Accelerating Data Race Detection with Minimal Hardware Support

Rodrigo Gonzalez-Alberquilla[1] Karin Strauss[2,3] Luis Ceze[3] Luis Piñuel[1]

[1] Univ. Complutense de Madrid, Madrid, Spain
{rogonzal, lpinuel}@pdi.ucm.es
[2] Microsoft Research, Redmond WA, USA
kstrauss@microsoft.com
[3] University of Washington, Seattle WA, USA
luisceze@cs.washington.edu

Abstract. We propose a high performance hybrid hardware/software solution to race detection that uses minimal hardware support. This hardware extension consists of a single extra instruction, StateChk, that simply returns the coherence state of a cache block without requiring any complex traps to handlers. To leverage this support, we propose a new algorithm for race detection. This detection algorithm uses StateChk to eliminate many expensive operations. We also propose a new execution schedule manipulation heuristic to achieve high coverage rapidly. This approach is capable of detecting virtually all data races detected by a traditional happened-before data race detection approach, but at significantly lower space and performance overhead.

1 Introduction

Writing much-needed multithreaded programs often requires dealing with concurrency bugs that result from subtle interaction between threads. Among these bugs, data races are the most common. Unsynchronized accesses to shared data could lead to crashes or silent data corruption, so current languages including Java [6] and the new C++ standard [11] disallow or discourage data races.

Researchers have proposed a variety of mechanisms to detect and avoid data races, including many hardware-only [8, 9, 12, 13, 18] and software-only [14, 15, 17] solutions. Hardware-only solutions are typically complex. They require extensive hardware support, like changes to the cache hierarchy, extending cache coherence messages with additional information, and modifying the cache coherence protocol state machine to check for events of interest. The storage requirements, many times close to key processor structures, are also quite prohibitive. Software-only solutions, on the other hand, require no modification to the architecture, but are typically too slow to be an always-on feature. The analysis operations performed in software are slower, and these algorithms require a significant amount of metadata and frequent inter-thread communication.

We propose a hybrid solution: hardware support is boiled down to the bare minimum, reducing complexity, and making detection of inter-thread communication much faster than prior approaches. We augment the ISA with one simple instruction that takes an address as input and returns the coherence state of the cache block containing that address. We also propose a new algorithm that uses this support to effectively detect data races. Our solution leverages two key insights: (1) the dynamic information we need can be extracted from coherence state already tracked by the hardware; (2) there is a well-defined category of dynamic data races that are much cheaper to detect and yet can be proven to include all static data races given sufficient executions. We also show how to perturb execution schedules to speed up the exposure of data races to the detection mechanism, achieving high accuracy compared to traditional happened-before data race detection, but at significantly lower space and time overheads.

Sections 2, 3 and 4 explain the proposed hybrid system, Sections 5, 6 and 7 evaluate it and compare it to previous work, and Section 8 concludes.

2 Background

Terminology. A *data race* exists if there is no synchronization order between any two accesses to the same address by different threads, at least one of them being a write access. A *static race* is a pair of static instructions that, when executed, may be involved in a data race, and a *dynamic race* is one manifestation of a data race at execution time. An *epoch* is the set of dynamic instructions in a thread executed between two consecutive synchronization operations. To simplify our discussion, we assume a static direct mapping of threads to cores. We discuss how to relax this assumption in Sections 4.1 and 4.4.

Data race detection. The basic approaches to data race detection are happened-before-based [13, 14] and lockset-based algorithms [15, 18]. We focus on the former, which leverage Lamport's happened-before relation [2] (to partially order memory accesses based on observed synchronization operations) and program order to determine if conflicting accesses are logically concurrent. Due to space constraints, we omit an explanation of happens-before race detection (HapB) and FastTrack (FastT), a state-of-the-art software implementation of HapB for Java, but we expect the reader to be familiar with them [1, 2, 14].

Cache coherence. Without loss of generality, we assume an invalidate-based MESI protocol. The coherence state of a cache line implicitly carries valuable information about recent accesses, *e.g.*, if the block is in M state in a cache, the line was last written by the local processor; the E state indicates the cache has read that block before; the S state indicates the cache has read or written that block before, and then another cache may have requested that block; the I state indicates that a remote write happened. We leverage this implicit information for lightweight memory access monitoring.

3 Minimal hardware support for data race detection

Our proposed minimal hardware support for data race detection consists of simply exposing the coherence state of a cache block to software via one addi-

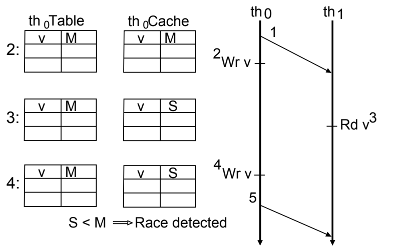


Fig. 1: Race detection with coherence state.

Transition at local thread	Access at local thread	Access at remote thread	Race type
$M \rightarrow S$	write	read	W→R
$M \rightarrow I$	write	write	W→W
$M \rightarrow I$	read	write	R→W
$E \rightarrow I$			
$S \rightarrow I$			

Table 1: Types of downgrades and races.

tional instruction. A software layer then records and uses the state information to detect data races. To leverage this support, we propose a new race detection algorithm, "AccessedBefore", or simply AccB. Its key idea is to use a software-managed address-indexed table to track the last observed state of cache blocks and detect if they have been downgraded within the boundaries of an epoch. A downgraded block within an epoch indicates a potential data race: a remote cache has issued an upgrade request to the block. Note that all state necessary to the analysis is local to a thread, so no inter-thread communication is required; HapB, in contrast, requires substantial inter-thread communication.

3.1 AccessedBefore (AccB) Algorithm

Figure 1 illustrates how AccB works. Thread 0 performs a synchronization operation and starts an epoch (1). When thread 0 performs a write to variable v (2), the corresponding cache block transitions to M state and the software layer records the pair of $address$ and $state < v, M >$ in its local table. When thread 1 subsequently reads variable v (3), the block cached by thread 0 is downgraded to S state. At this point, the software layer is unaware of the downgrade. Finally, when thread 0 is about to write v again (4), the software layer reexamines the current state of v's block (S) and the state recorded in its local table (M), observes a downgrade has happened and detects the race. If this last write never happens, the downgrade check is performed when the epoch ends (5).

Table 1 shows the different types of downgrade and the races they indicate. For example, the first row corresponds to the example in Figure 1. Table 2 summarizes AccB's operation by showing the actions taken by the software layer on each relevant event. Again, note that all analysis actions are local to a thread: the only communication between threads happens through the cache coherence protocol (which would be present even in the absence of AccB). Also, the information collected into the local table only pertains to a *single epoch*, as we are not interested in downgrades across synchronization operations. In addition, AccB epochs can be redefined to the instructions executed between two source synchronization operations because AccB does not require any notion of ordering with previous epochs from other threads, while HapB does. These are three important advantages of our approach when compared to HapB, which has additional storage and communication requirements.

Event of interest	Algorithm action
Beginning of epoch	Clear local table.
Before memory access	Check the current state of the corresponding cache block against the entry in its local table (if any) to detect downgrades.
After memory access	Record the state of the corresponding cache block in its local table.
End of epoch	Check every entry in its local table and their corresponding state in the cache to detect remaining downgrades.

Table 2: Events of interest and related algorithm actions.

In essence, AccB looks for access conflicts between concurrently running epochs, which must be the result of a race (or false sharing as described below). Thus, a race will be detected if the epochs with racy accesses overlap in time.

Interestingly, for every race in a program there must exist an execution in which the epochs of the racy instructions overlap in time (we formally proved this statement but leave it out due to space constraints). We propose an optimization to quickly expose races to AccB: carefully perturbing the execution schedule to increase the probability of overlapping racy epochs.

3.2 Sources of Inaccuracy

We now discuss the two sources of inaccuracy in AccB: (1) false sharing, and (2) block evictions. The next section discusses optimizations to mitigate them.

False sharing. To keep the hardware support required by our proposal to a minimum, we do not extend the memory access information to granularity finer than what is already provided by coherence protocols: a cache block. False sharing of the block may result in false positives. Other race detection approaches at the same granularity would have the same limitation (*e.g.*, HapB). Moreover, our approach can be easily extended to finer granularity if necessary (at extra cost) and is orthogonal to software techniques to mitigate false positives.

Cache block eviction. On eviction, a block loses its state thus the cache loses its ability to detect downgrades, so races may be missed (false negatives).

4 Implementation

4.1 Hardware support

We extend the ISA of an off-the-shelf multiprocessor with a `StateChk` (`StChk off(base), reg`) instruction, which returns the state of `off(base)`'s cache block in register `reg`. If the block is not present in the cache, `StateChk` returns a special NotPresent (NP) state to distinguish from a block in Invalid state. The last valid state is returned if the cache block is in a transient state.

Implementing the `StateChk` instruction requires minor changes to (1) cache data paths, and (2) cache controllers. A new multiplexer creates a path for coherence state into the processor via the existing cache data path. Cache controllers suffer one modification: if the requested block is not currently cached, the cache controller returns the NP state without triggering a miss request.

We assume L1 caches to be the point of coherence, but other configurations are possible. They belong to one of two categories: (1) coherence is maintained among caches private to a hardware thread (*e.g.*, private non-inclusive L1 and L2 caches), and (2) coherence is maintained in caches shared by more than one hardware thread (*e.g.*, SMT processor with a single L1 data cache). In the first case, the proposed mechanism works seamlessly: the state is obtained from the private cache where a hit happens (NP in case of a miss in all private caches). In the second case, accesses and resulting changes of state by different threads need to be distinguished by replicating the state for each thread.

4.2 Software Layer

Data structures. A thread-local hash table records information about accesses performed during an epoch. This table is indexed by data address and stored in main memory. Each entry contains the expected state (based on the type of the last access to the cache block) for the corresponding block and the address of the instruction that performed the last local access to the address.

Instrumentation points. We use dynamic binary rewriting to instrument every source synchronization operation (thread creation, mutex and conditional variable creation, lock release, and waiting) and every memory operation not involved in a synchronization operation. The epoch ending instrumentation is inserted right before source synchronization operations. It searches the local table for any downgraded variables in the ending epoch and subsequently clears the table in preparation for the next epoch.

The memory access instrumentation checks the state of the corresponding address in the cache via a `StateChk` instruction, and compares it with the state recorded in the table. If it detects a downgrade, it reports the race with the corresponding address and the instruction address of the previous access. It then updates the state in the table with the maximum (following the order $M > E = S > I$) of the recorded state and the current state. Using the maximum is safer than executing `StateChk` again after the instrumented access executes because downgrades could be missed in the window between the instrumented access executes and the second `StateChk` instruction executes.

4.3 Optimizations

These optimizations improve accuracy and reduce instrumentation overhead.

Coverage improvement with schedule perturbation. AccB only detects races between epochs that overlap in time. We perturb executions to encourage an increased variety of overlapping epoch sets. When an epoch starts, the thread randomly chooses an action: (1) to continue executing normally, or (2) to join its thread to a rescheduling barrier. The thread waits at this barrier until a bounded random timeout occurs. At this point, all threads that joined this first barrier start executing their epochs. Once a thread finishes its epoch, it joins a checking barrier. When all threads that joined the first barrier join this second barrier, or it times out, epoch checks are done and all threads continue.

Further reducing overheads with extra hardware support (AccB++). We can further accelerate AccB with very simple modifications: we add a small number of metadata bits to caches and use them to reduce the number of accesses to the local table. The coherence state of each cache block is augmented with two extra bits, namely, locally read bit (lrd) and locally written bit (lwr), and caches are augmented with a single downgraded bit (dgd). These bits record the nature of the local accesses within the last epoch (lrd or lwr) to a particular cache block, and downgrades (dgd) to any cache block touched by the local thread within that epoch. An additional instruction gang-clears these bits in the local cache and is used by the software layer in the beginning of every epoch. The cache controller is modified to set lrd or lwr on a local read or local write access, respectively, and to set dgd on downgrades due to remote requests (but only if either lrd or lwr for that block is set). Finally, the `StateChk` instruction returns these three bits together with the regular coherence state.

The software layer uses these additional bits to detect accesses followed by downgrades within an epoch. A `StateChk` instruction is inserted immediately before each memory access and the dgd bit is checked. If the dgd bit is set, a data race is detected. These bits optimize how the local table is used: the lrd and lwr bits reduce the number of accesses to the table, since only information about the first read and write accesses to a variable in each epoch need to be recorded (this is sufficient to report one data race – others may be detected once the first is eliminated). On every memory access, instead of checking if the address is present in the table, the lrd and lwr bits are checked. If none are set, this is the first access to this variable within the current epoch, so the address and the corresponding instruction address are added to the table. If only the lrd bit is set and the access being instrumented is a write, this is the first write access to the variable, so the instruction address of the table entry is updated. If the lwr bit is already set, no new updates are needed. Note this does not completely eliminate the use of the local table because it is still necessary for end-of-epoch checks and for recording the instruction address of accesses.

A small victim cache next to the data cache reduces the impact of cache block evictions. Whenever a block that has its lrd and/or lwr bit set is evicted from the data cache, it is cached in the victim cache. This allows reporting a race even if the block involved in the race has been evicted from the data cache.

4.4 System Issues

Thread migration. Thread migration can lead to changes to the coherence states observed by AccB, affecting its accuracy. To mitigate this potential problem, the software layer may check via an instruction like x86's CPUID in which core the thread is running at every epoch end and compare it with the core identification number recorded in the previous epoch. If they are different, a migration has taken place and the instrumentation ignores any races detected for that epoch. Note that to preserve locality, the OS typically keeps the mapping between threads and cores as stable as possible. Finally, epochs typically run much faster than context switch time scales. Therefore, thread migration is unlikely to lead to major accuracy degradation in AccB.

Speculation. Speculative execution can cause additional false positives in a few scenarios: (1) `StateChk` is executed speculatively in the local core, (2) load is executed speculatively in a remote core, and (3) prefetch request is issued in a remote core. The simplest solution is to allow false positives, which are likely to be low. Other solutions to the first problem are to either reuse mechanisms traditionally used for load speculation (*e.g.*, replay or snoop) or to only set the access bit when the load retires. Solutions to (2) consist of limiting speculation to when it is safe. For example, allowing a speculative load to proceed only when it reaches the point-of-no-return in designs like CHERRY [7] or if the cache block is in the local cache in a valid state. (3) can be easily mitigated by turning prefetching off during debugging runs; an alternative is marking prefetches until later access confirmation, at the cost of extra complexity.

5 Experimental Setup

We evaluate AccB using the PIN [5] dynamic binary instrumentation framework with a tool that includes the software layer from Section 4 and a detailed memory hierarchy: 8 32KB 8-way set associative LRU DL1s with 64-byte blocks and MESI coherence. The latency of `StateChk` is the same as a cache hit.

We compare AccB with an implementation of HapB and FastT using the same instrumentation framework. All algorithms are exposed to the same memory interleavings for accuracy comparisons. HapB is complete, so we verified that every race found by AccB has also been found by its HapB counterpart.

HapB Implementation. We have carefully optimized HapB by using hash-sets for read- and write-sets and Bloom-filters to speed up intersections of hash-sets. Same-thread epochs are stored in an ordered linked list and pruned as soon as an old epoch is ordered before all current epochs (space-optimal implementation). Vector clocks are implemented as regular arrays.

FastT Implementation. We implemented FastT for C++. Unlike the original implementation for Java, which embeds metadata in the object, the implementation for C++ stores metadata in a global table because C++ is not type safe.

Benchmarks. We use the SPLASH-2 benchmarks [16] and commercial workloads (Apache httpd server, MySQL database, AGet, PBZip) compiled with gcc's standard *-O2* optimization flag and run with 8 threads. We do not report

	$brns$	$chlsk$	fft	fmm	lu		$ocean$		$radx$	$rayt$	$vrnd$	$water$		$aget$	$pbzip$
					cnt	$ncnt$	cnt	$ncnt$				$nsqr$	spt		
Speedup (× — HapB/AccB)															
AccB	1.11	1.03	1.02	1.16	0.98	0.95	1.19	1.21	0.98	1.08	5.71	1.23	0.99	–	–
AccB++	1.99	1.31	1.27	1.55	1.54	1.23	1.19	1.21	1.04	1.23	5.90	1.71	1.30	–	–
FastT †	0.03	0.01	0.07	0.01	0.08	0.08	0.33	0.33	0.18	0.21	0.29	0.08	0.06	–	–
Space overhead (%)															
AccB avg	0.8	9.4	31.8	3.3	19.9	20.2	25.6	26.1	32.9	41.0	0.2	15.8	30.6	0.1	5.6
AccB max	77.1	13.0	87.5	29.6	33.3	33.2	110.6	113.3	116.5	40.4	0.4	25.7	80.1	0.1	32.1
Accuracy (%)															
AccB	97.8	–	–	95.4	–	–	100.0	100.0	–	100.0	100.0	–	–	100.0	100.0

Table 3: Performance, space overhead and accuracy comparison of AccB and HapB.

performance for AGet and PBZip because they are non-deterministic. We verify that AccB detects races reported in the literature for Apache and MySQL [3].

6 Evaluation

6.1 AccB versus HapB

Table 3 compares AccB and HapB in terms of performance, space overhead and accuracy. The first group of rows in Table 3 show the speedup of an application instrumented with AccB, and with extra hardware support (AccB++), compared to HapB. For example, barnes instrumented with AccB runs 11% faster than when instrumented with HapB. The speedup grows to almost 2× with AccB++. Overall, AccB++ achieves speedups of up to almost 6×. A few benchmarks (lu, radix, and water spatial) experience modest slowdowns with AccB, caused by the type of synchronization used in these benchmarks: most synchronization is based on barriers, which allow HapB to clean up all information about old epochs and significantly reduce its checking overheads. AccB incurs extra overheads because it performs table checks on every memory access in addition to end-of-epoch checks. Note that AccB++ always shows speedups [4].

The second group shows average and maximum space overheads for AccB over HapB. For example, AccB uses on average 0.8% and at most 77% of the storage used by HapB for barnes. For most benchmarks, AccB uses significantly less space than HapB. In some cases (ocean, radix), AccB incurs a higher maximum space overhead compared to HapB (but the average is still lower). This is due to uncommon program behavior: frequent barriers and large accessed sets.

The last row shows accuracy, *i.e.*, how many races AccB detects compared to HapB for 500 runs. AccB detects all races for most benchmarks. Section 6.3 provides more insight into those very few races not detected by AccB.

[4] † The results show that FastT is much slower than HapB. The reason is twofold: first, FastT experiences additional overheads compared to its original Java implementation due to the global table required by C++; second, HapB performs intersections at the end of each epoch, FastT performs checks at every access.

	AccB	AccB++
Lookups	3326.7%	15.6%
Updates	100.0%	29.8%
Branches	4.2%	5.1%

Table 4: Number of operations executed by AccB and AccB++ compared to HapB.

	HapB	AccB
Avg. entries per epoch	360.3	623.2
Avg. epochs in history	16.5	0
Avg. simultaneous entries	71.6k	9.1k
Size (MB)	2.15	0.28

Table 5: Overheads, storage requirements of HapB and AccB.

6.2 Overheads Characterization

Performance. Table 4 characterizes the performance overheads of AccB and AccB++ compared to HapB, aggregated for all benchmarks. This study is data structure independent: it counts high level operations to each algorithm's data structures, *i.e., lookups* and *updates*. The numbers show the relative frequency of events for AccB and AccB++, normalized to HapB. Lookups (row 2) and updates (row 3) are direct accesses to AccB's local table and to HapB's sets. Branches (row 4) refer to branches taken while manipulating these data structures. AccB incurs many more lookups than HapB because AccB performs lookups at every memory access, while HapB performs them only at epoch ends. Even though AccB's lookups are more frequent, AccB is still faster than HapB because there is high locality in AccB's table accesses and most are cache hits (besides being thread-local). Also, HapB is very control flow intensive, as demonstrated by the large number of branches. HapB's data structures are larger (due to multiple epochs, not just the current), which results in worse cache behavior. Finally, HapB requires transferring vector clocks and epoch information, which implies additional communication among threads, *i.e.*, costly misses.

With simple additional support, AccB++ has lower overheads than AccB. AccB++ reduces lookups by two orders of magnitude and updates by more than 60%. AccB++ has higher number of branches, but still much lower than HapB.

Space. Table 5 shows the space overhead of HapB and AccB averaged across all benchmarks. It reports the number of entries per epoch (row 2), overall number of epochs kept in history (row 3), total number of entries used by all epochs in all threads simultaneously (row 4) and overall storage requirements (row 5).

AccB records more entries per epoch than HapB (row 2). This is due to AccB only ending epochs at synchronization sources, which makes AccB epochs longer. AccB keeps no history while HapB keeps history on 16.5 epochs on average (row 3). AccB requires a much lower total number of entries (over 7× fewer). Overall, AccB reduces space overhead by more than 7×. Storage requirements for AccB++ are similar to AccB. In addition to being larger, the storage HapB requires is *shared* and accessed by all threads when their epochs end. Conversely, the storage AccB requires is much smaller and *purely local*.

$brns$	$chlsk$	fft	fmm	lu		$ocean$		$radx$	$rayt$	$vrnd$	$water$		$aget$	$pbzip$
				cnt	$ncnt$	cnt	$ncnt$				$nsqr$	spt		
99.1	98.3	100.0	81.3	100.0	100.0	100.0	98.6	100.0	100.0	100.0	100.0	100.0	51.8	75.0

Table 6: Relative percentage of false positives in AccB compared to HapB.

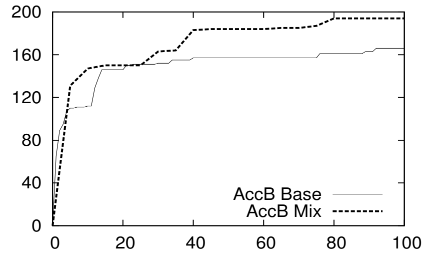
6.3 Accuracy Characterization

False positives. Table 6 shows the false positives detected by AccB relative to HapB at the same tracking granularity. AccB never has more false positives than HapB. False positives are inherent to the tracking granularity (cache blocks) for both AccB and HapB and can be reduced with additional software support (*e.g.*, by changing the data layout to avoid false sharing), but this is beyond the scope of this paper.

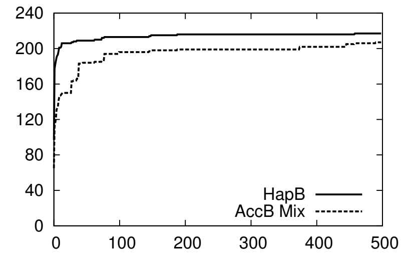
False negatives. As explained in Section 3.2, AccB has two sources of false negatives (*i.e.*, missed races). The first is due to limited cache capacity, which causes cache blocks to be evicted and the access information to be lost (*CBE* – cache block evictions). The second is due to epochs with races not overlapping on AccB executions. We separate the two effects by modifying our simulator with unbounded space to store evicted cache blocks, such that the CBE problem is completely eliminated. No new races were found, so all races missed by AccB for these benchmarks are due to non-overlapping epochs, a problem that can be addressed with scheduling perturbations and/or multiple runs. Other benchmarks with larger epochs could cause the CBE problem. However, architectures with private L2 caches are common today, so there is much more space than the DL1s used in this evaluation. Alternatively, a victim cache that only stores evicted downgraded lines may be sufficient to mitigate the problem.

Sensitivity to scheduling perturbations and number of runs. Figure 2(a) shows how the aggregate number of static races detected by AccB, with and without scheduling perturbations (AccB Mix and AccB Base), grows with the number of executions for fmm. After about 25 runs, AccB Mix clearly shows new races while AccB Base does not. This happens when the scheduling perturbations start exposing more diverse epoch overlaps. These results also show that scheduling perturbations indeed help AccB find races faster.

Figure 2(b) shows how fast AccB Mix approximates the number of static races detected by HapB over 500 runs. AccB detects most races in the first few executions (about 2/3 are detected within the first 10 runs). The number of races AccB Mix detects continues growing after that, although increasingly more slowly. We manually inspected a few of the races that AccB had not detected after 500 runs and found that for each undetected race there was another race that originated at the same programming mistake (*e.g.*, missing critical section) and that was successfully detected by AccB.



(a) Aggregate number of static races found as the number of executions increases for AccB and AccB with scheduling perturbations (AccB Mix).



(b) Aggregate number of static races found as the number of executions increases for AccB Mix, compared to HapB.

Fig. 2: Sensitivity to scheduling perturbations and number of runs.

7 Related Work

Conflict exceptions [4] (CE) relates to our work in the type of bugs it detects. CE detects when a synchronization-free region (epoch) conflicts with another concurrent synchronization-free region. Such conflicts can only happen when a data race exists. This is in essence the same type of event AccB detects. However, CE detects these events in a fully precise way, in order to throw an exception. This requires significantly more hardware (50% cache overhead for access bits). We sacrifice some precision in order to keep hardware at a minimum. AVIO [3] is an atomicity violation detector that also augments and leverages coherence state. However, atomicity violations do not necessarily imply data races.

The works most related to ours are by Min and Choi [8], and Nagarajan and Gupta [10]. Both propose using traps to expose certain cache coherence events to enable analysis of parallel program behavior. Nagarajan and Gupta [10] showcased their mechanism with deterministic replay and barrier speculation. Min and Choi [8] developed a limited form of happened-before detection for a subclass of programs (structured parallelism only). In contrast, our hardware proposal does not rely on software traps; it is essentially a *load operation that returns coherence state*. Software traps are arguably more flexible, but are also much more costly to implement. Importantly, these proposals focus on other applications of tracking coherence events. We propose a *new race detection algorithm* that uses our novel hardware support to reduce performance overheads, and also significantly reduce space overhead compared to happened-before.

8 Conclusions

In this paper, we propose a data race detection solution that requires minimal hardware support. This solution captures many of the same races a more traditional mechanism based on happened-before captures, but at much lower overheads. We expect the overhead reductions and the hardware simplicity to make this solution sufficiently compelling for multicore designers to include support in their designs.

Acknowledgements: This work was supported in part by the National Science Foundation under grant CCF-1016495, the Spanish government under CICYT-TIN 2008/00508 and an FPU grant, a Microsoft Faculty Fellowship, and gifts from Intel.

References

1. FLANAGAN, C., AND FREUND, S. N. FastTrack: Efficient and Precise Dynamic Race Detection. In *Conference on Programming Language Design and Implementation* (2009).
2. LAMPORT, L. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM* (1978).
3. LU, S., ET AL. AVIO: Detecting Atomicity Violations via Access-Interleaving Invariants. In *International Conference on Architectural Support for Programming Languages and Operating Systems* (2006).
4. LUCIA, B., ET AL. Conflict Exceptions: Providing Simple Concurrent Language Semantics with Precise Hardware Exceptions. In *International Symposium on Computer Architecture* (2010).
5. LUK, C.-K., ET AL. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Conference on Programming Language Design and Implementation* (2005).
6. MANSON, J., PUGH, W., AND ADVE, S. The Java Memory Model. In *Symposium on Principles of Programming Languages* (2005).
7. MARTINEZ, J., ET AL. Cherry: Checkpointed Early Resource Recycling in Out-of-order Microprocessors. In *International Symposium on Microarchitecture* (2002).
8. MIN, S. L., AND CHOI, J.-D. An Efficient Cache-based Access Anomaly Detection Scheme. In *International Conference on Architectural Support for Programming Languages and Operating Systems* (1991).
9. MUZAHID, A., ET AL. SigRace: Signature-Based Data Race Detection. In *International Symposium on Computer Architecture* (2009).
10. NAGARAJAN, V., AND GUPTA, R. ECMon: Exposing Cache Events for Monitoring. In *International Symposium on Computer Architecture* (2009).
11. NELSON, C., AND BOEHM, H.-J. Concurrency Memory Model. C++ standards committee paper., October 2007.
12. PRVULOVIC, M. CORD: Cost-effective (and Nearly Overhead-free) Order-recording and Data Race Detection. In *International Symposium on High Performance Computer Architecture* (2006).
13. PRVULOVIC, M., AND TORRELLAS, J. ReEnact: Using Thread-Level Speculation Mechanisms to Debug Data Races in Multithreaded Codes. In *International Symposium on Computer Architecture* (2003).
14. RONSSE, M., AND DE BOSSCHERE, K. RecPlay: a fully integrated practical record/replay system. *ACM Transactions on Computer Systems 17*, 2 (1999).
15. SAVAGE, S., ET AL. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems 15*, 4 (1997).
16. WOO, S., ET AL. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *International Symposium on Computer Architecture* (1995).
17. YU, Y., RODEHEFFER, T., AND CHEN, W. RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking. In *Symposium on Operating Systems Principles* (2005).
18. ZHOU, P., ET AL. HARD: Hardware-Assisted Lockset-based Race Detection. In *International Symposium on High Performance Computer Architecture* (2007).