# HARD: Hardware-Assisted Lockset-based Race Detection

Pin Zhou[§†*], Radu Teodorescu[†] and Yuanyuan Zhou[†]

[†] University of Illinois at Urbana-Champaign
[§] IBM Almaden Research
pinzhou@us.ibm.com, {teodores,yyzhou}@cs.uiuc.edu

## Abstract

*The emergence of multicore architectures will lead to an increase in the use of multithreaded applications that are prone to synchronization bugs, such as data races. Software solutions for detecting data races generally incur large overheads. Hardware support for race detection can significantly reduce that overhead. However, all existing hardware proposals for race detection are based on the happens-before algorithm which is sensitive to thread interleaving and cannot detect races that are not exposed during the monitored run. The lockset algorithm addresses this limitation. Unfortunately, due to the challenging issues such as storing the lockset information and performing complex set operations, so far it has been implemented only in software with 10-30 times performance hit.*

*This paper proposes the first hardware implementation (called HARD) of the lockset algorithm to exploit the race detection capability of this algorithm with minimal overhead. HARD efficiently stores lock sets in hardware bloom filters and converts the expensive set operations into fast bit-wise logic operations with negligible overhead. We evaluate HARD using six SPLASH-2 applications with 60 randomly injected bugs. Our results show that HARD can detect 54 out of 60 tested bugs, 20% more than happens-before, with only 0.1–2.6% of execution overhead. We also show our hardware design is cost-effective by comparing with the ideal lockset implementation, which would require a large amount of hardware resources.*

## 1 Introduction

Multithreading is a common programming technique used in many server and scientific applications to achieve good performance. The emergence of multicore architectures will further strengthen the trend of multi-threaded programming. Unfortunately, despite the performance benefit, multithreading also significantly increases software complexity and is prone to synchronization bugs.

One of the most common synchronization bugs is the *data race*, which occurs when at least two threads access the same shared variable without synchronization, and at least one access is a write. Data races are notoriously difficult to expose, reproduce and diagnose due to their non-determinism and timing sensitivity. Therefore, data races can easily lurk into extensively tested programs and cause serious damage in production runs.

With the recent impressive advances in micro-architecture, interest has risen in the architecture community in using some of the available hardware budget to minimize the prohibitive bug detection overhead and improve the ease of software debugging [15, 27, 34, 18, 35, 17, 38, 37, 13, 26, 12]. A few studies [15, 27, 26] have proposed hardware support for efficient dynamic race detection. These systems significantly reduce the huge performance penalty of software-only approaches from 10–30 times slowdown [30] to an acceptable range, and, therefore, are suitable for the on-the-fly race detection in production runs.

Almost all previous hardware race detection proposals [15, 27, 26] are based on the happens-before algorithm [23, 7, 14, 27, 26]. This algorithm is based on Lamport's happens-before relation [11]. It dynamically monitors the program execution and partially orders the memory accesses based on synchronizations and execution order. A data race is reported if there is no temporal ordering between two conflicting accesses. The basic idea of the hardware implementations of this algorithm is to store access histories or timestamps in the hardware cache. The stored information is communicated through the underlying cache coherence protocol and used to check for any access anomalies related to the happens-before ordering.

A major limitation of the happens-before algorithm is that it can only detect those races that manifest during the monitored execution. Most races manifest only under some interleavings. Figure 1 shows such an example. The data race on $x$ will not be detected in the interleaving shown in Figure 1, because the accesses to $x$ are ordered by the lock

*The work was done when Pin Zhou worked at the University of Illinois at Urbana-Champaign. This research is supported by NSF CNS-0347854, NSF CCR-0325603 and DOE DE-FG02-05ER25688 grants.

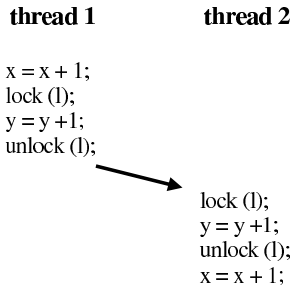



**Figure 1.** Happens-before cannot detect the data race on x in this execution interleaving.

operations performed for accessing $y$. It can be detected by happens-before only if the code fragment in thread 2 is executed before the code fragment in thread 1. Based on real world experience, many data races require more than tens even hundreds of repeated runs to manifest only once. Therefore, since it is impossible to exhaustively test every possible thread interleaving, the happens-before algorithm can easily miss many bugs because they are not exposed during the monitored run, as we demonstrate in our experimental results (see Section 5).

To overcome this limitation, researchers have proposed another algorithm called the lockset algorithm [30, 33, 5]. This algorithm is insensitive to thread scheduling and can catch data races that do *not* actually manifest during a particular execution. At run time, the lockset algorithm checks for violations of the locking discipline. One simple example of the locking discipline is that all accesses to the same shared variable should be protected by at least one common lock. To perform such checks, the algorithm maintains the set of locks currently held by a thread (called the thread *Lock Set*) for each thread, and the set of locks that have protected a variable so far (called the *Candidate Set*) for each shared variable. A lock is added or removed from a thread lock set when the thread acquires or releases the lock. The candidate set is initialized as all possible locks, and updated upon every access to the corresponding variable by intersecting with the thread lock set. An empty candidate set means no common locks protect the variable and, therefore, indicates a potential race.

Unfortunately, all existing implementations of the lockset algorithm are software-based and, as a result, introduce significant overheads. For example, Eraser [30] has reported a factor of 10–30 times slowdowns for some applications. To reduce this overhead, several recent studies have proposed various solutions for trading off bug detection accuracy for performance or focusing on object-oriented languages which can increase the monitoring granularity from variable to object. For example, RaceTrack [36] reduces the overhead for object-oriented programs to only 2-3 times slowdowns, but it cannot be applied to legacy code written in C and its overhead is still not low enough for production runs.

Despite many hardware implementations of the happens-before algorithm, no study has explored the possibility of implementing the lockset algorithm in hardware to take advantage of the unique bug detection power of this algorithm *with low overhead*. This is because implementing lockset in hardware is challenging and needs to address two main issues. The first issue is how to efficiently store and maintain candidate sets for variables in hardware. Unlike the happens-before algorithm whose access history and timestamps have fixed format and length, the candidate sets in the lockset algorithm have variable sizes, and each element in a set is a lock ID or address (2-4 bytes). Therefore, we need to derive a solution to represent a candidate set using a small and fixed-size structure in hardware without requiring applications to change their source code to use different lock primitives or limit the number of locks. The second issue is how to efficiently perform the set operations in the lockset algorithm. Such operations include adding or removing a lock from the thread lock set upon a lock acquire or release, and intersecting the lock set and the candidate set at every shared access. These expensive and frequently performed set operations are the main source of the huge performance penalty in exiting software implementations . To minimize the overhead, it is critically important to perform these set operations efficiently.

### 1.1 Our Contributions

This paper proposes the first hardware implementation (called HARD) of the lockset algorithm. The goal of HARD is to detect data races, including those that do not manifest during the monitored run, with little overhead. Essentially, HARD exploits the lockset algorithm's full bug detection capability with a very small overhead. This makes it suitable for production runs. It is also sufficiently general that it can be applied to applications written in most programming languages (C/C++, Java, etc).

HARD efficiently stores the candidate sets with variable sizes in hardware bloom filters [2] using only 16-bit long vectors. The expensive set operations can then be converted to fast bitwise logic operations, which can be performed very efficiently in hardware with negligible overhead. The candidate sets are communicated among processors by piggybacking on cache coherence protocol messages to minimize inter-processor traffic.

Additionally, to reduce false positives, HARD goes one step further than previous lockset work by proposing a technique to handle barriers, which are widely used in many parallel applications. Our technique resets the bloom filter vectors of all variables after exiting a barrier. This effectively reduces the number of false positives caused by barriers.

We evaluated HARD using six SPLASH-2 benchmarks with 60 randomly injected data races. We also compare our lockset implementation with a happens-before implementation. Our results show that during the monitored runs (without selecting inputs and interleavings), HARD detects 54 out of 60 tested bugs, 20% more than happens-before, with only 0.1–2.6% of execution overhead. We also show our default hardware design is cost-effective by comparing with the ideal lockset hardware implementation which would require a large amount of hardware resource.

This paper is organized as follows. Section 2 describes the lockset algorithm. Section 3 presents the design details of HARD. Sections 4 and 5 present the evaluation methodology and experimental results. Section 6 discusses related work, followed by Section 7 which concludes this paper.

# 2 Background: The Lockset Algorithm

## 2.1 Basic Lockset Algorithm

The lockset algorithm was first proposed in [30]. It detects data races by dynamically checking that all shared-memory accesses follow a locking discipline. The simplest discipline is that accesses to every shared variable should be protected by some common lock. By monitoring all shared reads, writes and lock acquire and release primitives as the program executes, it can detect any violations of the locking discipline that occur when accessing shared objects.

For each shared variable $v$, the lockset algorithm maintains a set of locks that have protected $v$ so far in a *candidate set* associated with $v$, or $C(v)$. For each thread $t$, the lockset algorithm also maintains a set of locks currently held by thread $t$ in the *lock set* of $t$, or $L(t)$. A lock $l$ is added to or removed from the thread lock set $L(t)$ when thread $t$ acquires or releases the lock $l$. When a new variable $v$ is initialized, its candidate set $C(v)$ holds all possible locks. When the variable is accessed by thread $t$, $C(v)$ is updated with the intersection of $C(v)$ and the thread's current lock set $L(t)$. This will ensure that any lock that always protects $v$ will be contained in $C(v)$. If $C(v)$ is empty, there is no single lock protecting $v$, and this indicates a potential race.

Implementing lockset entirely in software is expensive. It needs to instrument the lock acquire and release primitives to update the lock set for each thread. It also needs to maintain a candidate set table which stores the candidate set for each variable. More specifically, every access is instrumented such that when a variable $v$ is accessed, its candidate set $C(v)$ will be found by searching the table. Then $C(v)$ will be updated by intersecting $C(v)$ and the running thread's lock set $L(t)$. Finally, the new $C(v)$ will be checked and if it is empty, a potential race is indicated. Such heavy-weight monitoring at such fine granularity (every access to a shared variable) introduces a severe performance hit in software-only implementations, slowing down applications by up to 30 times [30].

## 2.2 Existing False Alarm Pruning Techniques for Lockset

The above basic lockset algorithm can result in many false positives. This is because sometimes the programmer will intentionally access certain shared memory locations without using locks, if he knows that races cannot occur. One such case is the initialization of shared variables. It is a common programming practice that accessing shared variables in initialization phase without lock protection. This is generally safe because only the initializing thread has a reference to that data. Another case is when shared variables are written during initialization and are read-only afterward. These variables can be safely accessed without holding any locks.

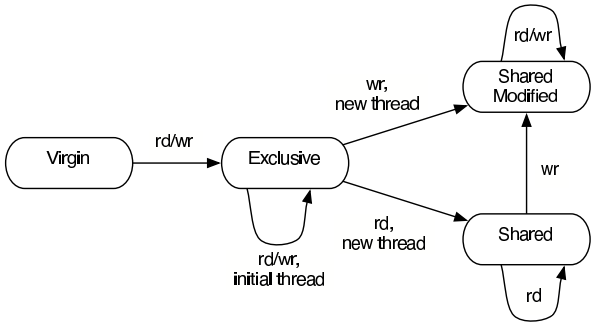



**Figure 2.** State diagram for pruning false positives introduced due to variable initialization. All variables are initially in Virgin state. The state of the variables changes depending on the rd/wr accesses by different threads as shown in the diagram. A race is reported for a variable only if it is currently in the Shared-Modified state.

In order to filter out these cases, lockset defines multiple states for each memory location. Figure 2 shows the four states a variable can be in and the actions that can trigger transitions. A newly allocated variable $v$ is set to the Virgin state. When it is first accessed, it transitions to the Exclusive state. As long as $v$ continues to be accessed by the same thread, it will remain Exclusive. While in Exclusive state, its candidate set $C(v)$ will not be updated, therefore no races will be reported. This will ensure that variables initialized by a single thread will not trigger false alarms, even if accessed without holding locks.

If $v$ is later accessed by a different thread, its state will change either to Shared if it is a read access, or to Shared-Modified if it is a write access. The Shared state indicates that the variable $v$ was initialized and then has only been read by threads. Accessing $v$ without a lock at this point is safe. Therefore, to avoid false positives, its candidate set $C(v)$ will be updated, but no races will be reported. In contrast, the Shared-Modified state indicates that $v$ has been written and read by multiple threads, therefore its candidate

set $C(v)$ will be updated and potential races will be reported if any.

# 3 Hardware Implementation of Lockset Detector

To implement the lockset algorithm in hardware, the following two major issues need to be addressed. First, how to efficiently store and maintain candidate sets for variables in hardware? Second, how to efficiently perform the set operations required by the lockset algorithm, including addition, deletion and intersection?

## 3.1 Overview

To efficiently represent candidate sets with variable sizes and provide fast set operations, HARD uses bloom filters for the candidate sets of all variables and the current lock sets of all threads.

Figure 3 gives an overview of the HARD design for a CMP architecture using a snoopy-based coherence protocol. Each L1 cache line is augmented with two bits called *LState* to record the states used in the lockset algorithm for false positive pruning (see Section 2), and a bloom filter vector (*BFVector*) to store the candidate set of this line. Each L2 cache line also records the candidate sets and LStates for the corresponding L1 lines. Note that, the LState is different from the coherence state (CState) used in the coherence protocol. The size of the BFVector is 16 bits, which introduces 1/16 overhead for a cache with 32 Byte cache lines.

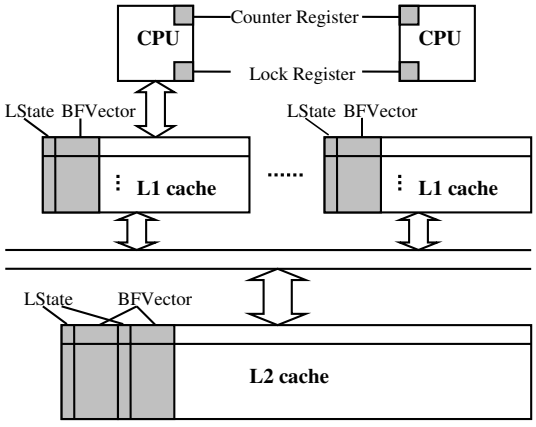



**Figure 3.** Design Overview of HARD. The L2 line size is twice of the L1 line size.

Fetching a line from memory will initialize its candidate set to all possible locks by setting all bits of the BFVector to 1, and initialize its LState to Exclusive. The BFVector for the candidate set and the LState will be updated on each access according to the lockset algorithm (Section 2). The BFVector and LState for each line are kept consistent among processors by the underlying coherence protocol, as they are part of the data content of the corresponding line (Section 3.4).

Inside each processor, we add two special registers, a *Lock Register* and a *Counter Register* for storing the lock set of the running thread. The 16-bit Lock Register stores the union of the bloom filter vectors of all the lock addresses currently held by the processor. Representing both the candidate set and lock set using bloom filter vectors makes adding and removing locks from the set simple and fast. The 32-bit Counter Register stores a set of 2-bit counters where each counter is associated with one bit in the Lock Register. It is used to support removing a lock from the Lock Register in case of possible hash collisions (Section 3.3).

Because the lockset algorithm only handles lock-based synchronization, it generates spurious race reports if the program uses other synchronization primitives, such as fork/join, barriers etc. Previous studies proposed some partial solutions to handle fork (or start) by using the ownership model [33] and to handle join by using dummy locks [4]. These solutions can be incorporated into HARD as well. For barriers which are commonly used in scientific applications, we propose a technique to prune the false positives by resetting the BFVectors of all variables after exiting the barrier (details see Section 3.5).

## 3.2 Candidate Sets and Lock Sets with Bloom Filter

The bloom filter was first proposed by Bloom [2] to support fast membership testing of a set. It uses multiple hash functions to map an element into a bit vector. For each member element, its corresponding bits in the vector are set to 1. To test whether an element is a member or not, its corresponding bits based on the hash function are tested. If one of the bits is 0, the element does not belong to the set. Otherwise, the element belongs to the set (assuming no hash collisions).

We use bloom filters to represent both the candidate set and the lock set for two main reasons. First, BFVectors use a small fixed number of bits to store sets with variable sizes (each element of a set is a 2-4Byte lock ID or address). Second, it provides very fast set operations, such as set membership, intersection, addition and deletion, which are key operations frequently performed by the lockset algorithm. For instance, computing the intersection of the candidate set and lock set is as simple as performing a bitwise logic AND of the corresponding BFVectors.

Figure 4 shows how we map a lock address to a bloom filter vector. The size of the BFVector is 16 bits. For each lock address in the set, 8 bits (bit 2 to bit 9, starting from the least significant bit) are used to map this address to a bloom filter vector. The 8 bits are broken into 4 parts, with 2 bits each. Each part is used to directly index 4 bits in the bloom filter vector. All indexed bits are set to 1. This partial

address indexing idea was also used in [22, 37]. We use a direct index instead of a more complex hash function to simplify the hardware logic as much as possible.

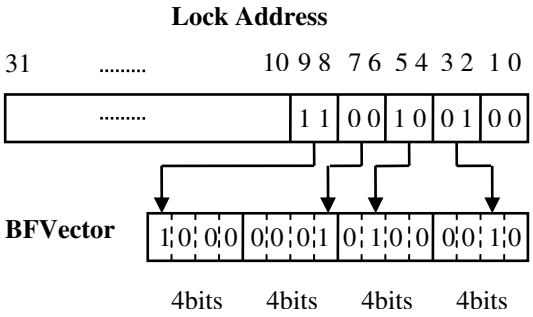



**Figure 4.** Map a Lock Address to a bloom filter.

For a set, if there is at least one bit having value 1 in each of its four bloom filter vector parts, the set is not empty. Otherwise the set is empty. The bloom filter always correctly identifies an empty set. However, due to the hash collisions, it could mistakenly identify a set as not empty. Recall that an empty candidate set means a data race. Therefore, the use of the bloom filter could potentially cause missing some data races as we show in Figure 5. Figure 5(a) shows the candidate set for variable $v$, $C(v) = L1, L2$, and the corresponding BFVector. When $v$ is accessed by thread $t$, the lock set held by $t$ is $L(t) = L3$. $L(t)$ and its BFVector are shown in Figure 5(b). Now the new $C(v)$ should be $C(v) \cap L(t) = \phi$, however, due to the hash collision, the BFVector for the new $C(v)$ is not empty (Figure 5(c)), which will hide this data race.

Of course, if the vector is long enough, the probability of collision and thus missing races will be very low. However, we also have to minimize the vector length to reduce the hardware cost. Therefore, choosing an appropriate size for the bloom filter is critical. The guideline is to use the smallest vector size with acceptable missing race probability (e.g., $\leq$1%).

Now, the question becomes how to estimate the missing race probability. Assuming the vector is divided into 4 parts, the length of each part is $n$ ($n > 1$), the size of the candidate set size is $m$, and the lock addresses are randomly distributed. For each element in the lock set, the probability that it collides with one part of the BFVector of the $C(v)$ is $CR_{part} = 1 - (\frac{n-1}{n})^m$, and the probability of colliding with all four parts of the BFVector is $CR_{whole} = CR_{part}^4$. $CR_{whole}$ gives the false positive probability in membership testing (missing race probability in our case). For the set size $m = 1, 2, 3$ and the vector size of 16 (i.e., $n = 4$ for 4 parts), $CR_{whole} = 0.0039, 0.037, 0.111$, respectively. Because the sizes of candidate sets and lock sets are usually small in programs, we choose the vector size of 16. Our experiments show that no races were missed as a result of having a bloom filter vector of size 16.

### 3.3 Counters for Thread Lock Sets

In addition to using a bloom filter (Lock Register) to represent the current lock set of a thread, we also maintain a Counter Register consisting of 2-bit counters for each bit in the Lock Register.

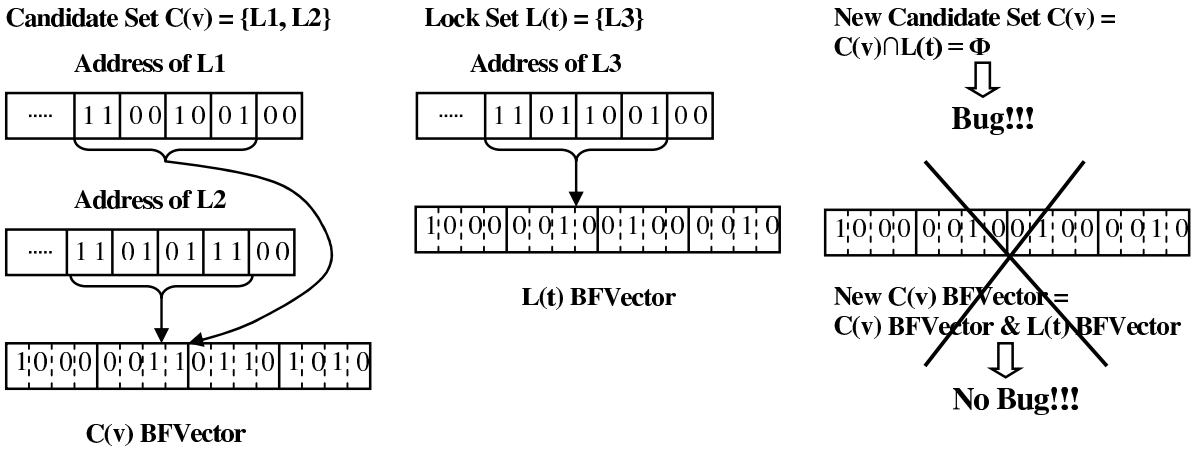
The lock set of a thread is updated when a lock or unlock operation is performed during execution. When a lock is acquired, the new lock needs to be added to the lock set. The addition is a bitwise logic OR of the original lock set BFVector (Lock Register) and the new lock BFVector. When a lock is released, it needs to be removed from the lock set. Updating the BFVector of the lock set becomes a problem, because we cannot just reset all the bits corresponding to the lock to 0 because of possible hash collisions. Otherwise, we may remove some bits belonging to other locks.

We solve this problem by using the Counter Register, consisting of 16 2-bit counters associated to the lock set. Whenever adding a lock, the corresponding bits in the BFVector are set to 1, and the corresponding counters are increased by 1 until they are saturated. When removing a lock, the corresponding counters from the counter register are decreased by 1, and the corresponding bits in the BFVector are only reset if the counters reach 0. 2-bit counters are sufficient because the size of lock set is small, and the conflicts will be rare.

### 3.4 Candidate Set and LState Communication

In our design, we assume a snoopy-based coherence protocol, thus we store the candidate set and LState at each cache line. Along with the cache data, the candidate set and LState are communicated among the processors through the cache coherence protocol. When a cache line is transferred as a result of a coherence request, the associated candidate set and LState are also sent to the requesting processor. The requesting processor performs the intersection of its lock set with the received candidate set to generate the new candidate set, and update the LState based on the state transition (Figure 2).

In addition to the above extension, the coherence protocol also needs to support the following requirement. When a processor reads a cache line that is in Shared CState after this read, if the newly computed candidate set is different from the old one, the new candidate set and the LState should be broadcast to other processors and the L2 cache. Other L1 caches that hold this line and the L2 cache should snoop this message to update their candidate sets and LStates for this line. This is used to keep the candidate sets and LStates in all valid L1 cache lines and L2 cache lines consistently up-to-date. Figure 6 graphically represents this process. In our experiments, such broadcast happens not very often because the candidate set of a variable usually does not change after a few accesses to this variable.




(a) Candidate Set (b) Lock Set (c) New Candidate Set

**Figure 5.** A False Negative Caused by the Bloom Filter.

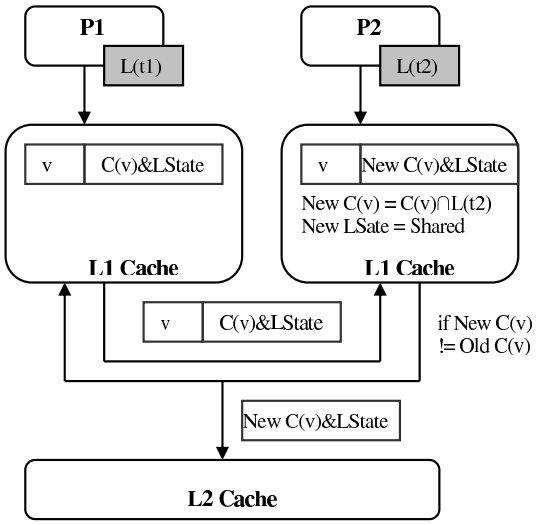



**Figure 6.** Processor P1 responds to a read request for line v by processor P2 by sending the line, the candidate set C(v) and the LState. Processor P2 will use C(v) and its Lock set for current thread t2 – L(t2) to compute the new C(v), and get the new LState (should be Shared here) based on the transition graph. The new C(v) and LState will be broadcast to other L1 caches and L2 cache, if C(v) changes.

Sending the candidate set and LState on the bus or through the network can increase the coherence traffic, which introduces some performance overhead as shown in our experimental results. However, since we store the candidate set and LState in only 18 bits, the amount of extra traffic and its performance impact is small.

For a directory-based protocol, the candidate set and the LState are stored in the directory instead of together with each cache line. Every shared access gets the candidate set and LState information from the directory, and then puts the new information back. The management of the candidate set and LState is simpler in a directory-based protocol. However, even for a local cache access, the processor needs to get the lockset related information from the directory, instead of from the cache directly. This can be done on the background, but may delay the detection of races.

### 3.5 False Positive Pruning for Barriers

Barriers are commonly used in scientific applications (e.g., Splash-2 applications). They cause many false positives in the lockset algorithm, because the accesses from different threads to a variable can be ordered by barriers, then there is no need to guard such variable using locks. Figure 7 shows an example. Before the barrier, array A[0..7] is read and written by thread t1 only. After the barrier, only thread t2 reads and writes array A. Even though during the execution, array A is accessed by both threads t1 and t2, all these accesses are not protected by locks. The code is race free because there are no concurrent accesses to array A (the access order is enforced by the barrier), but the lockset algorithm will normally report races among these accesses.

To eliminate the false alarms caused by barriers, we set the candidate set BFVectors of all variables (representing all possible locks) after exiting a barrier. This way, the accesses and their lock information before the barrier are discarded, because these accesses and the accesses after the barrier to the same variable have the happens-before relation. This approach is only an approximation. It would be accurate if all threads went through the same barrier. Fortunately, this is indeed the common case. In our experiments, all the benchmarks that use barriers follow this pattern. If different groups of threads go through different barriers, or some threads do not go through the barrier, this approach may

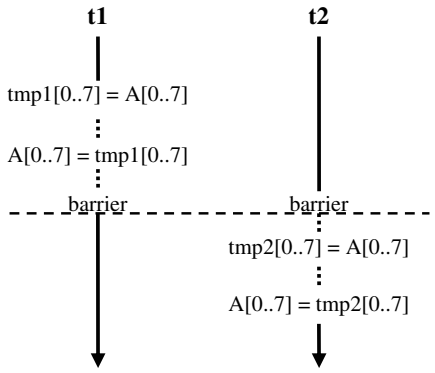



**Figure 7.** A False Positive Caused by the Barrier.

| Core Parameters | |
|---|---|
| CPU frequency | 2.4GHz |
| Int, Mem, FP FUs | 3, 2, 2 |
| BTB | 2K, 2 way |
| ROB, I-window sizes | 128, 64 |
| Fetch, Issue, Retire widths | 6, 4, 4 |
| LD, ST queue entries | 64, 48 |
| L1 cache | 16KB, 4-way, 32B/line, 3 cycle latency, 16b BFVector/line |
| **Memory System Parameters** | |
| L2 cache | 1MB, 8-way, 32B/line, 10 cycles latency, 16b BFVector/line |
| Memory | 200 cycles latency |

**Table 1.** Parameters of the simulated architecture.

cause false negatives because not all discarded accesses can be ordered by the same barrier. To handle this situation, we can combine HARD with the happens-before algorithm as in [36, 21, 25], which remains as our future work.

### 3.6 Other Design Issues

**Cache Displacement:** HARD stores the candidate set at each cache line. In our current implementation, when a shared line is displaced from the L2 cache, its candidate set information is lost. Since the L2 cache is typically a few megabytes large, keeping the candidate set only in the cache provides a detection window that is hundreds of thousand of instructions large, before lines have to be evicted back to the memory. This detection window is usually enough because most races occur within a short window of execution.

**False Sharing:** Candidate sets are stored at L1 cache line granularity. If an L1 line contains multiple shared variables, the algorithm requires these variables to be protected by common locks, which is unnecessary. Thus, the false sharing of the candidate sets can lead to false positives, an issue also faced by previous work [12, 27]. This problem can be mitigated by using a smarter parallelizing compiler that will not allocate multiple shared variables to the same line. This can also has important performance benefits.

## 4 Experimental Methodology

Our experiments are conducted using SESC [28], a cycle-accurate execution-driven simulator, that models a 4-core CMP augmented with the HARD functionality. Our simulator provides a detailed model of a CMP with out-of-order processor cores, detailed memory system and bus traffic. The parameters of the architecture are shown in Table 1.

We run six lock-based SPLASH-2 applications with randomly and dynamically injected data race bugs[1]. For each application, we *randomly* inject a single *dynamic* instance of a data race into each run of the application. This is done by omitting a randomly selected dynamic instance of a lock primitive and the corresponding unlock primitive. For each application, we use the test input set and perform 10 runs, each time injecting different data races.

To demonstrate the advantages of lockset over happens-before, we also implement the happens-before algorithm in our simulator, and compare the functionality of these two algorithms. For the happens-before implementation, we store the timestamps at cache-line granularity, very similar to storing the candidate sets and LStates in HARD. Note that, for some shared variables, not all injected bugs manifest in the underlying thread interleaving as an actual *race*.

As described in Section 3, due to the space limitations for maintaining the candidate sets, HARD does not provide an exact implementation of the lockset algorithm. Instead, it makes three approximations: (1) maintaining candidate sets at cache-line granularity instead of variable granularity; (2) using a bloom filter vector instead of a complete set representation; (3) only maintaining candidate sets for data in the cache. To show that these three approximations do not significantly affect its races detection capabilities, we also compare HARD with an ideal implementation of lockset. In this ideal implementation, we maintain the candidate set at variable granularity for all variables using complete set representation, as in software implementations of the lockset algorithm [30]. Similarly, our happens-before implementation makes two of the three approximations (1 and 3), and we also build the ideal one that maintains the timestamps at variable granularity for all variables.

To study the effects of different cache and bloom filter configurations for HARD, we vary the granularity of storing candidate sets and LStates from 4B to 32B, the L2 cache size from 128KB to 1MB, and the bloom filter vector size from 16 bits to 32 bits.

## 5 Experimental Results

### 5.1 Overall Results

Table 2 compares the effectiveness of HARD with that of a happens-before implementation. For each application, we

[1]Since almost all the remaining SPLASH-2 applications hardly use locks for synchronizations, we do not select them in our evaluation.

| Application | HARD | | | | Happens-before | | | |
|---|---|---|---|---|---|---|---|---|
| | default | | ideal | | default | | ideal | |
| | # of Bug Detected | # of False Alarms | # of Bug Detected | # of False Alarms | # of Bug Detected | # of False Alarms | # of Bug Detected | # of False Alarms |
| cholesky | 9/10 | 91 | 10/10 | 38 | 6/10 | 37 | 10/10 | 13 |
| barnes | 10/10 | 54 | 10/10 | 20 | 10/10 | 41 | 10/10 | 18 |
| fmm | 8/10 | 73 | 10/10 | 40 | 7/10 | 70 | 8/10 | 36 |
| ocean | 8/10 | 62 | 10/10 | 1 | 8/10 | 62 | 10/10 | 1 |
| water-nsquared | 9/10 | 5 | 10/10 | 0 | 5/10 | 0 | 6/10 | 0 |
| raytrace | 10/10 | 48 | 10/10 | 2 | 8/10 | 36 | 8/10 | 0 |

**Table 2.** Overall results: the effectiveness of HARD and a happens-before implementation. "Ideal" means the ideal lockset and happens-before implementations described in Section 4.

inject 10 races (10 dynamic instances of missing locks) in 10 runs, one for each run (see Section 4). We compare the race detection effectiveness of HARD with that of happens-before using identical executions. We use the race-free execution (without injected any bugs) to measure the false positive rate.

As shown in Table 2, with the default configuration, HARD can detect more races than happens-before. All missed races are caused by losing the candidate set information due to L2 cache displacement. The 16-bit BFVectors do not cause missing races, as shown later in Table 6. Columns 4 and 8 of Table 2 show that with more hardware resources, all these races can be detected by HARD, while some are still missed by happens-before.

Happens-before suffers from its sensitivity to thread interleaving. It requires special thread interleavings in order to detect a race, so it detects 20% fewer bugs than HARD in the default setup. For example, it misses 3, 5 and 2 bugs in fmm, water-nsquared and raytrace, respectively. In contrast, HARD is insensitive to thread interleaving and can detect more bugs (actually all of them with the ideal hardware resources).

To estimate the number of false positives, we map the reported races back to the source code, and the number of false positives is counted at source code level. Therefore, each false alarm could contain many dynamic instances of false races. Our results show the false alarm rate of HARD with the default setup is noticeably higher than happens-before for three applications. This is because the lockset algorithm only handles lock-based synchronization. Even with our false alarm pruning for barriers, other synchronizations beyond locks and barriers can still introduce false alarms in HARD. Since happens-before can handle all kinds of synchronizations, it does not have this problem. For both HARD and happens-before, the common sources of false alarms in the ideal setup are mainly hand-crafted synchronizations and benign races. In the default setup, another major source of false alarms in both HARD and happens-before is the false sharing of the candidate set, LState and timestamps within a cache line. For applications like cholesky, barnes, fmm, ocean, raytrace, the number of false alarms caused by false sharing is significant.

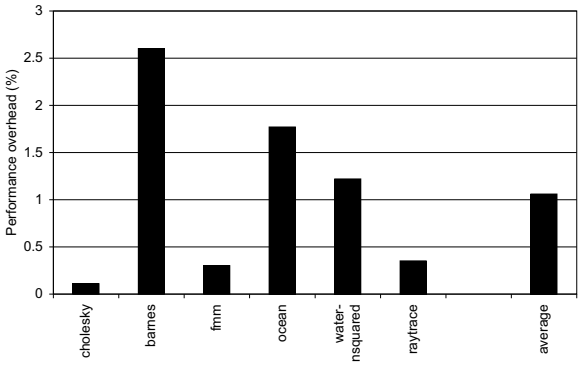



**Figure 8.** The performance overhead of HARD as percentages of the original execution time without HARD.

Figure 8 shows the performance overhead of HARD. We can see that HARD's overhead is relatively small and ranges from 0.1% to 2.6% for all applications. This overhead comes from three main sources. The first is the increased bus traffic caused by communicating the candidate set and LState information. The second one is the longer access time to the shared variables due to computing and checking the new candidate set for each shared access. The third is the overhead of updating the thread lockset upon lock and unlock primitives. Of the three, the bus traffic increase is the main contributor to the performance degradation observed. We believe the overhead is still very low and would allow the deployment of HARD in production systems.

## 5.2 Sensitivity Analysis

This section measures the effects of candidate set and LState granularity, L2 cache size and Bloom filter vector size on HARD's functionality in terms of both the number of detected bugs and false alarms. We then compare each configuration with the ideal case (4B granularity, $\infty$

| Application | # of Bug Detected | | # of False Alarms | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | HARD | Happens-before | HARD | | | | Happens-before | | | |
| | 4-32B | 4-32B | 4B | 8B | 16B | 32B | 4B | 8B | 16B | 32B |
| cholesky | 9 | 6 | 25 | 36 | 74 | 91 | 3 | 10 | 17 | 37 |
| barnes | 10 | 10 | 20 | 25 | 43 | 54 | 18 | 25 | 25 | 41 |
| fmm | 8 | 7 | 40 | 40 | 53 | 73 | 36 | 39 | 49 | 70 |
| ocean | 8 | 8 | 1 | 1 | 2 | 62 | 1 | 2 | 50 | 62 |
| water-nsquared | 9 | 5 | 0 | 0 | 2 | 5 | 0 | 0 | 0 | 0 |
| raytrace | 10 | 8 | 2 | 9 | 31 | 48 | 0 | 5 | 18 | 36 |

**Table 3.** Effectiveness of HARD and happens-before with different monitoring granularities.

| Application | # of Bug Detected in HARD | | | | # of Bug Detected in Happens-before | | | |
|---|---|---|---|---|---|---|---|---|
| | 128KB | 256KB | 512KB | 1MB | 128KB | 256KB | 512KB | 1MB |
| cholesky | 6 | 8 | 9 | 9 | 5 | 6 | 6 | 6 |
| barnes | 9 | 10 | 10 | 10 | 10 | 10 | 10 | 10 |
| fmm | 7 | 7 | 8 | 8 | 6 | 7 | 7 | 7 |
| ocean | 7 | 8 | 8 | 8 | 8 | 8 | 8 | 8 |
| water-nsquared | 8 | 9 | 9 | 9 | 5 | 5 | 5 | 5 |
| raytrace | 8 | 9 | 10 | 10 | 7 | 8 | 8 | 8 |

**Table 4.** Number of bugs detected by HARD and happens-before, for different L2 cache sizes.

| Application | # of False Alarms in Lockset | | | | # of False Alarms in Happens-before | | | |
|---|---|---|---|---|---|---|---|---|
| | 128KB | 256KB | 512KB | 1MB | 128KB | 256KB | 512KB | 1MB |
| cholesky | 49 | 78 | 81 | 91 | 31 | 31 | 35 | 37 |
| barnes | 52 | 54 | 54 | 54 | 39 | 39 | 41 | 41 |
| fmm | 73 | 73 | 73 | 73 | 68 | 70 | 70 | 70 |
| ocean | 60 | 62 | 62 | 62 | 52 | 58 | 60 | 62 |
| water-nsquared | 5 | 5 | 5 | 5 | 0 | 0 | 0 | 0 |
| raytrace | 48 | 48 | 48 | 48 | 34 | 34 | 34 | 36 |

**Table 5.** Number of false alarms of HARD and happens-before for different L2 cache sizes.

L2 cache, and storing accurate information instead of using a bloom filter hash).

#### 5.2.1 Varying the Granularity of Candidate Sets and LStates

In the first experiment, we vary the granularity of the candidate set and LState for HARD and the timestamps for happens-before from 4B to 32B, while keeping the other parameters identical to the default setup. The number of detected bugs and false alarms of HARD and happens-before for different granularities are shown in Table 3. For both HARD and happens-before, since the granularity will only affect the number of false alarms caused by false sharing, the number of detected bugs remains the same across all configurations. Also, for both HARD and happens-before, the number of false alarms is increasing as the granularity increases, because the false sharing increases when the granularity going from 4B to 32B.

#### 5.2.2 Varying L2 Cache Size

In the second experiment, we vary the size of the L2 cache from 128KB to 1MB, while keeping the other settings just like the default setup. The number of detected bugs and the number of false alarms are shown in Tables 4 and 5, respectively.

As shown in Table 4, the number of detected bugs increases slightly with the L2 cache size. This is because the number of L2 displacements decreases, and thus the possibility of missing bugs is lower. However, because the footprint of the applications is fairly small, the effect of different L2 sizes is not very significant. Moreover, in order to detect a race, we do not need to catch all access violations. Missing some violations (e.g., due to L2 displacement) is fine as long as we can catch at least one violation. This further weakens the effect of the L2 size.

The trend for the number of false alarms is also increasing for L2 sizes from 128KB to 1MB (Table 5). For the ideal

setup, although it has an infinite L2 size, its false alarm rate is the least, since it keeps candidate sets at a granularity of 4B instead of 32B. The small granularity significantly reduces the number of false alarms.

#### 5.2.3 Varying Bloom Filter Size

In the third set of experiments, we vary the size of the bloom filter vector from 16 bits to 32 bits, while keeping the other configurations the same as the default. We only show the number of detected bugs and false alarms for HARD in Table 6, because happens-before does not use a bloom filter.

| Application | # of Bug Detected | # of False Alarms | |
|---|---|---|---|
| | 16/32b | 16b | 32b |
| cholesky | 9 | 91 | 91 |
| barnes | 10 | 54 | 54 |
| fmm | 8 | 73 | 73 |
| ocean | 8 | 61 | 62 |
| water-nsquared | 9 | 5 | 5 |
| raytrace | 10 | 48 | 48 |

**Table 6.** Effectiveness of HARD with different BFVector sizes.

As we can see, using 16-bit and 32-bit bloom filters can detect the same number of bugs for all tested applications. This is because the applications' candidate sets and lock sets are usually small. For all test applications, the maximum sizes of candidate sets and lock sets are 1, except radix which has maximum candidate set size and lock set size of 3. Therefore, the conflict probability in the 16-bit bloom filter is very small based on our analysis (Section 3). The ideal setup can detect more races, but not because of bloom filter sizes, but because of the infinite L2 cache.

In terms of false alarms, the 32-bit and 16-bit bloom filters are almost the same except for ocean in which the 16-bit bloom filter has one less false alarm. This is because a hash conflict hides this false alarm. Similar to the L2 size, the larger bloom filter will not only catch more races, but can also generate more false alarms.

## 6 Related Work

### 6.1 Race Detection

Data race detection has been intensively studied. Previous work in this area can be classified into four main categories: dynamic analysis, post-mortem analysis, static analysis, and model checking.

Dynamic data race detection tools rely on program instrumentation or hardware support to monitor memory accesses and synchronization operations. Some of the most widely used dynamic detection algorithms include lockset, happens-before, and a combination of the two.

The lockset algorithm was first introduced in Eraser [30] and it verifies that every shared variable is protected by at least one lock. If not, it means there exists the potential for a data race to occur. More details on the algorithm are given in Section 2. In [33], the checker detects races at object granularity rather than at the level of individual variables for Java programs, which gives better performance than Eraser. However, the coarser granularity leads to many false positives. Choi et al. [5] proposed a weaker-than relation to identify redundant accesses from the viewpoint of data race detection to reduce overhead.

The happens-before algorithm is based on Lamport's happens-before relation [11] which combines program order and synchronization events to establish a partial temporal ordering of instructions. A data race occurs when a temporal ordering between two conflicting memory accesses cannot be established. Several previous studies [23, 7, 14] are based on happens-before. Task Recycling [7] maintains long memory access histories to verify that the happens-before relation holds. This results in very significant space overheads. Although abbreviated histories can be used for approximations, this results in a higher rate of false negatives. In contrast, the technique proposed in [14] for programs with nested fork-join parallelism limits the length of each variable's history list to a small constant, yet ensures a manifested race will always be detected.

Other studies [21, 25, 36] have proposed combinations of lockset and happens-before. The hybrid algorithm in [21] uses happens-before to reduce the false positives generated by using lockset alone, but still preserves the coverage of the lockset technique. MultiRace proposed in [25] also combines lockset and happens-before and detects races at the granularity of variables and objects. RaceTrack [36] also uses a hybrid approach in addition to being able to dynamically adjust monitoring granularity to improve performance.

Post-mortem methods [20, 19, 1] collect an execution log and analyze it to find races. The advantage of this approach is that it can be performed offline and has less impact on execution time. If the log contains sufficient information, both real and potential races can be found. However, the log is usually huge for a long execution.

Static race detection methods [32, 4, 9, 8] are based on compile-time analysis of the source code to find all possible data races in any possible execution of the program. Because static tools have access to the entire source code, they can perform a global analysis to try to prove the program is race-free. In general this technique conservatively reports all potential races, even some that can never occur in real executions. As a result, static techniques usually produce a large number of false positives.

Model checking is a formal verification technique. It exhaustively tests the model or code on all inputs by exploring state spaces. However, the requirement of a model or specification, and the large state spaces in most programs make it

hard to use. Both [6] and [10] use software model checking to detect data races.

### 6.2 Hardware Support for Debugging

The high performance cost of software-only debugging tools has prompted increased interest recently into hardware support for debugging. Several general frameworks aimed at dynamically detecting a wide range of bugs have been proposed, such as iWatcher [38], AccMon [37], and PathExpander [13]. Other researchers have proposed architectures that assist in low-overhead collection of execution traces to be used for postmortem bug detection [34, 18, 35, 17].

A few studies [15, 24, 29, 27, 26] have looked at hardware support for race detection. Interestingly, all proposals focused on variations of the happens-before algorithm. Min and Choi [15] use cache coherence protocol events to filter the number of calls to software monitors that check for races. Perkovic and Keleher [24] target race detection for distributed shared memory systems that use release consistency. Also in the context of distributed shared memory machines, Richards and Larus [29] propose extending the coherence protocol for on-the-fly race detection. ReEnact [27] extends the hardware support proposed for thread-level speculation to implement race detection based on happens-before. CORD [26] uses scalar clocks and timestamps for cost-effective order recording and data race detection. Besides the race detection work, AVIO [12] uses hardware support for detecting atomicity violations which also belong to synchronization bugs.

### 6.3 Other Related Work

Bloom filters [2] are frequently used in hardware to improve space and time efficiency. They are used to minimize load/store queue (LSQ) searches [31], to identify cache misses early in the pipeline [22], and to filter cache-coherence traffic in snoopy bus-based SMP systems to reduce energy consumption [16]. Bloom filters have also been employed for efficient disambiguation of memory accesses in speculative threads [3].

## 7 Conclusions and Future Work

This paper has presented HARD, an efficient hardware implementation of lockset-based race detection to detect more data races (compared to other hardware race detectors) with minimal overhead. It efficiently stores the candidate sets and speeds up the expensive and frequent set operations by using bloom filters. It also proposes a technique to prune the false alarms caused by barriers. We have evaluated HARD using six Splash-2 applications with 60 randomly injected races. HARD detects 54 out of 60 test bugs, 20% more than happens-before, with only 0.1–2.6% of execution overhead. Our hardware design is also cost-effective, because its bug detection capability is very close to the ideal lockset implementation, which would require large amount of hardware resource.

We are in the process of extending this work in two ways. First, we plan to evaluate HARD for more applications especially server programs, such as apache and mysql, and Java benchmarks like SPECjbb and SPECWeb. Second, we will combine with the happens-before algorithm to prune false alarms caused by other synchronizations. The combination is fairly straightforward, but requires more hardware resource. It will be challenging to minimize the hardware cost without losing any functionality.

## 8 Acknowledgments

We thank the anonymous reviewers for useful feedback. We thank Wei Liu for useful discussions and suggestions. We also thank Karin Strauss, Paul Sack, Luis Ceze and James Tuck for proofreading the paper.

## References

[1] S. V. Adve, M. D. Hill, B. P. Miller, and R. H. B. Netzer. Detecting data races on weak memory systems. In *Proceedings of the 18th International Symposium on Computer Architecture (ISCA)*, volume 19, pages 234–243, New York, NY, 1991. ACM Press.

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

[3] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval. Bulk disambiguation of speculative threads in multiprocessors. In *ISCA '06: Proceedings of the 33rd International Symposium on Computer Architecture*, pages 227–238, Washington, DC, USA, 2006. IEEE Computer Society.

[4] J. Choi, A. Loginov, and V. Sarkar. Static datarace analysis for multithreaded object-oriented programs. Technical Report RC22146, IBM Research, 2001.

[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 *Proceeding of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 258–269. ACM Press, 2002.

[6] X. Deng, M. B. Dwyer, J. Hatcliff, and M. Mizuno. Invariant-based specification, synthesis, and verification of synchronization in concurrent programs. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 442–452. ACM Press, 2002.

[7] A. Dinning and E. Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *PPOPP '90: Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*, pages 1–10. ACM Press, 1990.

[8] D. Engler and K. Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 237–252, New York, NY, USA, 2003. ACM Press.

[9] C. Flanagan, K. Leino, M. Lillibridge, C. Nelson, J. Saxe, and R. Stata. Extended static checking for java. In *Proceeding of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, 2002.

[10] T. A. Henzinger, R. Jhala, and R. Majumdar. Race checking by context inference. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 1–13, New York, NY, USA, 2004. ACM Press.

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

[12] S. Lu, J. Tucek, F. Qin, and Y. Zhou. Avio: Detecting atomicity violations via access interleaving invariants. In *Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006.

[13] S. Lu, P. Zhou, W. Liu, Y. Zhou, and J. Torrellas. PathExpander: Architectural Support for Increasing the Path Coverage of Dynamic Bug Detection. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2006.

[14] J. Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 24–33. ACM Press, 1991.

[15] S. L. Min and J.-D. Choi. An efficient cache-based access anomaly detection scheme. In *ASPLOS-IV: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, pages 235–244, New York, NY, USA, 1991. ACM Press.

[16] A. Moshovos, G. Memik, B. Falsafi, and A. Choudhary. Jetty: Filtering snoops for reduced energy consumption in SMP servers. In *Proceedings of the Seventh International Symposium on High Performance Computer Architecture (HPCA-7)*, 2001.

[17] S. Narayanasamy, C. Pereira, and B. Calder. Recording shared memory dependencies using strata. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 229–240, New York, NY, USA, 2006. ACM Press.

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

[19] R. Netzer and B. Miller. Detecting data races in parallel program executions. In *Advances in Languages and Compilers for Parallel Computing, 1990 Workshop*, pages 109–129, Irvine, Calif., 1990.

[20] R. H. B. Netzer. Race condition detection for debugging shared-memory parallel programs. Technical Report CS-TR-1991-1039, Univ. of Wisconsin-Madison, 1991.

[21] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 167–178. ACM Press, 2003.

[22] J.-K. Peir, S.-C. Lai, and S.-L. Lu. Bloom filtering cache misses for accurate data speculation and prefetching. In *Proceedings of the 16th Annual ACM International Conference on Supercomputing (ICS)*, 2002.

[23] D. Perkovic and P. J. Keleher. Online data-race detection via coherency guarantees. In *OSDI '96: Proceedings of the second USENIX symposium on Operating systems design and implementation*, pages 47–57. ACM Press, 1996.

[24] D. Perkovic and P. J. Keleher. Online data-race detection via coherency guarantees. In *OSDI '96: Proceedings of the second USENIX symposium on Operating systems design and implementation*, pages 47–57, New York, NY, USA, 1996. ACM Press.

[25] E. Pozniansky and A. Schuster. Efficient on-the-fly data race detection in multithreaded c++ programs. In *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 167–178. ACM Press, 2003.

[26] M. Prvulovic. CORD: Cost-effective (and nearly overhead-free) Order-Recording and Data race detection. In *HPCA*, Feb 2006.

[27] M. Prvulovic and J. Torrellas. ReEnact: Using thread-level speculation mechanisms to debug data races in multithreaded codes. In *ISCA*, June 2003.

[28] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos. SESC simulator, January 2005. http://sesc.sourceforge.net.

[29] B. Richards and J. R. Larus. Protocol-based data-race detection. In *SPDT '98: Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, pages 40–47, New York, NY, USA, 1998. ACM Press.

[30] 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(4):391–411, 1997.

[31] S. Sethumadhavan, R. Desikan, D. Burger, C. R. Moore, and S. W. Keckler. Scalabel hardware memory disambiguation for high ILP processors. In *Proceedings of the 36th Annual International Symposium on Microarchitecture (MICRO-36)*, Dec 2003.

[32] N. Sterling. Warlock: A static data race analysis tool. In *USENIX Winter Technical Conference*, 1993.

[33] C. von Praun and T. R. Gross. Object race detection. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 70–82. ACM Press, 2001.

[34] M. Xu, R. Bodik, and M. D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA)*, 2003.

[35] M. Xu, M. D. Hill, and R. Bodik. A regulated transitive reduction (rtr) for longer memory race recording. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 49–60, New York, NY, USA, 2006. ACM Press.

[36] Y. Yu, T. Rodeheffer, and W. Chen. RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 221–234, New York, NY, USA, 2005. ACM Press.

[37] P. Zhou, W. Liu, L. Fei, S. Lu, F. Qin, Y. Zhou, S. Midkiff, and J. Torrellas. AccMon: Automatically Detecting Memory-related Bugs via Program Counter-based Invariants. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Micro-architecture (MICRO)*, 2004.

[38] P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torrellas. iWatcher: Efficient Architectural Support for Software Debugging. In *Proceedings of the 31st International Symposium on Computer Architecture (ISCA)*, 2004.