-overhead checkpointing scheme. Ideally, such a scheme would already be in place for reliability purposes, and SigRace would reuse it. One possible scheme is ReVive [20], which performs incremental memory-state checkpointing. With ReVive, all processors are interrupted at intervals of several milliseconds, at which point, a software handler creates a global light-weight checkpoint. The checkpoint consists of saving the register state of all processors and writing back all the dirty cache lines to memory. Then, during execution, the memory controller logs every first update to a main memory location since the previous checkpoint (i.e., the log saves the value in memory before the first write-back of a dirty line from caches to the location). Rolling back to the previous checkpoint involves undoing the trail of memory updates from this log until the checkpoint, and then restoring the registers. The ReVive design in Prvulovic *et al.* [20] adds a 6.3% execution time overhead.

In addition, the kernel collects and buffers the inputs to the program during Normal execution — such as interrupts, system call returns, and I/O input — and passes them to the re-execution at appropriate times. Support similar to this is provided by Flashback [25] and Rx [21], which require no hardware modifications.

With these two mechanisms, we will now see that SigRace re-executes following the same paths until the first data race is found.

3.3.2 Re-Execution Operation

Re-execution forces the application to follow the same order of epochs as in the original execution, and leaves each thread at the beginning of the *epoch* that the thread was executing when the race was detected. This is shown in Figure 3, where a race was detected at the points shown in Figure 3(a), and re-execution brings the threads to points *A*, *B*, *C*, and *D* in Figure 3(b). Note that re-execution does not bring each thread to the actual block that it was executing when the race was detected. This is because we do not rely on the ability to reproduce block boundaries exactly.

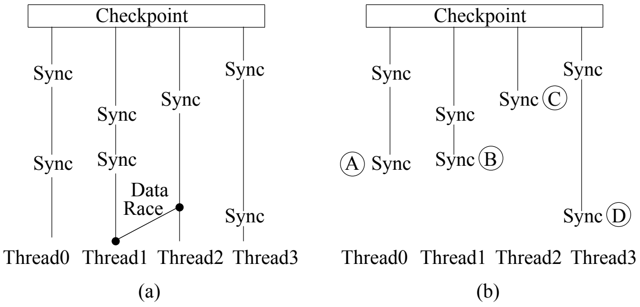


Figure 3: Detection of a data race during Normal (a) and Re-execution (b) modes.

To reproduce the order of epochs, SigRace uses the history of logged timestamps (Section 3.2). They encode the history of synchronization operation orders — i.e., which thread completed a synchronization operation before which other thread. SigRace uses these timestamps to follow the same synchronization orders.

Specifically, each processor has a Thread Re-execution Timestamp (TRT) register into which, as it re-executes, it successively loads the timestamps logged since the checkpoint. Recall that each timestamp was saved *after* the processor went past a synchronization operation. In addition, there is a shared software structure in memory called Global Re-execution Timestamp (GRT) that contains the most up-to-date logical time of each processor during the re-execution. In other words, while the TRT is the "thread view" of the current re-execution time, the GRT is the "true global view". Each processor compares its TRT to the GRT to see when the other processors have executed all the earlier epochs and the processor can proceed. Proceeding means for the processor to perform its next synchronization operation, update its own component of the GRT, execute its next epoch, and read its next logged timestamp into its TRT.

The actual algorithm is as follows. Let us call $grt[.]$ the GRT and $trt_p[.]$ the TRT of processor p. Each i in $grt[i]$ is the latest epoch from processor i that has been executed. For example, Figure 4 repeats the timeline of Figure 1(a) and shows with an arrow the current position of each replaying processor. As a result, $grt[.] = [2, 1, 0]$. All processors are waiting at a synchronization operation and we need to decide which one(s) to execute next. Each processor has loaded into its trt the timestamp it had *after* the synchronization (e.g., $trt_1[.] = [2, 2, 0]$). When a given processor p finds that $grt[i] \geq trt_p[i]$ for all $i \neq p$, then processor p executes the synchronization operation, sets $grt[p] = trt_p[p]$, executes its next epoch, and loads its next logged timestamp into $trt_p[.]$. The last two operations are not performed if there is no next logged timestamp. In the figure, the only processor for which the inequality is true is Processor *1*. Consequently, Processor *1* will execute the release and set GRT to [2,2,0]. Since it has no further timestamp logged, it will wait there.

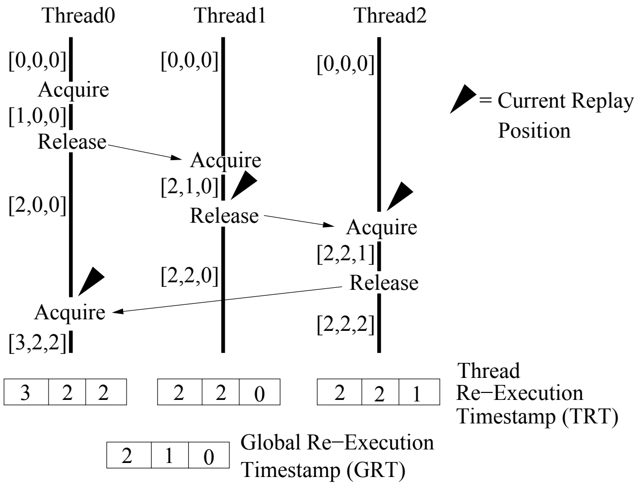


Figure 4: Re-execution using the logged timestamps.

3.4 Race Analysis under SigRace

When all threads have reached their last logged timestamp, execution enters the *Analysis* mode. In this mode, only the threads involved in the data race execute, while the others stall. Specifically, first, the two processors executing the Conflicting threads load into a local register called the *Conflict Signature* the intersection of the two Conflicting blocks' signatures — namely the union of $R_{i0} \cap W_{jN}$, $W_{i0} \cap R_{jN}$, and $W_{i0} \cap W_{jN}$ as per Section 3.2. The Conflict Signature holds the hashed address(es) involved in the race. Then, the two Conflicting threads execute normally up to their next synchronization points, while the hardware automatically intersects their loads and stores against the Conflict Signature. Every time a non-null intersection occurs, a trap is triggered, which records the memory address and the PC. Finally, when both threads have reached their next synchronization points, a software handler compares the record of trapping addresses in both processors, to see if there are common addresses. If so, SigRace has found a data race, which it reports to the user. Otherwise, it was only a false positive and is ignored.

As each Conflicting thread reaches its next synchronization point, it may have executed past its Conflicting block. This is fine, since it enables us to capture as many of the references involved in the data race(s) as possible.

After the Analysis step, execution *seamlessly* returns to the Normal mode of execution. This is enabled by the fact that SigRace continued to perform timestamp/signature logging and signature intersection during Re-execution and Analysis modes — exactly like it did during Normal mode. In this way, the trail of timestamps and signatures is up to date at the point where Analysis completes and all processors resume Normal execution.

Because the Analysis step may push program execution beyond what was executed before the rollback, it is possible that the Analysis step discovers new data races. To address this case, SigRace proceeds as follows. Every time two blocks are found to conflict during Analysis ($R_{i0} \cap W_{jN}$, $W_{i0} \cap R_{jN}$, or $W_{i0} \cap W_{jN}$ are not null), a handler compares their intersection against the contents of the Conflict Signature. If the latter is a superset, no action is taken because this race is already being processed (call it *Race1*). Otherwise, the handler saves the signature intersection and records the need to analyze the new data race (call it *Race2*) later. In this case, after *Race1* is fully analyzed, execution is rolled back, and we proceed to perform Re-execution and Analysis for *Race2*. Note that we cannot analyze the two races concurrently because, by the time we detect the presence of *Race2*, processors have already issued some of the references associated with it.

Overall, to minimize the amount of re-execution, SigRace is designed as follows. When a processor in Normal execution detects a pair of Conflicting blocks, it does not immediately request a rollback. Instead, it continues executing for several more blocks (e.g., 5–10) or until it synchronizes, before interrupting all other processors and requesting rollback. The goal is to collect as many potential races as possible. During Analysis, the Conflict Signature of each processor contains the racing addresses of all the races that the processor is involved in characterizing. In this way, multiple races are analyzed concurrently. Finally, SigRace also saves the Conflict Signatures of the races that it has finished analyzing. In this way, if SigRace has to re-execute the same code a second time, it can ignore the race already analyzed.

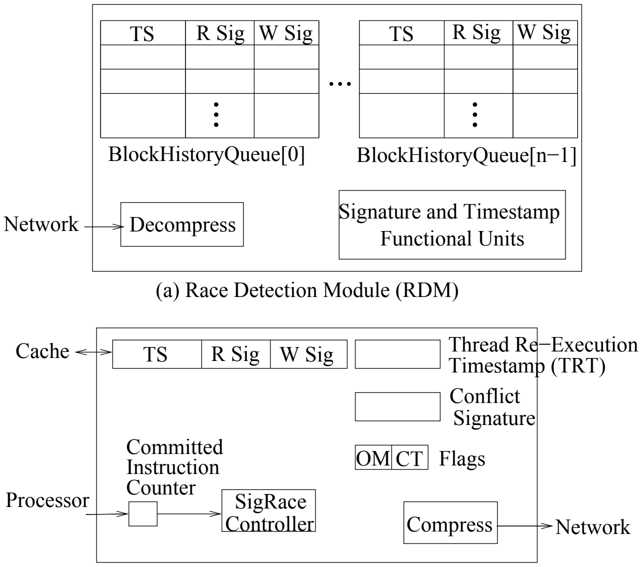
4. SIGRACE IMPLEMENTATION

Our implementation of SigRace requires some hardware and software changes to a chip multiprocessor. The hardware changes are the Race Detection Module (RDM) and some additions to the per-processor cache hierarchy. The cache tag and data arrays are *unmodified*. Also, SigRace does not use speculative multithreading. On the software side, SigRace needs an augmented synchronization library. In this section, we describe the hardware and software components, and then how SigRace is virtualized to make it usable.

4.1 Hardware Modifications

The RDM is a simple on-chip hardware module that is connected to the on-chip network. As shown in Figure 5(a), it contains the

BlockHistoryQueue[.], which stores past timestamps (TS) and signatures for all the processors (Section 3.2). It also includes functional units that operate on signatures (like in Bulk [4]) and timestamps.



(a) Race Detection Module (RDM)

(b) Additions to the Private Cache Hierarchy

Figure 5: Hardware support needed by SigRace.

SigRace also requires some per-processor hardware that is placed in the cache hierarchy in a module that interfaces with the processor, the cache and the network (Figure 5(b)). The module includes storage for the current epoch timestamp and the current block's R and W signatures. The addresses hashed into signatures have a finer granularity than cache line, so that false sharing of a line does not trigger incorrect data race alarms. A good choice is to use word addresses. The module also includes the Thread Re-Execution Timestamp (TRT) for re-execution (Section 3.3) and the Conflict Signature for analysis (Section 3.4). There are two flags, namely the *Operation Mode* (OM) that denotes whether the hardware is in Normal, Re-execution, or Analysis mode, and the *Conflicting Thread* (CT) that denotes whether the thread is a Conflicting one (Section 3.2). There is also a *Committed Instruction Counter*. When the latter reaches the maximum value set for a block — or an approximate value, since there is no need to be exact — it sends a signal to terminate the current block. The SigRace controller then initiates the following actions: dump TS, R and W into the corresponding BlockHistoryQueue[i], and clear R, W, and the Committed Instruction Counter.

The TS and R and W signatures are compressed before being sent to the on-chip network, and decompressed as they get into the RDM. We call these network messages the *Summary* messages. Their compressed size is ≈100 bytes — for 2 signatures of 2 Kbits each and a 160-bit timestamp. This is less than the size of two cache lines, and is sent out every time a block completes (≈2,000 committed instructions). Summary messages from the same processor need to arrive at the RDM in order; messages from different processors can arrive in any order. This centralized RDM design is fine for the small numbers of processors considered in this paper (8). In large, distributed machines, the RDM can be distributed as well.

4.2 Software Interface

High-level synchronization constructs such as M4 macros [11] and OpenMP directives [6] are commonly used by programmers and parallelizing compilers. These constructs can enable SigRace transparently. Specifically, we rewrite such constructs to encapsulate the SigRace operations. As a result, the application code does not need be modified at all, and all we need is to relink it with the new M4 or OpenMP library.

To accomplish this, we start by adding three processor instructions that operate on local SigRace structures (Table 1). Two of the instructions (*collect_on* and *collect_off*) enable and disable the collection of addresses into signatures, and the counting of committed instructions. A variation of these instructions could perform these actions only on a range of addresses. These instructions are used to prevent the signatures from being polluted by unrelated accesses (such as those from the OS or the instrumentation added to the macros) or by obviously-private accesses (e.g., those to the stack). They can also be used to mark a benign data race or an epoch that should *skip the checking*. The other instruction (*sync_reached*) is invoked when execution reaches a synchronization operation. Specifically, it is invoked immediately before performing a release-type operation and immediately after performing a successful acquire-type operation. It tells the SigRace controller to dump TS, R and W into the RDM, clear R, W, and the Committed Instruction Counter, and increment the counter in TS that corresponds to the local thread.

Instruction	Description
collect_on	Collect addresses into R and W, and count committed instructions.
collect_off	Do not collect addresses into R or W, or count committed instructions.
sync_reached	Dump TS, R, and W into the RDM. Clear R, W and the Committed Instruction Counter. Increment the counter in TS that corresponds to the local thread.

Table 1: Instructions to manage SigRace structures.

For simplicity, we assume that these instructions make their side effects visible only when they commit — like the updates of signatures by loads and stores. A design where these actions happen earlier in the pipeline can also be conceived.

With these instructions, we can build new macros for all the synchronization primitives. As an example, we consider the M4 macros for UNLOCK and LOCK. Table 2 shows the conventional implementation and the one adapted for SigRace in Normal execution mode (SN_UNLOCK and SN_LOCK). In the SigRace version, synchronization variables have a *lock* and a *timestamp* field — shown as $1.lock and $1.timestamp, respectively.

In SN_UNLOCK, before unlocking the *lock* field of the variable, the *sync_reached* instruction executes (Table 2). Then, the TS of the processor — which has already been updated by *sync_reached* — is saved in the *timestamp* field of the variable (Line 3). Then, the lock is released. Finally, the updated TS is explicitly saved in a TS log in memory, in case it is needed for re-execution (Line 5). In SN_LOCK, after the lock is acquired and *sync_reached* executed, the new TS is generated. This is done by taking the current TS — which is already updated by *sync_reached* — and the value stored in *timestamp*, and applying the algorithm of Section 2.1 (shown as *GenerateTS*). Finally, the TS is saved in the TS log.

Finally, we need to augment the macros to work for all exe-

Operation	Implementation	Code
Unlock	Conventional	1: UNLOCK(‘{ 2: unlock($1);}’)

	SigRace (Normal Execution Mode)	1: SN_UNLOCK(‘{
		2: sync_reached;
		3: $1.timestamp = TS;
		4: unlock($1.lock);
		5: AppendtoTSLog(TS,TSLog);
		6: }’)

Lock	Conventional	1: LOCK(‘{ 2: lock($1);}’)

	SigRace (Normal Execution Mode)	1: SN_LOCK(‘{
		2: lock($1.lock);
		3: sync_reached;
		4: TS = GenerateTS(TS,
		5: $1.timestamp);
		6: AppendtoTSLog(TS,TSLog);
		7: }’)

Table 2: UNLOCK and LOCK macros: conventional implementation and one for SigRace in Normal execution mode.

cution modes. As an example, Table 3 shows the resulting final S_UNLOCK macro, which builds on top of SN_UNLOCK. The code is surrounded by *collect_off* and *collect_on* to prevent these accesses from polluting the signatures. If the OM flag indicates we are in Re-execution mode, we load the next timestamp from the old timestamp log into the TRT (Line 4), and spin until the GRT reaches the appropriate value (Line 5) (Section 3.3). If, instead, we are in Analysis mode, we have completed the execution of an epoch in this mode in one of the Conflicting threads. It is now time to analyze the record of traps observed (Line 8) (Section 3.4) and, depending on the outcome, proceed in Normal mode.

Irrespective of the mode, we then need to perform the unlock operation (Line 11) as was described in Table 2. Then, if we are in Re-execution mode, we update the GRT with the corresponding counter from the TRT (Line 13) and then check if the old timestamp log is empty. If so, we set the mode to Analysis (Line 15) and check the CT flag to see if this is a Conflicting thread. If so, we set up the Conflict Signature and continue execution (Section 3.4). Otherwise, the thread stalls until the Conflicting threads complete the analysis. After that, we return to Normal mode. Similar code is generated for the other synchronization constructs.

4.3 SigRace Virtualization

Previous discussions have largely used thread and processor interchangeably. In reality, SigRace has to function in an environment where threads migrate across processors and the number of threads and processors may be different. In this section, we consider this environment. We do it in three steps. First, we allow threads to migrate across processors but the number of threads and processors is the same (*Migration* environment). Second, we augment Migration to allow the number of threads to be different (and typically larger) than the number of processors; some threads are waiting for an available processor (*Multiplex* environment). Finally, we augment Multiplex to allow processors that support multiple hardware contexts (*Multithreaded* environment). We discuss each environment under Normal execution, and then consider the Re-execution and Analysis modes.

```
1:   S_UNLOCK(‘{
2:     collect_off
3:     if (Flags.OM == Re-execution){ /* Re-exec. mode? */
4:       LoadfromTSLog(TRT,OldTSLog);
5:       WhileNotMyTurn(TRT,GRT) {};
6:     }
7:     else if (Flags.OM == Analysis){ /* Analysis mode? */
8:       AnalyzeRecordOfAccesses(); /* Analyze data */
9:       Flags.OM = Normal; /* End of Analysis mode */
10:    }
11:    SN_UNLOCK($1)
12:    if (Flags.OM == Re-execution){ /* Re-exec. mode? */
13:      UpdateGRT(TRT,GRT);
14:      if (OldTSLogEmpty) {
15:        Flags.OM = Analysis; /* Analysis mode */
16:        if (Flags.CT) { /* One of the Conflicting threads? */
17:          LoadConflictSignature();
18:          /* Set up the Conflict Signature. Continue */
19:        }
20:        else { /* Not Conflicting thread */
21:          StallUntilEndAnalysis(); /* Stall */
22:          Flags.OM = Normal; /* End of Analysis */
23:        }
24:      }
25:    }
26:    collect_on
27:  }’)
```

Table 3: Resulting UNLOCK macro for SigRace.

4.3.1 Enabling Thread Migration

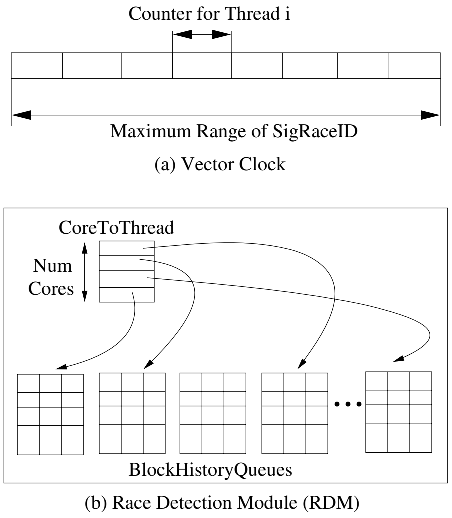
Epoch timestamps and signatures belong to threads rather than processors. Consequently, in the Migration environment, the timestamp is saved when a thread is pre-empted and restored on the processor where the thread runs next. Signatures are not saved and restored because, on thread pre-emption, the currently-running block finishes. At that point, the signatures are sent to the RDM and then cleared.

The threads of a program have a statically-assigned *SigRaceID*, which goes from 0 to the number of threads in the program minus one. They use their SigRaceID to index into vector clocks of processors and array of BlockHistoryQueues in the RDM. Specifically, counter *i* in a vector clock belongs to the thread with SigRaceID = *i*, irrespective of which processor the thread is currently running on. Such thread always updates counter *i* in the vector clock of the processor it is running on. Moreover, signatures from that thread will always be dumped on BlockHistoryQueue[*i*] in the RDM.

In this environment, the hardware in Figure 5 is affected as follows. First, the components in Figure 5(b) belong to a thread. Consequently, the operating system saves and restores them on context switch — except for the signatures and the Committed Instruction Counter, which are cleared. Second, the RDM in Figure 5(a) includes a new hardware structure. It is an indirection table called the *CoreToThread* table. This table has as many entries as cores in the chip. It contains the mapping between core number and SigRaceID of the thread currently running on the core. The operating system updates the table on context switches. During execution, when the RDM receives a message from core *j*, the hardware reads CoreToThread[*j*]. It then uses the value read, say *i*, to store signatures and timestamp in BlockHistoryQueue[*i*].

4.3.2 *Different Thread & Processor Numbers*

The hardware for the Multiplex environment extends the one for Migration by supporting a range of SigRaceID values larger than the number of cores. Specifically, each vector clock in processors and each (software) timestamp field in sychronization variables is sized up to have as many counters as the maximum range of SigRaceID (Figure 6(a)). Similarly, the RDM has as many BlockHistoryQueues as the maximum range of SigRaceID, and the width of the CoreToThread table is increased accordingly (Figure 6(b)).



(a) Vector Clock

(b) Race Detection Module (RDM)

Figure 6: Supporting more threads than cores.

Before a program runs, it declares the number of threads that it will use, and the hardware and software structures mentioned are sized accordingly. While the program runs, the RDM intersects an incoming signature message against all BlockHistoryQueues — even those that belong to threads that are currently not running.

4.3.3 *Enabling Multiple Contexts per Processor*

The Multithreaded environment extends the Multiplex one in that each hardware context in a processor counts as an additional virtual core. This requires increasing the number of counters in the vector clocks and in the timestamp fields of synchronization variables, and the number of BlockHistoryQueues in the RDM.

Each hardware context has a copy of the hardware shown in Figure 5(b). Moreover, the messages that processors send to the RDM have to include both the core ID and the hardware context ID within the core. Only then can the RDM identify the appropriate BlockHistoryQueue to update.

4.3.4 *Re-Execution and Analysis Modes*

In all three environments described, threads are re-executed without any scheduling constraints. Specifically, Re-execution does not need to reproduce the thread schedule followed during the Normal execution. All that is required is that the order of successful synchronization operations be the same as in the Normal execution. This is ensured by reading the timestamp log from memory (Section 3.3.2) and enforcing it. At worst, in the Multiplex environment, performance may suffer because a thread that owns a critical lock may be temporarily not scheduled, preventing other thread from making progress.

In addition, re-execution does not need to reproduce the same block sizes as in the Normal execution. The reason is that Re-execution brings the threads to the beginning of epochs, rather than to specific blocks within epochs.

The checkpointing support described in Section 3.3.1 can still be used. Such support is able to return the memory state of the whole machine to a certain point in the past — without knowing about the number of threads in the program or how they were scheduled. If, however, it is desired to checkpoint only one of several applications that may be running, a different, application-level checkpoint design is needed. Such a design is outside this paper's scope.

As expected from the discussion on the Normal execution mode, there are a few structures used during Re-execution that need to change. First, the TRT (Figure 5(b)) is thread-private, and is saved and restored on context switch. In addition, the TRT and GRT have as many counters as the range of SigRaceIDs in the program. Moreover, threads use their SigRaceID to index into the TRT register, irrespective of what core they are currently running on.

Finally, the Analysis mode requires no change, since only the conflicting threads are participating in the execution. Both the Conflict Signature and the Conflict Thread structures (Figure 5(b)) are thread-private variables and the hardware saves and restores them on context switch.

5. EVALUATION

To evaluate SigRace, we consider four issues: (1) the signature configuration, which determines the number of false positives, (2) the block size and number of entries in each BlockHistoryQueue[i], which determine the window of monitored execution, (3) the effectiveness of SigRace in detecting data races, and (4) the overheads of SigRace. In the following, we first overview the experimental setup and then consider each issue in turn.

5.1 Experimental Setup

Since we are interested in the high-level parameters of SigRace, we use the PIN [10] binary instrumentation tool to design a simulator of the SigRace hardware, and run the applications on a real 8-processor shared-memory machine. This approach has the benefit of execution-driven simulation without incurring the slow speeds of typical cycle-accurate simulators. Table 4 shows the default parameters used in the simulation.

Num. of processors: 8 L1 size: 32 Kbytes L1 line size: 64 bytes Coh. protocol: MESI Checkpt. interval: 1 M committed instr./proc.	Timestamp size: 8 x 20 = 160 bits Sig. size: 2 Kbits each R and W Block size: 2,000 committed instr. BlockHistoryQueue[i] size: 16 entries
Benchmarks: SPLASH2 kernels: FFT, Cholesky, LU SPLASH2 applications: Barnes, Volrend, Ocean, Radiosity, Raytrace, Water-ns, Water-spatial PARSEC kernels: Dedup, Streamcluster PARSEC applic: Blackscholes, Fluidanimate, Swaptions	

Table 4: Default parameters used in the evaluation.

We model an 8-core chip multiprocessor where 32-Kbyte L1 caches are connected in a multistage network and kept coherent with a MESI cache coherence protocol. The timestamp size is very conservatively set to 160 bits. The default values for the size of signatures, block, and BlockHistoryQueue[i] are set according to the sensitivity analyses presented later. We take periodic global checkpoints. A checkpoint is created as soon as a processor has committed 1 M instructions. We use the checkpointed information as a starting point of our Re-execution and Analysis algorithms.

We evaluate SigRace with the SPLASH2 and PARSEC [1] benchmarks. These benchmarks are representative of parallel workloads and exhibit a variety of memory access patterns. For SPLASH2, we use the default inputs, while for PARSEC, we use the *simmedium* input size. We report data for 10 SPLASH2 and 5 PARSEC benchmarks. As shown in Table 4, we separate them into SPLASH2 kernels, SPLASH2 applications, PARSEC kernels, and PARSEC applications.

5.2 Signature Configuration

We test multiple signature configurations, denoted as $B_i\_S_j$. We first partition the address into 2 portions. The possible configurations are the B_i in Table 5. Then, we use multiple Bloom filters in parallel using the *H3* hash function as in [23] — half of them process one portion while the other half the other. The configurations are the S_i in Table 6.

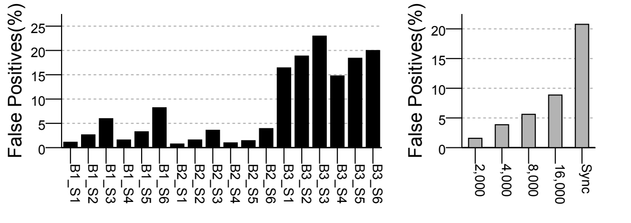
Configuration	Address Partition	
	LSB	USB
B_1	8	24
B_2	10	22
B_3	16	16

Table 5: Address partitions. LSB and USB stand for Lower and Upper Sliced Bits.

Configuration	# of Bloom Filters (*k*)	Bits per Bloom Filter (*n*)	Sig Size (*k* × *n*)
S_1	16	256	4Kbit
S_2	16	128	2Kbit
S_3	16	64	1Kbit
S_4	8	512	4Kbit
S_5	8	256	2Kbit
S_6	8	128	1Kbit

Table 6: Signature organizations.

We run the applications and count the number of signature intersections that indicate a collision while there is none. The ratio of this number over the total number of signature intersections is the false-positive rate. Figure 7(a) shows the average false-positive rate of the applications for our default parameters. In the rest of the paper, we use $B_2\_S_2$, where the false-positive rate is 1.57%.



(a) Signature Configuration

(b) SigRace Block Size

Figure 7: False positive rate versus signature configuration (a) and versus block size (b).

5.3 Block and BlockHistoryQueue[i] Size

If we choose a large SigRace block then, with the same BlockHistoryQueue[i] (BHQ[i]) size, we can monitor a larger instruction window for possible data races. However, as the block size increases, the signature false-positive rate also increases. Figure 7(b) shows the false-positive rate for different block sizes beyond our default of 2,000 committed instructions. *Sync* means terminating a block only at synchronizations. We see that larger blocks induce more false positives.

For a given block size, if we increase the number of entries in BHQ[i], we cover a larger instruction window. However, we have to do more signature operations and the BHQ takes more area.

To evaluate these issues, we run the applications with different numbers of entries in BHQ[i] and different block sizes. When the RDM checks an incoming signature against a BHQ[i], the hardware operates on each of the entries in the BHQ[i] until it finds a block that is a predecessor of the incoming one. If there is such a predecessor, then SigRace does not lose any race detection opportunity. We call this event a *Hit*. Otherwise, SigRace loses race detection opportunity beyond the oldest entry in BHQ[i]. We are interested in the execution window that starts at the previous checkpoint and ends at the block just before the oldest entry in BHQ[i]. We call it the *Lost Detection Window*.

Figure 8(a) shows the lost detection window as a percentage of the checkpoint interval, while Figure 8(b) shows the hit rate of a signature against a BHQ[i], and Figure 8(c) shows the number of timestamp comparisons in a BHQ[i] per signature until hitting in the BHQ[i] or exhausting all full BHQ[i] entries. All figures have the same X axis and share the same legend.

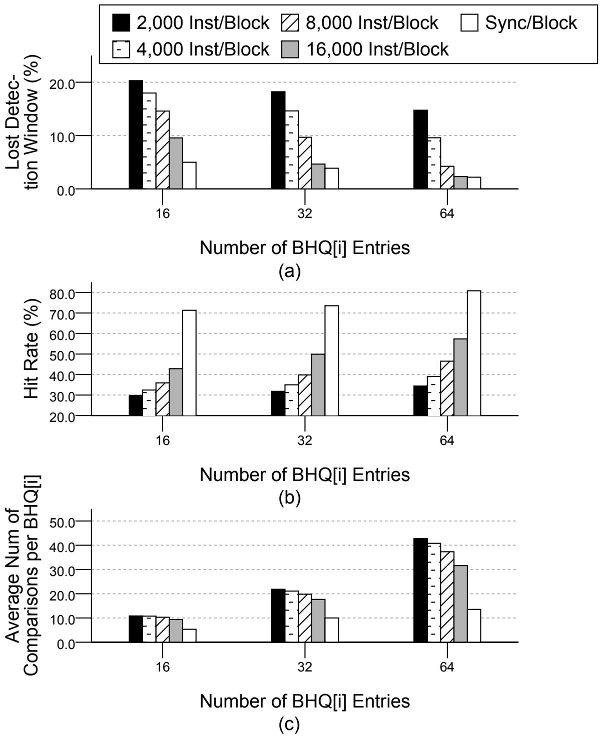


Figure 8: Lost detection window (a), hit rate (b), and number of timestamp comparisons (c) for different numbers of BHQ[i] entries and block size. All figures share the same legend.

Application	Finding Existing Races						Finding Injected Races				
	Ideal SigRace		SigRace		W-ReEnact		Racy	Static Races Found		Runs w/ Races Found	
	Stat	Dyn	Stat	Dyn	Stat	Dyn	Runs	SigRace	W-ReEnact	SigRace	W-ReEnact
FFT	–	–	–	–	–	–	25/25	600	150	25	25
Cholesky	16	19964	16	3539	16	388	3/25	2	2	1	1
LU	–	–	–	–	–	–	25/25	28	75	25	25
Barnes	11	4416	11	719	6	419	1/25	3	1	1	1
Volrend	27	26846	27	11607	18	6858	23/25	345	74	23	21
Ocean	1	29	1	29	1	6	7/25	8	8	7	7
Radiosity	15	59307	15	16951	12	14660	8/25	29	11	8	6
Raytrace	4	30	4	17	3	12	21/25	66	53	21	21
Water-ns	–	–	–	–	–	–	5/25	2	4	1	2
Water-spatial	8	82	4	27	2	3	3/25	6	6	3	3
Dedup	–	–	–	–	–	–	3/25	0	0	0	0
Streamcluster	13	68566	12	14307	12	436	6/25	7	2	5	2
Blackscholes	–	–	–	–	–	–	0/25	0	0	0	0
Fluidanimate	–	–	–	–	–	–	12/25	95	90	12	12
Swaptions	–	–	–	–	–	–	–	–	–	–	–
Total	95	179240	90	47196	70	22782	142/350	1191	476	132	126

Table 7: Effectiveness of SigRace and ReEnact with per-word timestamps in finding existing races and injected races.

We see that, as the number of BHQ[i] entries increases, the lost detection window decreases (Figure 8(a)) and the hit rate increases (Figure 8(b)). However, we have to do more timestamp comparisons until a hit or BHQ[i] exhaustion (Figure 8(c)), and the BHQ takes more area. On the other hand, for a fixed number of BHQ[i] entries, as the block size increases, we lose less window (Figure 8(a)), the hit rate increases (Figure 8(b)) and the number of comparisons decreases (Figure 8(c)) — however, we saw in Figure 7(b) that false positives increase. Overall, we choose as default a block size of 2,000 committed instructions and 16 entries in BHQ[i]. This leads to an average of 20% loss in detection window.

5.4 SigRace Effectiveness

5.4.1 Data Race Detection

To assess SigRace's effectiveness, we use it to find (i) existing data races in our applications and (ii) races that we inject in the applications. We also simulate a cache-based race detector, namely a version of ReEnact [19] with per-word timestamps (*W-ReEnact*). Table 7 shows the results.

Columns 2-7 (*Finding Existing Races*) list the number of races found by *Ideal Sigrace*, SigRace, and W-ReEnact. Ideal SigRace is a SigRace where each BHQ[i] keeps information for *all* the blocks between consecutive checkpoints — rather than for 16 blocks as in SigRace. Races are identified by the two instructions involved in the race and the address accessed. The table counts both static and dynamic races. Dynamic races are the dynamic instances of static races.

The table shows that 8 of the applications have data races. These races include, for example, reads of shared structures outside a critical section before accessing them inside the critical section. They are likely to be all benign races. However, we believe that it is important for any race detector to detect even benign races. This is because, often, benign races are a symptom that the code has a bug or something that the programmer does not understand. In any case, as described in Section 4.2, if the programmer wants SigRace to skip checking for these races, he can mark the code with *collect_off*.

The table shows that SigRace detects 90 static and 47,000 dynamic races. Compared to W-ReEnact, SigRace detects on average 29% more static races and 107% more dynamic races. SigRace's substantially higher effectiveness is due to its ability to monitor a longer window of program at a time. Finally, compared to Ideal SigRace, SigRace detects on average 95% of the static races and 26% of the dynamic ones.

We also inject races. For each application, we perform 25 runs. In each run, we randomly eliminate one dynamic lock-unlock pair or one dynamic barrier. Since the Swaptions code synchronizes with fork/joins, we could not subject it to this experiment. While these are contrived examples, they provide some insight.

Columns 8-12 (*Finding Injected Races*) show the detection capability of SigRace and W-ReEnact. Column 8 (*Racy Runs*) shows the fraction of those 25 runs that actually created races. Then, Columns 9-10 show the number of static races found by SigRace and W-ReEnact, respectively. We see that, on average, SigRace finds 150% more static races than W-ReEnact. This again shows the higher effectiveness of SigRace. Interestingly, there are two applications where W-ReEnact finds more races (LU and Water-ns). This is because, while SigRace typically monitors a longer program window, there are cases when lines remain in the caches for a long time. In this case, W-ReEnact can detect racing accesses that are far apart in the code (over 50,000 instructions apart in these examples). In general, it can be argued that races where the accesses are far apart are least dangerous, since the chances that these accesses appear in reverse order in a different run are lower. Finally, Columns 11-12 show the number of runs in which SigRace and W-ReEnact found at least one race. Again, the number for SigRace is higher.

5.4.2 Opportunity to Detect Data Races

SigRace has an advantage when addresses are in BHQ[.] and not in caches, while W-ReEnact has an edge in the opposite case. In this section, we estimate the frequency of each case. For simplicity, in this experiment only, signatures encode line addresses.

Of all the cache lines with shared data being displaced or invalidated from a cache, Figure 9(a) shows the fraction whose address is strictly present (not just due to aliasing) in the corresponding BHQ[i]. The figure shows the average for different cache sizes and application sets. For the 32KB default cache, the weighted average fraction is ≈59%. Then, Figure 9(b) shows the number of displacements or invalidations of lines with shared data per million instructions executed. For the 32KB default cache, the weighted average can be shown to be ≈2,800. Overall, roughly speaking, compared to SigRace, W-ReEnact loses detection opportunity for 0.59×2,800=1,652 lines per million instructions.

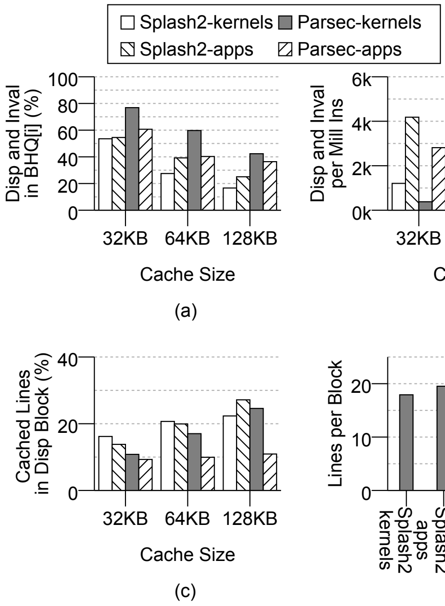


Figure 9: Opportunities for SigRace and W-ReEnact to detect races. Charts (a), (b), and (c) share the same legend.

Given a block being displaced from a BHQ[i], Figure 9(c) shows the fraction of addresses in the block's signatures that are not anywhere else in BHQ[i] and that are in the cache. For the 32KB cache, the weighted average fraction is ≈13%. Figure 9(d) shows the number of addresses of lines with shared data that are encoded in the signatures of one block. This number is on average 14. Overall, since SigRace executes ≈500 blocks per million instructions, compared to W-ReEnact, SigRace loses detection opportunity for 0.13×14×500=910 lines per million instructions. While these numbers give approximate information only, they show W-ReEnact loses more opportunities.

5.5 SigRace Overheads

We estimate the instruction, SRAM memory, bandwidth, and checkpointing overheads of SigRace. To estimate the instruction overhead, we run each application until the first true data race is fully analyzed. In the process, some false positives may occur. We count as instruction overhead all the instructions executed in Re-execution and Analysis modes to characterize the true data race and all the false positives found from the beginning of the program until that point. We stop after analyzing the first true race because then the programmer would stop execution. If the application has no true data race, we insert one in a random location.

Figure 10(a) shows the resulting instruction overhead as a percentage of committed instructions. The average bar is the mean of all the applications. The overhead depends on several things, most notably how far from the previous checkpoint is the conflict detected, and the rate of false positives. We see that, on average, the instruction overhead due to re-execution is 22%. About two thirds of it is caused by false positives.

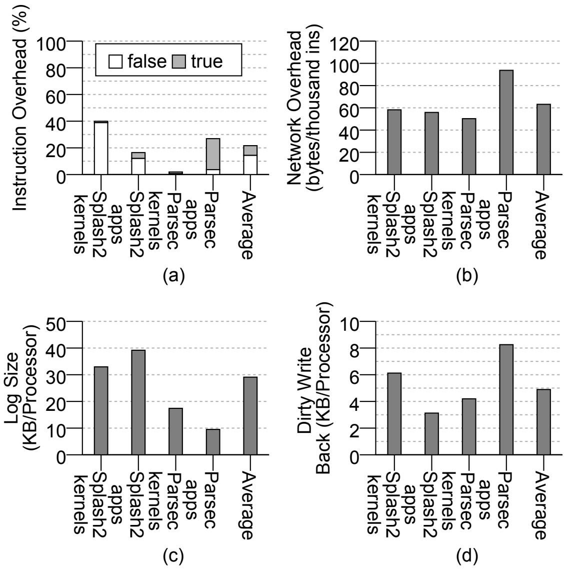


Figure 10: Instruction (a), bandwidth (b), and checkpoint-related (c and d) overheads.

From Figure 5, we see that the main SRAM memory overhead of SigRace per processor includes: a 16-entry BHQ[i] in the RDM (each entry containing a timestamp and a R and W signature), one extra timestamp and R and W signatures, the TRT, and the Conflict signature. Since timestamps are 160 bits and signatures 2K bits, this results in 8512 bytes in the RDM and 808 bytes in the cache hierarchy — independently of the cache size.

To compute the bandwidth overhead of SigRace, we count how many bytes of timestamp-signature messages (compressed) are deposited on the network. Figure 10(b) shows such number per 1,000 instructions committed. We see that, on average, the bandwidth overhead is 63 bytes per thousand committed instructions.

Finally, we measure some overheads of checkpointing every 1M instructions. As per Section 3.3.1, the memory controller saves the value overwritten by every first memory update. Figure 10(c) shows that, on average, this amounts to 29KB of log per processor between checkpoints. Also, at the point of checkpoint, the dirty lines in the cache are written back. As shown in Figure 10(d), this corresponds to, on average, 4.8KB of writebacks per processor.

6. CONCLUSIONS AND FUTURE WORK

This paper proposed SigRace, a novel approach to hardware-assisted data race detection that overcomes shortcomings of previous hardware proposals. To detect races, SigRace does not rely on cache state or coherence protocol messages. Instead, it relies on hardware address signatures. With SigRace, there are no changes to the cache or the cache coherence protocol messages, and there are no critical-path operations performed on local/external access to the cache. Moreover, lines can be displaced or invalidated from caches without affecting SigRace's ability to detect data races.

We presented the architecture of SigRace, an implementation, and its software interface. Application code is unmodified. Our experiments showed that SigRace is significantly more effective than a state-of-the-art conventional hardware-assisted race detector. SigRace found on average 29% more static races and 107% more dynamic races. Moreover, if we inject data races, SigRace found 150% more static races than the conventional scheme. Finally, SigRace had an average instruction overhead due to re-execution of 22%, a bandwidth overhead of 63 bytes per thousand committed instructions, and an SRAM memory overhead of ≈9KB per processor.

We are continuing our work in two main directions. The first one involves eliminating or minimizing the need to perform checkpointing — possibly at the cost of more re-execution. The second one involves improving the scalability of the happened-before clocks and RDM design.

7. ACKNOWLEDGMENTS

We thank the anonymous reviewers and the I-ACOMA group members for their comments. This work was supported in part by the National Science Foundation under grants CNS-0720593 and CCR-0325603; Intel and Microsoft under the Universal Parallel Computing Research Center; and gifts from IBM and Sun Microsystems. Suárez was supported by the Gobierno de Aragón under grant "gaZ: Grupo Consolidado de Investigación"; Spanish Ministry of Education and Science under contracts TIN2007-66423 and Consolider CSD2007-00050; and European Union Network of Excellence HiPEAC-2 (FP7/ICT 217068).

8. REFERENCES

[1] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *International Conference on Parallel Architectures and Compilation Techniques*, October 2008.

[2] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Comm. of the ACM*, 13(7):422–426, 1970.

[3] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. BulkSC: Bulk enforcement of sequential consistency. In *International Symposium on Computer Architecture*, June 2007.

[4] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval. Bulk disambiguation of speculative threads in multiprocessors. In *International Symposium on Computer Architecture*, June 2006.

[5] 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 *Programming Language Design and Implementation*, June 2002.

[6] L. Dagum and R. Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science & Engineering*, 5(1):46–55, 1998.

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

[8] Intel Corporation. Intel Thread Checker. http://www.intel.com, 2008.

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

[10] C.-K. Luk et al. Pin: Building customized program analysis tools with dynamic instrumentation. In *Programming Language Design and Implementation*, June 2005.

[11] E. Lusk, J. Boyle, R. Butler, T. Disz, B. Glickfeld, R. Overbeek, J. Patterson, and R. Stevens. *Portable programs for parallel processors*. Holt, Rinehart & Winston, 1988.

[12] S. L. Min and J.-D. Choi. An efficient cache-based access anomaly detection scheme. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.

[13] C. C. Minh et al. An effective hybrid transactional memory system with strong isolation guarantees. In *International Symposium on Computer Architecture*, June 2007.

[14] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically classifying benign and harmful data races using replay analysis. In *Programming Language Design and Implementation*, June 2007.

[15] R. H. B. Netzer and B. P. Miller. Detecting data races in parallel program executions. In *In Workshop on Advances in Languages and Compilers for Parallel Computing*, 1990.

[16] R. H. B. Netzer and B. P. Miller. Improving the accuracy of data race detection. In *Principles and Practice of Parallel Programming*, April 1991.

[17] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *Principles and Practice of Parallel Programming*, June 2003.

[18] M. Prvulovic. CORD: Cost-effective (and nearly overhead-free) order-recording and data race detection. In *International Symposium on High-Performance Computer Architecture*, February 2006.

[19] M. Prvulovic and J. Torrellas. ReEnact: Using thread-level speculation mechanisms to debug data races in multithreaded codes. In *International Symposium on Computer Architecture*, June 2003.

[20] M. Prvulovic, Z. Zhang, and J. Torrellas. ReVive: Cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In *International Symposium on Computer Architecture*, May 2002.

[21] F. Qin, J. Tucek, Y. Zhou, and J. Sundaresan. Rx: Treating bugs as allergies—a safe method to survive software failures. *ACM Transactions on Computer Systems*, 25(3):7, 2007.

[22] M. Ronsse and K. De Bosschere. RecPlay: A fully integrated practical record/replay system. *ACM Transactions on Computer Systems*, 17(2):133–152, 1999.

[23] D. Sanchez, L. Yen, M. D. Hill, and K. Sankaralingam. Implementing signatures for transactional memory. In *International Symposium on Microarchitecture*, December 2007.

[24] S. Savage et al. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[25] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *USENIX Annual Technical Conference*, June 2004.

[26] Sun Microsystems. Sun Studio Thread Analyzer. http://developers.sun.com/sunstudio, 2007.

[27] C. von Praun and T. R. Gross. Object race detection. In *Object-Oriented Programming, Systems, Languages, and Applications*, October 2001.

[28] L. Yen et al. LogTM-SE: Decoupling hardware transactional memory from caches. In *International Symposium on High Performance Computer Architecture*, February 2007.

[29] Y. Yu, T. Rodeheffer, and W. Chen. RaceTrack: Efficient detection of data race conditions via adaptive tracking. In *Symposium on Operating Systems Principles*, October 2005.

[30] P. Zhou, R. Teodorescu, and Y. Zhou. HARD: Hardware-assisted lockset-based race detection. In *International Symposium on High Performance Computer Architecture*, February 2007.