Cache Abstraction for Data Race Detection in Heterogeneous Systems with Non-coherent Accelerators

MAY YOUNG, ALAN J. HU, and GUY G. F. LEMIEUX, University of British Columbia, Canada

Embedded systems are becoming increasingly complex and heterogeneous, featuring multiple processor cores (which might themselves be heterogeneous) as well as specialized hardware accelerators, all accessing shared memory. Many accelerators are non-coherent (i.e., do not support hardware cache coherence) because it reduces hardware complexity, cost, and power consumption, while potentially offering superior performance. However, the disadvantage of non-coherence is that the software must explicitly synchronize between accelerators and processors, and this synchronization is notoriously error-prone.

We propose an analysis technique to find data races in software for heterogeneous systems that include non-coherent accelerators. Our approach builds on classical results for data race detection, but the challenge turns out to be analyzing cache behavior rather than the behavior of the non-coherent accelerators. Accordingly, our central contribution is a novel, sound (data-race-preserving) abstraction of cache behavior. We prove our abstraction sound, and then to demonstrate the precision of our abstraction, we implement it in a simple dynamic race detector for a system with a processor and a massively parallel accelerator provided by a commercial FPGA-based accelerator vendor. On eleven software examples provided by the vendor, the tool had zero false positives and was able to detect previously unknown data races in two of the 11 examples.

CCS Concepts: • **Software and its engineering → Software verification and validation**; • **Computer systems organization → Heterogeneous (hybrid) systems**; **Embedded software;**

Additional Key Words and Phrases: Data race, hardware accelerator, memory coherence, caching

ACM Reference format:
May Young, Alan J. Hu, and Guy G. F. Lemieux. 2022. Cache Abstraction for Data Race Detection in Heterogeneous Systems with Non-coherent Accelerators. *ACM Trans. Embedd. Comput. Syst.* 22, 1, Article 6 (December 2022), 25 pages.
https://doi.org/10.1145/3535457

1 INTRODUCTION

Embedded systems are becoming increasingly complex and heterogeneous, with multiple processor cores and diverse accelerators [10, 25]. Common accelerators include GPUs, TPUs, specialized processors for software-defined networking, and FPGAs to allow acceleration of arbitrary user-specified computations. With the rise of open ISAs like RISC-V and agile hardware development,

This work was supported by Discovery Grants from the Natural Sciences and Engineering Research Council of Canada (NSERC).
Authors' addresses: M. Young and A. J. Hu, Department of Computer Science, University of British Columbia, Canada; emails: {youngmay, ajh}@cs.ubc.ca; G. G. F. Lemieux, Department of Electrical and Computer Engineering, University of British Columbia, Canada; email: lemieux@ece.ubc.ca.

1539-9087/2022/12-ART6
https://doi.org/10.1145/3535457

even processor cores will increasingly be customized and heterogeneous, with custom instructions for specific computations.

Accelerators and processors commonly communicate via shared memory, creating the problem of memory coherence: how to prevent processors and accelerators from accessing stale data. One solution is to require all processors and all accelerators to support a common, hardware cache-coherence protocol. Non-coherent accelerators, however, offer several advantages: simpler hardware, lower cost, and lower power consumption. Non-coherent accelerators can also achieve higher performance (e.g., up to 3x [9]) by performing coherence actions only when necessary and by using the higher throughput of large DMA bursts. Furthermore, even if we wish to support hardware cache coherence in an accelerator, the computational patterns *within* the accelerator can be very different from those in CPUs, necessitating different cache coherence protocols [22], and creating the possibility of mutually non-coherent coherence domains.

Thus, the burden of coherence shifts to software, which must insert cache flushing and synchronization instructions into application programs (or into library code that tries to hide this complexity). Too much flushing or synchronization results in poor performance. Too little or incorrect synchronization results in bugs, often notoriously hard-to-find, irreproducible, non-deterministic, concurrency bugs.

Data races are a major source of concurrency bugs. A data race occurs when there are two (or more) operations affecting a memory location, of which at least one is a write, whose order of occurrence isn't fixed by the program [3]. The importance of data race detection has spawned an extensive and highly impactful body of research on their detection (briefly surveyed in Section 5), and manifested in widely deployed tools like Thread Sanitizer [20] and TSVD [12].

All prior work, however, has neglected the problem of data races arising from the interaction of caching and non-coherent memory accesses. This omission is understandable — the whole point of cache coherence is to maintain the abstraction of an atomic shared memory, which allows software (and data race analysis of software) to ignore caching altogether. Unfortunately, heterogeneous systems mixing coherent and non-coherent memory accesses break this abstraction, and this problem has become important with the proliferation of non-coherent accelerators. For example, GPUs are generally non-coherent with host CPUs and require that caching be disabled or explicit coherence operations be performed if an application requires a memory operation to be visible across all processors and accelerators [22].

Although the importance of the problem is due to non-coherent accelerators, the root cause of the problem is actually the caches. Uncached memory accesses behave exactly as existing data race theory expects: as explicit reads, writes, and synchronizations through a shared atomic memory. Even (non-cached) accesses to a local memory shared among parts of an accelerator can be handled this way. The problem with caching is that caches can generate reads or writes to shared memory at unpredictable times, due to cache line allocations, evictions, pre-fetching, writebacks, etc. To employ existing data race analyses, one could conceivably emulate the caches using software threads that model all possible behaviors of the specific caches in a heterogeneous system, and then analyze the resulting software combination. However, such an ad hoc approach is labor-intensive and error-prone, with no guarantee of soundness.

In this paper, we introduce the first systematic approach to find data races arising from the interaction of cached memory accesses and non-coherent memory accesses (or accesses from a different coherence domain), as arise in heterogeneous systems with non-coherent accelerators. Because the analysis of cache behavior is the root problem, our central contribution is a novel abstraction of cache behavior, which we prove to be sound (i.e., any data race in an execution is guaranteed to be detected). To demonstrate the precision of our abstraction, we implement it in a simple, proof-of-concept dynamic race detector for a commercial, FPGA-based accelerator, and

find zero false positives while discovering two previously unknown races in code published by the vendor.

2 EXAMPLE

For a concrete example, consider the code in Figure 1, for a simple, heterogeneous system consisting of a single scalar CPU and a single vector accelerator (Figure 2). (Details are in the figure captions.)

This is actual code from a test/demonstration program formerly supplied with the SDK for the VectorBlox MXP FPGA-based matrix accelerator.[1] The VectorBlox API is conveniently simple, but it captures all of the issues that arise in more complicated APIs like CUDA or OpenCL: cached and uncached accesses to memory from CPU or accelerator, as well as synchronization and cache management instructions. We will use this code and this API as a running example throughout this paper.

It is important to note that VectorBlox API functions are non-blocking and therefore execute asynchronously to the CPU program order. Internally, the MXP hardware places vector-compute requests and vector-DMA requests in separate queues and executes each queue in FIFO order. Between queues, it detects read-after-write hazards and uses interlocks to maintain program order.

Even this simple code uses two types of synchronization to avoid data races. The `vbx_sync()` on line 27 stalls the CPU until the accelerator has completed all outstanding requests. It is necessary to prevent a data race between when the DMA engine writes `vector_out` on line 26 and when the CPU reads `vector_out` on line 29, because otherwise, the CPU might (or might not) reach line 29 before the accelerator completes the DMA requested in line 26. The other type of synchronization happens because the `vbx_shared_malloc` calls on lines 5–7 specifies that the `vector_` variables require *uncached* accesses. If they had been allowed to be accessed via cached reads and writes (which would improve performance in lines 14, 17, and 19), the cached values for `vector_in1` and `vector_in2` might not be written back to memory in time for the DMAs on lines 23–24. An alternative to specifying the uncached memory accesses would be to insert flush instructions before line 23. Getting the synchronization correct to eliminate data races is notoriously hard — in fact, our analysis discovered a previously unknown data race even in this simple example (described in Section 4.4).

3 THEORETICAL FRAMEWORK

This section presents the main theoretical results of this paper. Sections 3.1 and 3.2 present preliminaries: the classic happens-before relation that underpins data race detection, and why the classic analysis fails when there are non-coherent accesses to memory. Section 3.3 introduces our basic abstraction, which makes data races resulting from cache behavior visible, yet hides details about the cache implementation. Directly applying this abstraction, though, would result in an exponential blow-up in the number of happens-before graphs that would need to be analyzed. Section 3.4 provides a set of theorems that eliminate this combinatorial blow-up, while still guaranteeing that the analysis is sound (i.e., it does not miss data races). Finally, Section 3.5 provides techniques to prune the happens-before graph, making the analysis tractable, while still preserving the existence of data races. Put together, these results create an abstract model of caches that exposes all possible data races in a heterogeneous system with non-coherent components, that hides as much of

[1]The code presented here has been modified slightly for brevity and clarity. VectorBlox was acquired by Microchip Technology in late 2019. Although the original SDK is no longer online, a copy of the SDK can be found at http://www.github.com/ubc-guy/mxp.

```
1  vbx_mm_t *scalar_in1 = malloc( N*sizeof(vbx_mm_t) );
2  vbx_mm_t *scalar_in2 = malloc( N*sizeof(vbx_mm_t) );
3  vbx_mm_t *scalar_out = malloc( N*sizeof(vbx_mm_t) );
4
5  vbx_mm_t *vector_in1 = vbx_shared_malloc( N*sizeof(vbx_mm_t) );
6  vbx_mm_t *vector_in2 = vbx_shared_malloc( N*sizeof(vbx_mm_t) );
7  vbx_mm_t *vector_out = vbx_shared_malloc( N*sizeof(vbx_mm_t) );
8
9  vbx_sp_t *v_in1 = vbx_sp_malloc( N*sizeof(vbx_sp_t) );
10 vbx_sp_t *v_in2 = vbx_sp_malloc( N*sizeof(vbx_sp_t) );
11 vbx_sp_t *v_out = vbx_sp_malloc( N*sizeof(vbx_sp_t) );
12
13 test_zero_array( scalar_out, N );
14 test_zero_array( vector_out, N );
15
16 test_init_array( scalar_in1, N, 1 );
17 test_copy_array( vector_in1, scalar_in1, N );
18 test_init_array( scalar_in2, N, 1 );
19 test_copy_array( vector_in2, scalar_in2, N );
20
21 scalar_time = test_scalar( scalar_out, scalar_in1, scalar_in2, N );
22
23 vbx_dma_to_vector( v_in1, (void *)vector_in1, N*sizeof(vbx_sp_t) );
24 vbx_dma_to_vector( v_in2, (void *)vector_in1, N*sizeof(vbx_sp_t) );
25 test_vector( v_out, v_in1, v_in2, N, scalar_time );
26 vbx_dma_to_host( (void *)vector_out, v_out, N*sizeof(vbx_sp_t) );
27 vbx_sync();
28
29 errors += test_verify_array( scalar_out, vector_out, N );
```

Fig. 1. Example code for CPU with vector accelerator. This code is part of a test program in the SDK of the VectorBlox MXP.[2] The CPU (with cache) and the vector accelerator (with non-coherent scratchpad memory) communicate via shared memory. The code performs the same vector addition twice, once on the CPU and once on the vector accelerator, to demonstrate the programming model and speedup. Lines 1–3 allocate three vectors in main memory. The names start with `scalar_` because they are intended for the scalar CPU to perform the vector addition. Lines 5–7 allocate three more vectors, also in main memory. These are intended for communication with the vector accelerator, and the `vbx_shared_malloc` directive tells the compiler to require the CPU to use *uncached* reads and writes when accessing these locations. Lines 9–11 allocate three vectors in the accelerator's private scratchpad memory. Lines 13–19 initialize the input and output vectors in the main memory. Line 21 calls a function for the scalar CPU to iterate through its vectors, performing the vector addition. Lines 23–26 perform the same vector addition using the accelerator. This entails the CPU requesting the accelerator to use DMA to copy the input vectors into its scratchpad memory (lines 23–24), perform the vector addition in the scratchpad (line 25), and use DMA to copy the result back into shared memory (line 26). Requests from the CPU to the accelerator are non-blocking, so the CPU continues to execute instructions while the accelerator performs the requested actions. The sync on line 27 stalls the CPU until the accelerator finishes. Line 29 compares the results from the scalar CPU and vector accelerator to check for errors. The code appears to be properly synchronized to avoid data races between the CPU and accelerator.

[2] https://github.com/ubc-guy/mxp/blob/master/examples/software/bmark/vbw_vec_add_t/test.c.

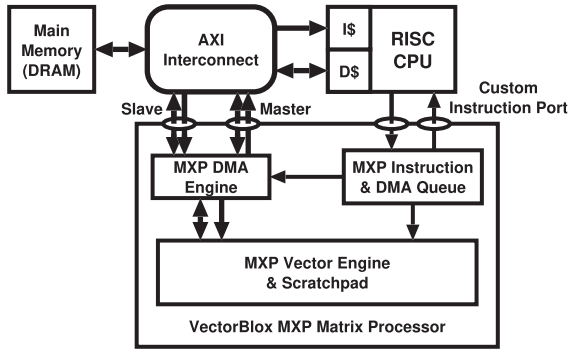


Fig. 2. Example heterogenous system with accelerator. The code in Figure 1 was written for an embedded system with a VectorBlox MXP [21] accelerator (with non-coherent scratchpad) and scalar CPU with caches (configurable with VectorBlox ORCA (RISC-V), Altera Nios II, or Xilinx MicroBlaze soft cores, or ARM Cortex-A9 or A53 hard cores), connecting to main memory through an AXI interconnect. The CPU sends the accelerator requests through a custom instruction port, but data transfers occur through main memory.

the implementation details of the caches as possible, and that can be efficiently used for data race detection.

3.1 Happens-Before Graph

A data race is defined as two (or more) operations on a memory location, of which at least one is a write, whose order of occurence isn't determined. Therefore, any data race analysis must reason about when two operations must be or might not be ordered, and hence we start with Lamport's *happens-before* relation [11]. The happens-before relation, which we'll denote by $x \rightarrow y$, is a strict, partial order on the events in a system's execution. It captures all ordering that *must* occur between the events in the execution. So, if $x \rightarrow y$, it means event x must occur before event y, regardless of any possible reordering of concurrent actions, non-determinism, etc. We can define the happens-before relation as the transitive closure of the *program order* for each thread of execution (i.e., the order of the instructions as they are executed by a CPU or accelerator[3]) and any *causal ordering*, where one event causes or enables the other event.

For example, in the code listing in Section 2, the program order on the CPU says that line 1 happens before line 2, which happens before line 3, etc. An example of causal ordering is that on line 23, the CPU executes an instruction (which is ordered in program order on the CPU), which requests the accelerator to perform a DMA operation at some later time. So, the CPU request happens before (causally) the DMA operation. However, the request is non-blocking, so the DMA operation itself is unordered with respect to the next CPU operation on line 24. The VectorBlox MXP accelerator performs operations in-order, so there would also be a program order relationship between when the accelerator performs the two DMA operations. Figure 3 shows the Hasse diagram for these four operations. For convenience, we will refer to "graphs" and "edges" interchangeably with partial order terminology.

[3]This definition assumes sequential consistency. The soft CPU cores and VectorBlox accelerator are in-order, so they meet this assumption. With a relaxed memory model, defining happens-before is more subtle, but essentially, entails removing program order edges as allowed by the memory model (e.g., [4] for the ARM). Alternatively, we can use the order in which operations appear at the memory interface of the CPU.

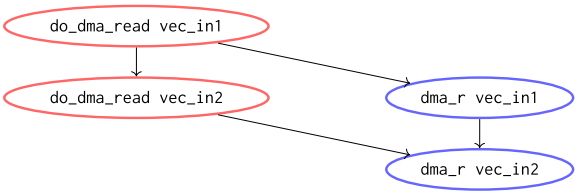


Fig. 3. Example graph of happens-before relation. The `do_dma_read` nodes are CPU operations requesting that the accelerator perform a DMA read, and the `dma_r` nodes are when the accelerator actually does the read. The two `do_dma_read` nodes are ordered by CPU program order. The two `dma_r` nodes are ordered because the VectorBlox DMA engine is in-order. The diagonal edges are causal edges.

3.2 Breaking the Abstraction of Invisible Caching

Data race detection reduces to building the happens-before graph (or an efficient abstraction thereof) and checking for two unordered nodes that access the same memory location, of which at least one is a write. In the classical literature on data race detection, the nodes correspond to memory reads and writes, which are assumed to happen atomically. The correctness of this assumption relies on the elegant abstraction that caching (and coherent memory systems in general) provide: the caches are invisible.[4] Unfortunately, modern heterogeneous systems, with different types of potentially non-coherent memory accesses, break this long-held assumption.

For example, Figure 4 shows an archetypal pattern by which a CPU requests computation from an accelerator. Without caching (or with coherent, and therefore invisible caches), there are no data races. When sending data from CPU to accelerator, the CPU writes data to shared memory, which happens before it requests that the accelerator act, which happens before the accelerator can read this data. Similarly, when sending results from the accelerator back to the CPU, the accelerator writing the results to shared memory happens before the CPU can pass the sync, which happens before the CPU reads the results from shared memory.

However, non-coherent accelerators break the abstraction that caching is invisible. The problem is that caches can allocate (or even pre-fetch) cache lines from memory, and later write data back to memory, at unpredictable times, and these memory operations are visible to the non-coherent accelerators. This creates new, subtle, and hard-to-detect data races.

3.3 Shared Memory with Visible Abstract Caches

Accordingly, if the classical assumption that caching is invisible is broken, then we need to model memory operations more precisely: memory operations to/from the caches must be visible.

In systems that communicate with non-coherent accelerators via shared memory, shared memory accesses fall into just a few idiomatic categories: CPUs make cached and uncached reads and writes, and non-coherent accelerators access main memory via DMA or other uncached transfers. Also, CPUs make requests to accelerators, and there are sync or barrier instructions to stall a thread until completion of requested actions. Cache flushing instructions can be used to force writebacks. For all of these operations except cached reads/writes, the rules for modeling their behavior in the happens-before graph are straightforward. Uncached reads and writes create nodes in the happens-before graph exactly as in classical data race analysis. DMA reads and writes do too, except that there is an additional causal edge from the node requesting the DMA to the node performing the DMA (e.g., Figure 3). Program order edges connect consecutive operations performed by a single thread or by an in-order accelerator. Synchronization/barrier instructions relate

[4]They are invisible from a correctness or functionality perspective. Obviously, they can be *very* visible from a performance perspective!

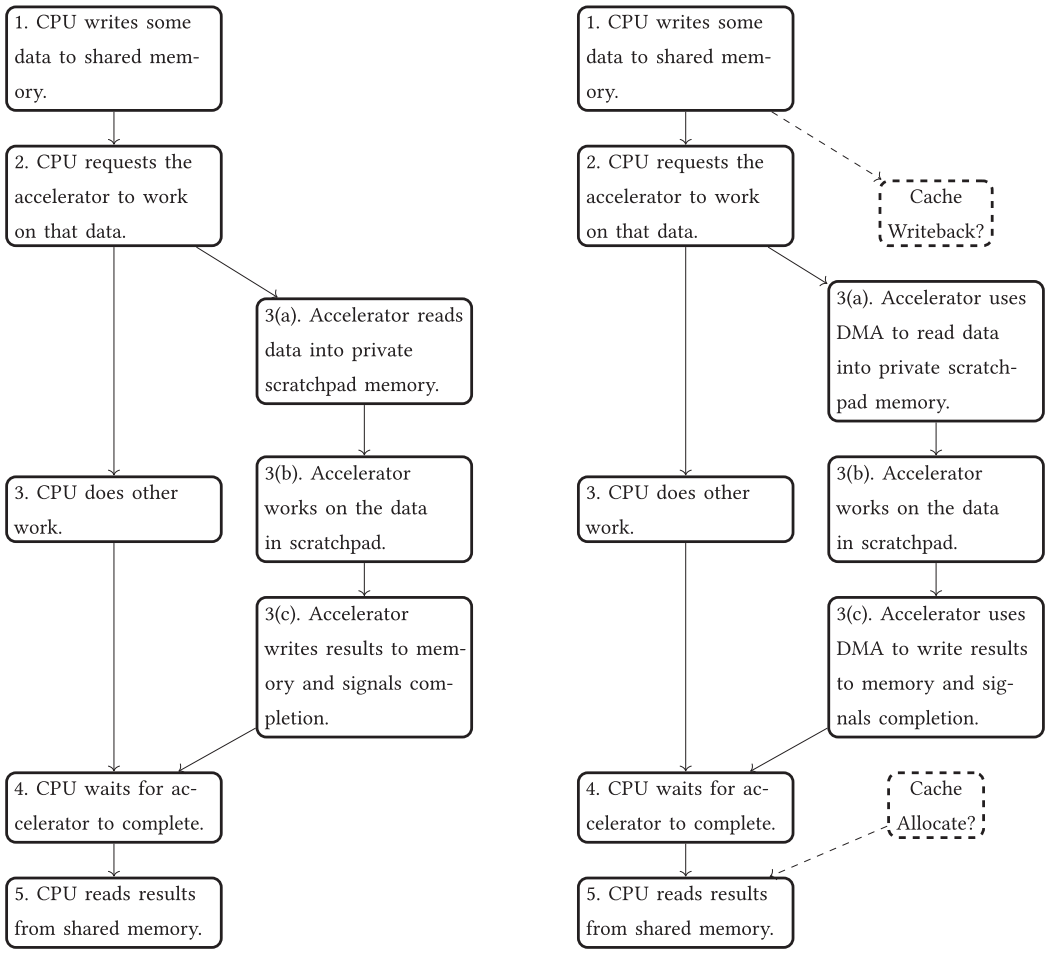


Fig. 4. Archetypal pattern of a CPU requesting computation from an accelerator. The lefthand side shows the happens-before graph under the assumption that caching is invisible (e.g., if the entire system is coherent), as in classical data race analysis. There are no data races, because the CPU's write to memory (node 1) happens before the CPU sends requests to the accelerator (node 2), which happens before the accelerator reads the data (node 3(a)). Similarly, the accelerator's write to memory (node 3(c)) happens before the CPU's wait (node 4), which happens before the CPU's read from memory (node 5). The righthand side shows the same graph, except that a non-coherent accelerator breaks the abstraction that caching is invisible. Now, there are multiple data races. For example, the write in node 1 doesn't really reach memory until the cache writeback, which is in a data race with the accelerator DMA read in node 3(a) (and even with the accelerator DMA write in node 3(c)). The writeback may or may not happen before the accelerator reads or writes the same memory. Similarly, the CPU read in node 5 depends on data being in the cache. This data might be the same data (if it's still valid in the cache) that was written in node 1, or cache line might have been evicted, in which case there is a cache allocation that reads from memory, and that node is in a data race with the accelerator DMA write in node 3(c). The main point is that modern systems with non-coherent accelerators break the classical abstraction that caching is invisible, creating new data races.

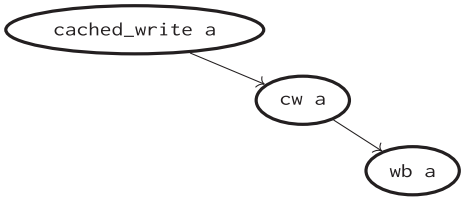
different program orders: they take their place in the program order of the thread that executes them, but they have causal edge from all operations they depend on. Table 1 lists the types of nodes in our happens-before graphs for the simple VectorBlox API. (The cache-related nodes are explained more below.)

Table 1. Operations Tracked for the VectorBlox Happens-Before Graph

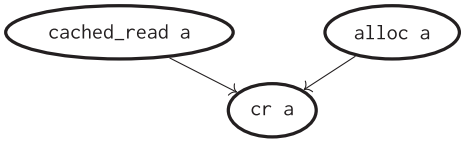
Type of Node	Meaning	Edges
cached_read	CPU tries to read data from cache.	Out to a new cr node.
cr	Cache returns data to CPU.	In from cached_read. In from a new alloc node. Out to next CPU instruction (or relaxed if memory model allows, e.g., if reads from different addresses are allowed to complete out-of-order).
alloc	Cache allocates cache line and reads data from main memory.	In from most recent alloc or wb to the same address. Out to the cr node that created it.
cached_write	CPU tries to write data to cache.	Out to a new cw node.
cw	Cache accepts data from CPU.	In from cached_write. Out to a new wb node. Out to next CPU instruction (or relaxed if memory model allows, e.g., if writes to different addresses are allowed to complete out-of-order).
wb	Cache writes back data to main memory.	In from the cw node that created it. In from most recent alloc or wb to the same address.
cache_flush	CPU tells cache to remove data, performing wb if dirty.	In from the most recent wb nodes for the flushed cache lines.
uncached_read	CPU reads data from main memory, bypassing cache.	(Nothing extra. Program order edges only.)
uncached_write	CPU writes data to main memory, bypassing cache.	(Nothing extra. Program order edges only.)
do_dma_read	CPU asks accelerator to read from main memory.	Out to a new dma_r node.
dma_r	Accelerator reads data from main memory.	In from the do_dma_read. In from most recent dma_r or dma_w.
do_dma_write	CPU asks accelerator to write to main memory.	Out to a new dma_w node.
dma_w	Accelerator writes data to main memory.	In from the do_dma_write. In from most recent dma_r or dma_w.
sync	CPU waits for outstanding DMA operations to complete.	In from the most recent dma_r or dma_w.

All operations (except sync) are parameterized with the addresses affected. The column "Edges" indicates the edges added to the happens-before graph when the specified node is added to the graph, using our basic abstraction in Section 3.3. These edges are in addition to the usual program-order edges in a classical happens-before graph (which might be relaxed if using a relaxed CPU memory model).

The challenge for data race analysis with mixed coherent and non-coherent memory accesses is how to model the caches and the unpredictable memory traffic they can generate. One could imagine modeling the behavior of a specific cache precisely, but such an approach is labor-intensive, error-prone, and brittle. Instead, a broadly applicable data race analysis should avoid modeling excessive details of specific caches, e.g., associativity, eviction and replacement policies, pre-fetching, etc. These details might change in different hardware configurations, are not reasonable for



(a) The CPU does a cached write to address a, which causes the cache to accept the data, which eventually generates a writeback to address a.



(b) The CPU does a cached read, which causes the cache to supply the data, but that requires the cache line to have been allocated already.

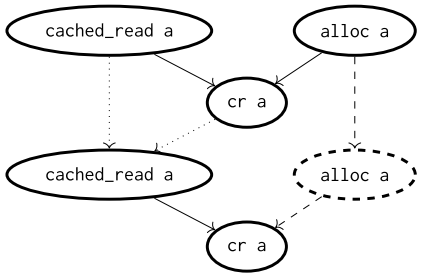
Fig. 5. Basic modeling of cached writes and reads in happens-before graph.

programmers to depend on, and are hard to model accurately. On the other hand, an excessively conservative abstraction will result in too many false data race detections. We do make the assumption of writeback caches, which are typical in multiprocessing systems, although our theory could be modified to handle writethrough caches. Also, we do require knowledge of the cache line size and writeback granularity, so that our analysis can correctly compute the memory addresses touched by cache allocations and writebacks.

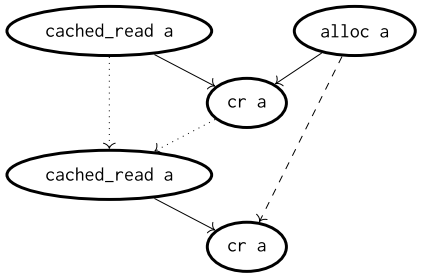
Accordingly, as the first layer of our modeling of caches, we create a very simple, very general model of cache behavior that suppresses most details (Figure 5). Caches are modeled similarly to accelerators: just as the `do_dma_write` CPU operation generates a causal edge to the corresponding `dma_w` operation on the accelerator, the `cached_write` CPU operation generates a causal edge to the corresponding `cw` node in the cache, which denotes the cache accepting the written data. The cache, in turn, would have a causal edge to a `wb` writeback node, because the cache line is dirty and will eventually be written back to memory. It is this `wb` operation that accesses main memory and must be checked for data races (Figure 5(a)). Similarly, a `cached_read` operation has a causal edge to its `cr` node in the cache, which denotes the cache supplying the requested data, and which has a causal edge to the next instruction in that thread (because the thread must stall until receiving data). The `cr` node has a causal edge *from* the `alloc` node that allocated this cache line from main memory, and it is the `alloc` node that is checked for data races (Figure 5(b)). Clearly, this model is general enough to encompass the behavior of any cache, regardless of its details — data must enter a cache somehow before it can be read, and dirty data must be written back eventually. Therefore, any data race that could happen with a specific cache architecture can also happen with this abstract cache model. Flushing is similar to sync: the `cache_flush` node takes its place in the program order, but has causal edges from all `wb` nodes for prior `cw` nodes, and to all `alloc` nodes for subsequent `cr` nodes, to the same address. Multiple threads/CPUs in a single cache-coherent domain can be modeled as if accessing a single, shared cache.

3.4 Avoiding Combinatorial Explosion of Happens-Before Graphs

The preceding abstract cache model captures memory operations that must occur *in some form*. However, in a real system, each memory read doesn't produce a separate cache line allocation,



(a) Between two consecutive cached reads to the same address, there may or may not have been an eviction. If address a was evicted before the second read, then there is a second `alloc` node (dashed lines) when the cache line is (re-)allocated. (The dotted lines are shorthand that there may be other instructions in between.)



(b) If address a was not evicted before the second read, then there is no second `alloc` node. Without an excessively detailed model of the cache, there's no way to know which happens-before graph to generate. Theorem 3.1 says that it is safe (data-race preserving) to ignore this second case and consider only the first case (6a). The proof intuition is that the second `alloc` node doesn't create any additional ordering in the graph, other than with itself. (The second `cr` node is already ordered after the first `alloc` node, by transitivity.) Therefore, if there were a race with the first `alloc` node, then adding the second `alloc` node can't eliminate the race. In other words, abstracting from both graphs into a single graph preserves the presence of data races.

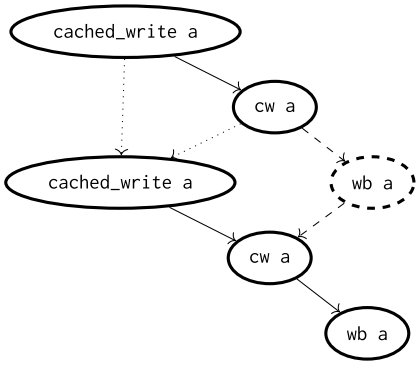
Fig. 6. Two possible happens-before graphs for a cached read followed by a cached read.

and each memory write doesn't produce a separate writeback. The abstract cache model provides no guidance as to what did or did not happen, so we add a second layer of abstraction, to prevent a combinatorial blow-up in the number of happens-before graphs to analyze.

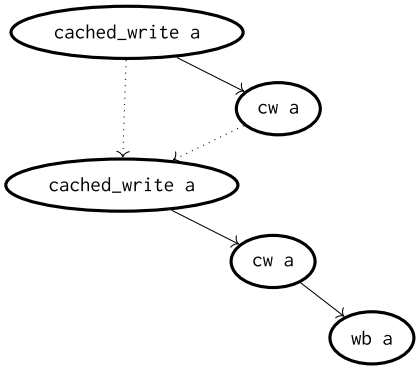
Specifically, the challenge is that without modeling excessive details, it is unknowable when (or even if) certain cache line allocations or writebacks occur. For example, for a cached read, the CPU's `cached_read` node generates a causal edge to a `cr` node, which has a causal edge from an `alloc` node, because the cache line must have been allocated before the value can be returned to the CPU. But maybe that particular `alloc` didn't happen, because the cache line was already in the cache and might not have been evicted (e.g., Figure 6).

There are four cases to consider:

(1) The first case, a cached read followed later by a cached write to the same address, requires no special handling, because the basic cache abstraction for the read does not interact with the basic cache abstraction for the write. (An allocate-on-write cache can be easily modeled by simply inserting a read of the entire cache line before performing the write, which then triggers the second case below.)



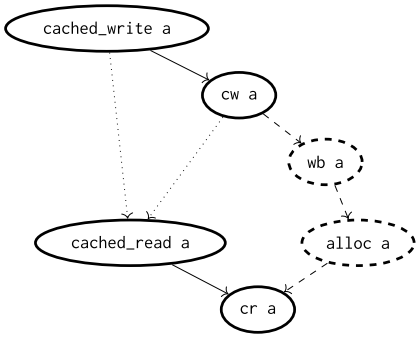
(a) Similarly to Fig. 6, between two consecutive cached writes to the same address, there may or may not have been a writeback. Above is with a writeback (dashed lines). (The dotted lines are shorthand that there may be other instructions in between.)



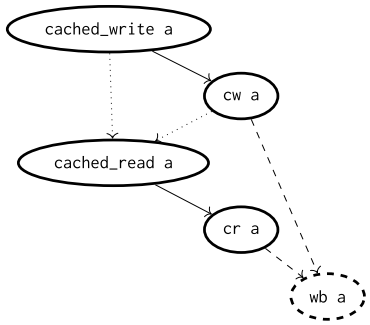
(b) This case is without the first writeback. Theorem 3.2 says that it is safe (data-race preserving) to ignore this second case and consider only the first case (7a). Similar to the intuition for Theorem 3.1, the first wb node doesn't create any additional ordering in the graph, other than with itself. (The first cw node is already ordered before the second wb node, by transitivity.) Therefore, if there were a race with the second wb node, then adding the first wb node can't eliminate the race.

Fig. 7. Two possible happens-before graphs for a cached write followed by a cached write.

(2) The second case is a cached read followed by another cached read (to the same address) (Figure 6). Because we don't know whether the cache line had been evicted after the first `alloc`, we don't know whether the second `alloc` happened or not. Which nodes/edges do we add to the happens-before graph? For efficiency, we must avoid case-splitting, which would create an exponential (in the number of memory operations) number of happens-before graphs to analyze. Fortunately, Theorem 3.1 (below) establishes that it is safe (i.e., data-race preserving) to build only the graph with both `alloc` nodes.
(3) A cached write followed by a cached write produces an analogous situation (Figure 7). Did the first write's wb happen? Theorem 3.2 establishes that it is safe to build only the graph with both wb nodes. As with the preceding case, we preserve data races and avoid a combinatorial explosion in happens-before graphs.
(4) The most interesting case is a cached write followed by a cached read (Figure 8). Did the writeback happen before the cached read occurs? If so, then the cached read needs an `alloc`



(a) Similarly to Fig. 7, above is the case in which the writeback happened, followed by an alloc node when the data comes back into cache (dashed lines). Not shown is a similar case, in which a writeback occurred, but the data remained clean in the cache (e.g., transitioning from Modified to Shared in a cache coherence protocol), which gives the same graph as above, except without the alloc node.



(b) This case is when the writeback happened after the cached read (from a dirty cacheline). Because of this assumption, we add an edge from the `cr` node to the `wb` node. Unfortunately, neither graph is a safe abstraction of the other.

Fig. 8. Two possible happens-before graphs for a cached write followed by a cached read.

node; if not, then the cache line is still dirty, and the `cr` node happens before the `wb` node. In this case, unlike in the preceding cases, neither graph is a safe abstraction of the other. Our solution is to create a graph that is a safe abstraction of *both* situations, even though it doesn't correspond to actual cache behavior (Figure 9). Theorem 3.3 establishes that Figure 9 is a safe abstraction of both graphs in Figure 8. Again, we preserve data races and avoid a combinatorial explosion in happens-before graphs.

We now formalize the above intuition and state and prove the theorems. We will assume happens-before graphs with operation nodes as shown in Table 1.

For notational convenience, we define predicate datarace(x, y, G) to mean that x and y are vertices in directed graph G, and they are in a data race, i.e., x and y are reads/writes to shared memory and at least one is a write, and there is no directed path in G from x to y or vice-versa. We also define datarace(G) to mean that there exist two vertices x and y in G such that datarace(x, y, G).

For a given execution of a heterogeneous system with non-coherent accelerator, let T be the true happens-before graph for this execution, modeled using the node types in Table 1 and the basic visible abstract cache constructions as in Section 3.3. Let F be the happens-before graph after fully applying the abstractions in this section. We seek to establish that datarace$(T) \Rightarrow$ datarace(F),

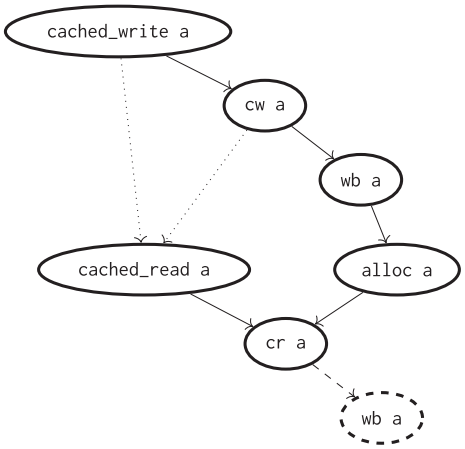


Fig. 9. A safe abstraction of the happens-before graphs for a cached write followed by a cached read. Theorem 3.3 says that this is a safe abstraction of all graphs in Figure 8. The intuition is that assuming the writeback happened means adding the `alloc` node, which, unlike the earlier proofs, does create new ordering between nodes, specifically that `wb` → `alloc` → `cr`. This means that adding the `alloc` node might eliminate a race with the original `wb` node, if the race node happens after the `cr` node. However, any such node would now have a race with the newly created, second `wb` node that happens after the `cr` node. So, this construction is also race-preserving. Because of the second `wb` node (dashed), the cached read should be treated as a cached write when the next memory operation to the same address is added to the happens-before graph. The second `wb` node can be thought of as an abstract flag indicating that the cache line *may* be dirty.

i.e., if there is a data race in the true happens-before graph, then there is still a data race in the happens-before graph after fully applying the abstractions in this section.

The overall proof that $\text{datarace}(T) \Rightarrow \text{datarace}(F)$ is established by defining a sequence of happens-before graphs $T_0, \ldots, T_n$, with $T = T_0$ and $T_n = F$. We generate T_{i+1} by applying one of the abstractions from Figures 6–9 to any of the earliest cached memory operations in T_i that has not yet had an abstraction applied. The crux of the argument is proving the invariant that $\text{datarace}(T_i) \Rightarrow \text{datarace}(T_{i+1})$, which we do below. A trivial induction on i completes the proof that $\text{datarace}(T) \Rightarrow \text{datarace}(F)$.

Suppose we are given happens-before graph T_i such that $\text{datarace}(T_i)$. For some arbitrary address a, we pick the earliest cached memory operation on a that hasn't been abstracted yet. The intuition given above described the four cases as {read, write} followed by {read, write}. However, it will be cleaner to formalize this as {possibly-dirty, not-dirty} followed by {read, write}. In the possibly-dirty case, there is a `wb` node in the graph from the last abstracted cache node (`cr` or `cw`) on a; in the not-dirty case, there is no such node.

The first case — where the cache is not dirty (clean or invalid) for address a just before the earliest unabstracted cached memory operation on a, which is a write — is trivial. No abstraction is performed, so $T_i = T_{i+1}$ and therefore $\text{datarace}(T_i) \Rightarrow \text{datarace}(T_{i+1})$.

The second case (not-dirty followed by a cached read, as shown in Figure 6) corresponds to Theorem 3.1.

Theorem 3.1. *Let T_i be defined as above. If the last abstracted cached memory operation in T_i on some memory address a does not have a* `wb` *node (which means it's not-dirty), the earliest unabstracted cached memory operation in T_i on address a is a read, and T_{i+1} is created by replacing the nodes in T_i for this operation with nodes as shown in Figure 6(a), then $\text{datarace}(T_i) \Rightarrow \text{datarace}(T_{i+1})$.*

Proof. The unabstracted read operation either did or did not have an `alloc` node in the true happens-before graph T.

If it did have an `alloc` node in T, then it also has one in T_i (because it hasn't been abstracted yet), and the construction in Figure 6(a) keeps this `alloc` node and makes no other changes. Therefore, $T_i = T_{i+1}$ and datarace$(T_i) \Rightarrow$ datarace(T_{i+1}).

On the other hand, if it did not have an `alloc` node in T, then it also doesn't have one in T_i (because it hasn't been abstracted yet), and the construction in Figure 6(a) adds an `alloc` node and makes no other changes. If datarace(T_i), then there are two nodes x and y in T_i such that $x \not\mapsto y$ and $y \not\mapsto x$ in T_i. Neither x or y can be the new `alloc` node, because that node doesn't exist in T_i. Can the introduction of this `alloc` node create a new path from x to y or vice-versa? The answer is no: the new node only connects an earlier `alloc` node to the new `cr` node, but they were already connected via transitivity throught the program order. Therefore, $x \not\mapsto y$ and $y \not\mapsto x$ in T_{i+1}, so datarace(x, y, T_{i+1}), which means datarace$(T_i) \Rightarrow$ datarace(T_{i+1}) in this case as well. □

The third case (possibly-dirty followed by a cached write, as shown in Figure 7) corresponds to Theorem 3.2.

Theorem 3.2. *Let T_i be defined as above. If the last abstracted cached memory operation in T_i on some memory address a has a* `wb` *node (which means it's possibly-dirty), the earliest unabstracted cached memory operation in T_i on address a is a write, and T_{i+1} is created by replacing the nodes in T_i for this operation with nodes as shown in Figure 7(a), then datarace$(T_i) \Rightarrow$ datarace(T_{i+1}).*

Proof. The `wb` node from the last abstracted cached memory operation either did or did not happen before the unabstracted `cw` node in the true happens-before graph T.

If it did happen before the unabstracted `cw` node in T, then it also happened before the unabstracted `cw` node in T_i (because it hasn't been abstracted yet), and the construction in Figure 7(a) keeps this `wb` node in the same order and makes no other changes. Therefore, $T_i = T_{i+1}$ and datarace$(T_i) \Rightarrow$ datarace(T_{i+1}).

On the other hand, if it did not happen before the unabstracted `cw` node in T, then it also did not happen before the unabstracted `cw` node in T_i (because it hasn't been abstracted yet), so T_i will have a structure like Figure 7(b), where the same `wb` node serves multiple `cw` nodes. The construction says to substitute the structure in Figure 7(a) instead, which would add an extra `wb` node to T_{i+1}. If datarace(T_i), then there are two nodes x and y in T_i such that $x \not\mapsto y$ and $y \not\mapsto x$ in T_i. Neither x nor y can be the new `wb` node, because that node doesn't exist in T_i. Can the introduction of this `wb` node create a new path from x to y or vice-versa? Just as in the preceding proof, the answer is no: the new node only connects the earlier `cw` node (or `cr` node, see the next case) to the new `cw` node, but they were already connected via transitivity through the program order. Therefore, $x \not\mapsto y$ and $y \not\mapsto x$ in T_{i+1}, so datarace(x, y, T_{i+1}), which means datarace$(T_i) \Rightarrow$ datarace(T_{i+1}) in this case as well. □

The last case (possibly-dirty followed by a cached read, as shown in Figure 8) corresponds to Theorem 3.3.

Theorem 3.3. *Let T_i be defined as above. If the last abstracted cached memory operation in T_i on some memory address a has a* `wb` *node (which means it's possibly-dirty), the earliest unabstracted cached memory operation in T_i on address a is a read, and T_{i+1} is created by replacing the nodes in T_i for this operation with nodes as shown in Figure 9, then datarace$(T_i) \Rightarrow$ datarace(T_{i+1}).*

Proof. This is the most complex case, but the reasoning is similar to the preceding proofs. The configuration in T_i must be as in Figure 8(a) (either with or without the `alloc` node) or Figure 8(b).

If the situation is Figure 8(a) with the alloc node, then T_{i+1} adds only a single wb node. This can't eliminate any races, so datarace(T_i) ⇒ datarace(T_{i+1}) in this case.

If the situation is Figure 8(a) without the alloc node, then T_{i+1} adds the same wb node as in the preceding case, with the same inability to eliminate data races. It also adds an alloc node, which *does* add ordering constraints to T_{i+1}, specifically that wb → alloc → cr. The only new paths this might introduce to the graph (and hence the only data races that might be lost) are from the wb node to the cr node and any nodes reachable from there. However, any node x that is reachable from the cr node, but unreachable from the wb node, in T_i will be unreachable from the newly added wb node in T_{i+1}. Thus, any data races that were lost will be replaced by new ones, and we still have datarace(T_i) ⇒ datarace(T_{i+1}) in this case.

If the situation is Figure 8(b), the newly added wb and alloc nodes do not create any new paths between other nodes, since the cw nodes was already connected to the cr node. This can't eliminate any races, so datarace(T_i) ⇒ datarace(T_{i+1}) in this case as well. □

With these three theorems for the three non-trivial cases in the invariant proof, we can conclude that if there is a data race in the true happens-before graph T, then there is guaranteed to be a data race in the happens-before graph F, in which the abstractions have been fully applied. The preceding subsection made data races resulting from cache behavior visible; this subsection guarantees that any data races are preserved even though we construct only a single happens-before graph.

3.5 Active Frontier

There is one more challenge for our theoretical framework. Even a few seconds of execution might generate billions of memory operations, meaning the full happens-before graph can be intractably large. To make our analysis scalable, it must be "on-the-fly", building the graph incrementally as it reads the trace file of memory operations, and just as importantly, deleting older nodes when they become irrelevant. So, rather than building the entire happens-before graph, the analysis maintains only an "active frontier" of nodes that might matter for data races.

For node creation, we maintain an as-soon-as-possible property: no node is created in the graph until the earliest point in time when that operation could execute (i.e., when all nodes that happen before it have already been created), *and the value being read or written is known to be visible to the CPU or accelerator.* The first part of this property is natural and the same as in classical data race analysis: no node should be added to the happens-before graph until every node that it depends on (that must have happened before it) has already been added to the graph. The italicized part is new to our abstractions, and means that an alloc node is created only when the corresponding cached_read node is created. Note that even though the alloc node is created late, it has a happens-before edge *to* the cr node created along with the cached_read node, so even though the alloc node isn't created until the cr node that needs it, the alloc node still captures in the happens-before graph all relevant executions in which this cache line allocation happened before this read, e.g., it might be allocated just before the cr due to a cache miss, or it might have been pre-fetched long before. So, the late creation of the alloc node does not eliminate any observable data races. What the late creation of the alloc node does eliminate are many spurious "data races" in which the racy value loaded into the cache is never observed. For example, without the italicized part of this property, *every* program would have data races as soon as any variable is initialized: an alloc node could have been created for that variable's location at the beginning of execution (because the alloc node has no in-edges), which might or might not have loaded the pre-initialization

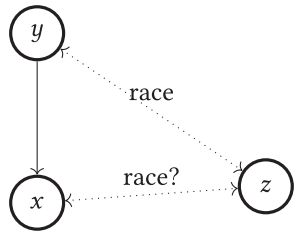


Fig. 10. Illustration of Lemma 3.4. Node x is a write node newly added to the active frontier, and node y is a memory operation to the same address that was already in the active frontier. If we delete y from the active frontier, might we miss a race between y and a node z added later? The answer is that we will still detect a race (between x and z instead of between y and z): $z \not\to x$ because x was added before z, and $x \not\to z$ or else $y \to x \to z$, which contradicts that y and z are in a race.

value of the variable into cache. However, such "races" are meaningless because those values are overwritten before the CPU uses them.

The policy for node deletion is more complex. We maintain a set of "marked" nodes, which is a set of operations that read or write shared memory, checking against which is sufficient to detect a race if the full graph had any races. When we add a new node to the graph, we check for any races with the newly added nodes, then update the set of marked nodes according to the following rules:

(1) If a newly added node x is a write to shared memory, mark node x, and unmark any other read or write node y to the same address.
(2) If a newly added node x is a read to shared memory, mark node x, and unmark any other read node y to the same address with $y \to x$.

Next, any node that is not either marked or reachable from a marked node via happens-before edges is deleted.

To have a sound data race analysis, the crucial result we need to establish is that the active frontier created via our node creation and deletion policies preserves data races. Formally, let T be the true happens-before graph, and let F be the happens-before graph after fully applying the abstractions, as defined in Section 3.4. We have already established that $\text{datarace}(T) \Rightarrow \text{datarace}(F)$. Now, let $A_0, A_1, \ldots$ be the sequence of active frontier happens-before graphs as the online algorithm processes the execution log. Our main result is to show that $\text{datarace}(F) \Rightarrow (\exists i.\text{datarace}(A_i))$. In other words, if there is a data race, then there will be a data race in at least one active frontier (and hence we will detect it).

Node deletion is the central problem for proving the theorems. Without node deletion, the sequence of active frontier graphs $A_0, A_1, \ldots$ becomes the full happens-before graph F when the trace file of memory operations has been completely processed. Therefore, we first establish lemmas for the node marking and deletion rules.

Lemma 3.4. *If a newly added node x is a write to shared memory, we can mark node x, and unmark any other read or write node y to the same address and not lose the ability to detect races in the execution.*

Proof. (Figure 10 illustrates the intuition behind this proof.) Suppose x is newly added to active frontier A_i, and y is any other node already in A_i. First, if neither $x \to y$ nor $y \to x$, then x and y are in a race, which we would detect immediately upon adding x, and hence we do not lose the ability to detect races in the execution, because $\text{datarace}(A_i)$. Now, is it possible that $x \to y$? The answer is no, by the as-soon-as-possible property: the edges from $x \to y$ can't be program order edges

(because y was already in the graph before x) and can't be causal edges (because y couldn't have happened without x in the graph). Therefore, the only remaining case is when $y \rightarrow x$. Suppose that later in the memory trace, we add a memory operation z to a new active frontier A_j, and that datarace(y, z, F), i.e., there's a data race between y and z in the full happens-before graph F. The fear is that since we unmarked y when creating A_i, it may have been deleted before A_j, in which case $\neg$datarace(y, z, A_j) and we miss this data race. But fortunately, by the same as-soon-as-possible property, we know that $z \not\rightarrow x$, even in the full graph F. On the other hand, if $x \rightarrow z$, then we have $y \rightarrow x \rightarrow z$, which contradicts the fact that datarace(y, z, F). Hence, $x \not\rightarrow z$ and $z \not\rightarrow x$, which means x and z are in a race. Therefore, we will still detect a race in A_j (between x and z), even if we deleted y from the active frontier. □

For read nodes, the ordering relationships are weaker, so we have a weaker result.

Lemma 3.5. *If a newly added node x is a read to shared memory, we can mark node x, and unmark any other read node y to the same address with $y \rightarrow x$, and not lose the ability to detect races in the execution.*

Proof. Similar to the proof above, suppose that in a later active frame A_j, we add a memory operation z (which must be a write) that's in a race with a node y that we unmarked when constructing A_i. By the same arguments as above, we know that $z \not\rightarrow x$ (because x was already in the graph) and $x \not\rightarrow z$ (else $y \rightarrow x \rightarrow z$ which contradicts that y and z are in a race), which means x and z are in a race. So, we will still detect a race. □

Lemma 3.6. *We can delete any unmarked node y and not lose the ability to detect races in the execution.*

Proof. If y is a read or write to shared memory, we have already established that if it's unmarked, it's safe to delete: reads and writes to shared memory are always marked upon entry to an active frontier, and they only become unmarked via the processes in Lemmas 3.4 and 3.5. If y is not a read or write to shared memory, then it can't be part of a race itself. Deleting y therefore can only reduce the amount of ordering in the happens-before relation, so the set of races can only stay the same or increase. □

With these lemmas, we can easily establish the main theorem of this subsection:

Theorem 3.7. *If there is a data race in the full happens-before graph, then there will be a data race in at least one active frontier (and hence we will detect it), i.e., $datarace(T) \Rightarrow datarace(F) \Rightarrow (\exists i.datarace(A_i))$.*

Proof. The result follows directly from Lemma 3.6, since we always preserve the existence of races as we maintain the active frontier by deleting unmarked nodes. □

The preceding theorem states that our method is sound (guaranteed to find races if they exist), even if we delete *all* unmarked nodes from the active frontier. However, our analysis also keeps in the active frontier the nodes that happen after the marked nodes. Doing so let's us establish a completeness result, that our active-frontier simplification does not lose precision:

Theorem 3.8. *If the online algorithm flags a data race, then the same race exists in the full happens-before graph. (This implies that $(\exists i.datarace(A_i)) \Rightarrow datarace(T)$.)*

Proof. (Figure 11 illustrates the intuition behind this proof.) Let node z be a newly added memory operation that triggers a race, and let node y be the pre-existing shared memory operation in the active frontier that it is in a race with. By definition, $z \not\rightarrow y$ and $y \not\rightarrow z$ in the active frontier, and we need to prove that $z \not\rightarrow y$ and $y \not\rightarrow z$ in the full happens-before graph. By the same

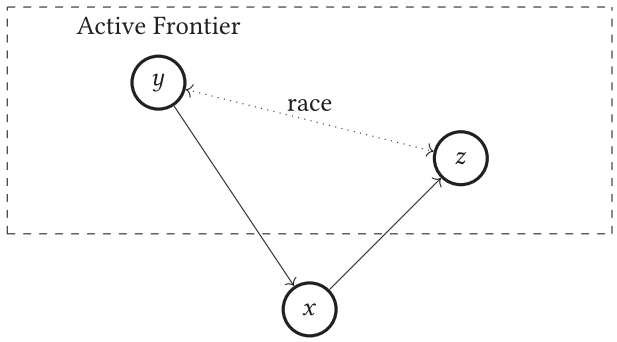


Fig. 11. Illustration of Theorem 3.8. Node z is a newly added memory operation, and the analysis detects a race between z and existing node y in the active frontier. Could it be that y and z are *not* in a race if we considered the full happens-before graph? The answer is no. It's easy to see that $z \not\to y$ in the full graph, because y was added to the graph before z. In the other direction, if $y \to z$ in the full graph but not in the active frontier, there must be a node x not in the active frontier, with $y \to x \to z$. However, x must have been added before z, because $x \to z$, and x could not have been deleted from the active frontier, because $y \to x$. Thus, any race detected in the active frontier is also a race in the full happens-before graph.

as-soon-as-possible argument we've used already, we know that $z \not\to y$ in the full graph, because if $z \to y$, then y couldn't have been created before z. To establish that $y \not\to z$ in the full graph, assume the opposite, that $y \to z$ in the full graph. By Lemma 3.6, we can assume without loss of generality that y is marked. Now, we know that $y \not\to z$ in the active frontier, so if $y \to z$ in the full graph, there must be a node x in the full graph such that $y \to x \to z$ in the full graph. Because $x \to z$, the node x must have been added to the active frontier before z. Because y is marked and $y \to x$, the node x cannot have been deleted from the active frontier. Therefore, $y \to x \to z$ in the active frontier as well, which contradicts that y and z are in a race in the active frontier. Therefore, both $z \not\to y$ and $y \not\to z$ in the full graph, and the same race exists in the full graph as in the active frontier. □

Combining all of the results in Section 3, these theorems allow us to abstract away cache details like associativity, eviction and replacement policies, and pre-fetching; to build and analyze a single happens-before graph instead of case-splitting on possible cache behaviors; to prune the happens-before graph down to tractable active frontiers; and yet still be guaranteed to detect any possible races.

4 EXPERIMENTAL RESULTS

4.1 A Dynamic Race Detector

As proof-of-concept for our theory, we built a simple prototype *dynamic race detector*. A dynamic race detector (e.g., Eraser [19], FastTrack [8], SPD3 [18]) detects *any possible* data race (even if it didn't execute in an unexpected order) in a *single* execution of a program. It represents a practical compromise between fully formal static verification, which detects any possible data race in *all possible executions* of a program, and conventional software testing, which detects *only data races which went the "wrong way" and produced observably incorrect results* in a single execution of a program. Compared to static formal verification, dynamic race detection is more scalable (unless the static formal verification is highly abstracted), and avoids the need to perform complicated (and generally imprecise) alias analysis, because the actual trace of memory accesses is known. However, like conventional software testing, attention must be paid to code coverage, to exercise

as many program paths as possible. From an experimental perspective, dynamic race detection is a pure and direct evaluation of the precision and effectiveness of our abstraction, without the confounding influences of which formal verification algorithms we use to enumerate program paths, what other abstractions we might employ, and how lossy we make the joins in our analysis.

Note that our race detector is rather rudimentary, being a straightforward tracking of the happens-before graph. We do not pretend that it is state-of-the-art. The novelty is that it implements our model of abstracted cache behavior, so that it can soundly detect data races in the heterogeneous system with mixed cached and uncached memory accesses. Our analysis is proven sound, so the main empirical question is whether the analysis is precise enough to avoid excessive false positives. Another important question is whether real code has data races caused by the interaction of cached and uncached memory accesses. We built the tool only to answer these two questions.

Our race detector is for the CPU/accelerator system in Figure 2. The race detector processes a trace of the memory accesses from a program execution. How to derive such traces is well-established (e.g., in debuggers and other analysis tools), but is labor-intensive to implement, so for our proof-of-concept, we instrumented our test programs manually to print out each memory operation as it executes.[5] To improve efficiency, we condense sequences of identical operations on consecutive addresses into a single operation on a memory range. When each memory operation is added, it is checked to see whether it is ordered with respect to all other memory operations that touch the same address range. If two operations are unordered, and at least one is a write, then the tool flags the data race and exits.[6]

4.2 Experimental Setup

To test our tool on real-world examples, we selected eleven open-source examples from the VectorBlox MXP SDK. The examples in the GitHub repository were chosen to be used in our experiments if they were non-trivial and were written in C. As is typical of compute kernels, the example programs are loop/vector-heavy, but have minimal branching structure and no input-dependent behavior. Therefore, the examples do not stress the weakness of *dynamic* data race detection — which might not cover all CPU program paths — and let us focus on the proven soundness of our analysis over a single CPU program path. Table 2 summarizes the examples.

All the examples were executed using the VectorBlox MXP simulator; it models a single-core CPU with one MXP vector accelerator. We analyzed each program in its original form, and also, because the examples were provided in the vendor's SDK and presumably race-free, we introduced races by removing necessary synchronization (e.g., sync statements, allowing shared variables to be cached with flushing, etc.). Because our analysis is sound, these race-injection examples were just sanity checks of our code and were done in an ad hoc manner. The primary questions in our experiments were whether the runtime was prohibitively large, and the rate of false positives, i.e., detected data races that were not really data races.

[5]Instrumenting at the source-code level does mean that our proof-of-concept implementation assumes sequential consistency of the source language memory model. This is a limitation of the implementation only, not the theory. Instrumenting the binary would bypass this problem, or one could use SC-preserving compilation (e.g., [15]).

[6]Our example CPU/accelerator system can be configured for several different CPU architectures, and these architectures vary in how they handle uncached memory accesses. For example, the Nios II has uncached read/write instructions that are allowed to incoherently bypass the cache; ARM makes cacheability a property of an address, rather than an access; and MicroBlaze specifies writethrough caches, rendering the distinction moot. Our tool has a flag for whether to include CPU data races between cached and uncached accesses (assuming Nios-II-style semantics) or flag only data races between CPU and accelerator. Given our emphasis on heterogeneous systems, our experiments use the latter setting.

Table 2. Test Programs

Test Name	Test Description	SLoC
vbw_libfixmath	Square root and division	190
vbw_mtx_fir	2D FIR filter using matrices	118
vbw_mtx_median_argb32	Median filter with 32-bit data type	117
vbw_mtx_median	Median filter with 8-bit data type	119
vbw_mtx_motest *	Motion estimation	228
vbw_mtx_sobel	Sobel filter	154
vbw_mtx_xp	Matrix transpose	287
vbw_vec_add *	Vector addition	122
vbw_vec_fft	FFT	2678
vbw_vec_fir	FIR filter using vectors	62
vbw_vec_power	Vector exponentiation	116

We evaluated our analysis on these eleven programs, selected from the accelerator vendor's SDK. Our selection criterion was all non-trivial programs written in C. "SLoC" indicates source lines of code. The asterisks (*) indicate examples where our analysis discovered previously unknown data races.

4.3 Time Required for Analysis

As noted already, our implementation was rudimentary, as it was not our goal to attain state-of-the-art performance. The shortest runtime took 30ms for 6,013 lines in the trace file (`vbw_libfixmath`), and the longest runtime took approximately nine days for almost 29 million lines in the trace (`vbw_mtx_sobel`). Ten out of the 11 examples completed in less than 45 minutes each. Figure 12 plots the analysis runtime for examples without races. (Examples with races completed quickly because the tool stops at the first race detected.)

4.4 Data Race Detection Results

The experiments with injected races were unremarkable. The tool was able to detect the injected races easily in all cases, which was unsurprising since the analysis is sound and the programs had minimal branching structure. Interestingly, in some cases, the system simulator still showed the test as passing, even though we had eliminated necessary synchronization, because the race went the right way by chance. This highlights how data races can produce elusive bugs, which are not caught in debugging, but emerge only late (and sporadically) in a production system, underscoring the importance of data race detection tools.

Whenever the tool flagged a data race, we analyzed the reported race manually to determine whether it was a real race or an artifact introduced by our abstractions. Happily, we had zero false positives: every time the tool flagged a possible data race, it was indeed a real data race. Hence, despite our safe abstractions, our tool achieved 100% precision (albeit on a small test set).

False negatives are harder to quantify. No other tool models the data races that we are targeting, so there is no independent way to confirm race-freedom. Manually, it's much harder to verify a program race-free than simply to check that an identified race is really a race. Fortunately, the test programs have fixed inputs, so technically, there is only one program path possible. (There are, of course, many possible interleavings of concurrent actions.) Hence, since our method is proven sound with respect to the program path that executed, and since there is only on program path possible, there should be zero false negatives. Empirically, on the examples with injected races, we detected races every time, so there were no false negatives, but no true negatives either. Overall, we never observed a known false negative, and the theory predicts that none should be possible.

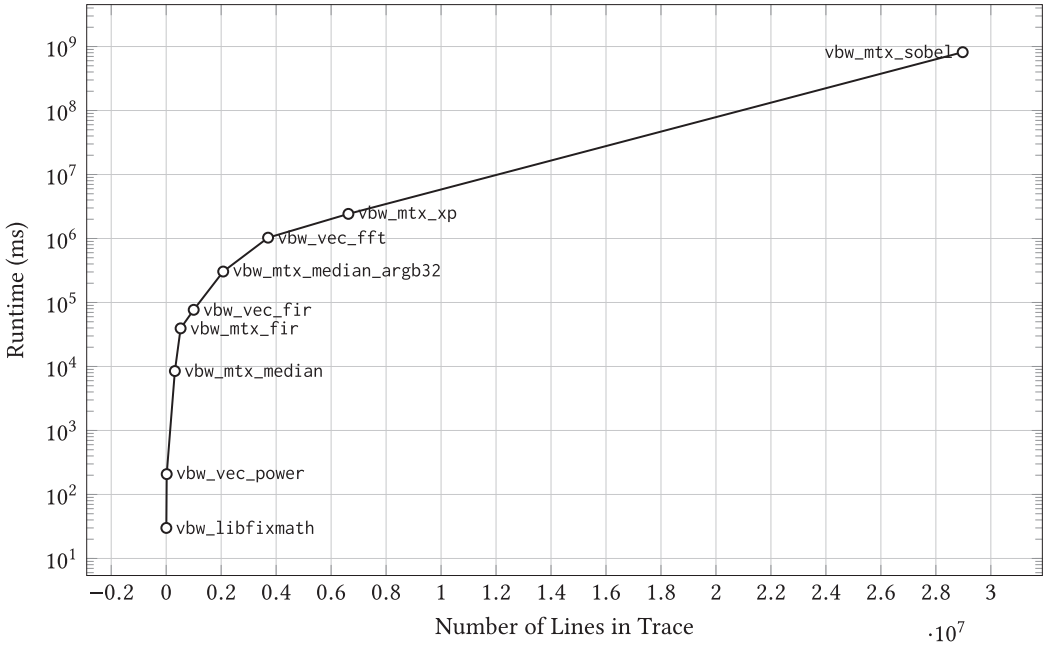


Fig. 12. Analysis runtime of examples without data races. (All examples with data races terminated quickly.) As noted in the text, our implementation is a straightforward tracking of the happens-before graph, so we make no claims of stellar performance. Nevertheless, this semi-log plot shows runtime growing sub-exponentially in the number of lines in the trace. With our abstraction of cache behavior, runtime is not prohibitively expensive, even with a rudimentary implementation.

Remarkably, the tool found subtle, previously unknown races in two out of the eleven examples (in their original form, without injected races). One example is actually the vector addition example `vbw_vec_add` in Figure 1. The tool detects a potential race at the first DMA read from main memory (line 23) vs. a writeback node for the cached writes at line 13:

```
test_zero_array( scalar_out, N );
...
vbx_dma_to_vector( v_in1, (void *)vector_in1, N*sizeof(vbx_sp_t) );
```

At the source-code level, the bug is not obvious: the conflicting variables are `scalar_out` and `vector_in1`, which are two completely different variables. Closer inspection of the code reveals that `scalar_out` is a cached vector that eventually has its values written back when the test zeroes this array, and `vector_in1` is an uncached vector in main memory that is accessed by the accelerator via DMA. In this case, the compiler for the VectorBlox simulator laid out memory such that the end of `scalar_out` mapped to the same cache line as the beginning of `vector_in1`. Thus, although the source code is careful to use uncached reads and writes on `vector_in1`, the write-backs for `scalar_out` can overwrite the first few bytes of `vector_in1` in main memory, due to writeback granularity. The DMA read may thus read wrong data values. (This data race is similar to the problem of false sharing, but this is a correctness bug rather than a performance issue.) Such a data race is completely invisible to prior data race detectors, which don't model the interaction between caches and non-coherent accesses. In practice, the race might be impossible on some platforms, yet appear unpredictably on others, making the code non-portable and nearly impossible to debug. Yet, our analysis uncovered the bug easily.

The other data race detected had the same root cause: in the `vbw_mtx_motest` example, the compiler had allocated memory in such a manner that writebacks from cached memory affected regions of memory that were supposed to be accessed only via uncached reads and writes. Once discovered, the bugs are easily fixed via careful data alignment, but both bugs were previously undiscovered in the SDK. These examples highlight the subtlety of data races for heterogeneous systems, and the ability of our analysis to find them.

5 RELATED WORK

As noted in the introduction, there is an extensive body of research on data race detection. Here, we briefly mention a few of the more influential and relevant lines of research.

Data race detectors can be broadly grouped into those based on locksets versus those that reason about the happens-before relation, plus hybrids of the two. Lockset approaches presume that any concurrent object should be protected by a lock and compare the sets of locks guaranteed to be held by concurrent accesses. Representative lockset-based data race detectors include Warlock [23], Eraser [19], RacerX [6], RELAY [26], and Locksmith [17]. Lockset approaches typically exhibit superb scalability, but suffer from imprecision and false positives. Locksets also reflect a higher-level view of memory accesses and don't reflect the data races in low-level code that concern us. Type-based analyses (e.g., [1, 7]) can be viewed as generalizations and extensions of lockset-based approaches.

Pure happens-before reasoning tends to be too slow, given its low-level view of tracking of all memory operations. State-of-the-art tools typically use happens-before reasoning for low-level precision, but combine it with more sophisticated, higher-level analyses for efficiency. Examples of such hybrids include the methods of O'Callahan and Choi [16], which explore a wide variety of lightweight static analyses combined with happens-before reasoning; FastTrack [8], which uses an adaptive representation for the happens-before relation that requires only constant space for common cases (thread local, lock protected, read shared) without loss of precision; and IFRit [5], that uses static analysis to determine interference-free regions to eliminate most potential data races. LiteRace [14] is also happens-before-based, but pioneered the use sampling, focusing the analysis only on portions of the code that have not been executed extensively. (IFRit also does some sampling.)

Our prototype data race detector uses pure happens-before reasoning and is definitely not state-of-the-art. However, the novelty is our abstraction of cache behavior, to allow reasoning about and detecting data races involving interaction between caches and non-coherent memory accesses. This is an issue not addressed by prior work.

Closer in spirit to our work, GRace [27] is a data race detector for GPU programs. Like our work, their focus is on data races and hardware accelerators. Unlike our work, their focus is exclusively on data races *within* the accelerator itself, whereas we focus exclusively on data races occurring from the interaction between cached memory and uncached accelerators.

The specification of memory models for shared-memory multiprocessors has some similarities to our problem. Both problems require reasoning about the ordering of operations on a shared memory in the presense of complicated hardware optimizations. For that problem, the research community has gravitated towards axiomatic specifications that relax typical ordering constraints in subtle ways, to permit the behaviors exhibited by high-performance microarchitectures (e.g., [2] is a classic survey, and we have already cited [4], which employs such an approach for the ARM and Power ISAs). Such a solution is appropriate for that problem, since the memory model is part of the ISA, so it is desirable to have underspecified behavior to allow future optimizations, and there are few different ISAs, so only a few different memory models need be formalized. In contrast, for modern, accelerator-rich systems, every configuration might have different data races,

so the task of specifying correct behavior must be more precise, yet less laborious than for general memory models. Accordingly, we follow a different direction: rather than try to create a more complex ordering formalism to abstract what a broad class of microarchitectures might do, we simply follow the microarchitecture of the specific caches, but provide a safe abstraction to prevent a combinatorial explosion of possible happens-before graphs.

6 CONCLUSION AND FUTURE WORK

We have introduced the first systematic approach to finding data races arising from the interaction of cached memory accesses and non-coherent memory accesses, as arise in heterogeneous systems with non-coherent accelerators. The key contribution is a novel abstraction for cache behavior. We formally prove the abstraction sound (i.e., any data race in an execution is guaranteed to be detected), and empirically demonstrate that even in a basic implementation, it is scalable enough to be useful, yielded zero false positives, and discovered two previously unknown data races.

The obvious direction for future work is to try embedding our abstraction into a state-of-the-art dynamic race detector, or conversely, to incorporate techniques from state-of-the-art dynamic race detectors into ours. The benefits would be greatly improved performance and scalability over our implementation, and/or the novel capability to detect an emerging class of data races for the state-of-the-art race detector. In general, this is potentially possible for any race detector that reasons about the happens-before graph. Specifically revisiting the hybrid dynamic race detectors cited in Section 5, we find that unfortunately, some tools are targeted at a much high-level view of concurrency, and some optimizations appear to be difficult to reconcile with our abstraction, but fortunately, most techniques appear readily compatible with ours. For example, O'Callahan and Choi [16] focus specifically on Java threads, and their methods exploit the assumption that at least some synchronization is lock-based. However, if we make the same assumption of some lock-based synchronization, we could employ their same lockset-based redundancy checks (lockset-subset and oversized-lockset) to prune some memory operations from the happens-before graph. Similarly, FastTrack [8] also presumes a higher, software-level view of concurrent threads, but if such information were available, their notion of "epochs" holds promise for pruning the happens-before graph and improving scalability. For example, they track only the latest write to a given address a, as we do in our active frontier optimization, but they have similar optimizations if they can establish that reads are thread-local or lock-protected. LiteRace's [14] sampling-based approach — which builds the happens-before graph for only a small fraction of memory operations, concentrating on lightly tested parts of the code — is completely orthogonal to the happens-before graph construction itself, and hence should be straightforward to merge with our method. Only with IFRit [5] is it unclear how to combine our approach with theirs. They are essentially constructing the happens-before graph over **"interference free regions" (IFRs)** instead of individual memory operations, but IFRs are precise-but-unsound[7] abstractions of potentially data-racing memory operations. It may be possible to adapt the notion of an IFR to incorporate our visible abstract caches and additional abstractions, but the combination of our sound-but-imprecise abstractions with their unsound-but-precise abstraction eliminates any guarantees about the resulting analysis. Nevertheless, it might be empirically effective. Perhaps most promising would be adding our abstractions to the widely deployed Thread Sanitizer system [20]. Thread Sanitizer has a "pure happens-before mode", which could be easily augmented with our abstract cache models. Their

[7]Note that the paper reverses the usual terminology and describes their method as "sound" but "imprecise" and/or "incomplete". IFRs are precise, because 100% of the data races detected using IFRs are true data races, but the technique is unsound because data races can be missed. The paper describes IFRs as "sound" in the sense that overlapping IFRs provably guarantees the existence of a data race.

"segments" are a sound way to group memory accesses together into a single node in the happens-before graph, improving scalability. And their hybrid lockset-based optimizations are orthogonal to their happens-before reasoning, so there should be no conflict.

As raised in footnotes 3 and 5, our simple, proof-of-concept dynamic race detector assumes sequential consistency, so another obvious improvement would be to extend our implementation for "full-stack" soundness (*a la* "full-stack memory consistency models" [24]), through the high-level language memory model, any compiler optimizations, and any relaxed ISA memory model. As noted earlier, this could be done by instrumenting at lower levels, e.g., the memory interface of the CPU (perhaps implemented using a system simulator). An alternative would be to formalize the composition of memory models (e.g., [13, 24]) and use the resulting relaxed, composite model instead of program order as the basis of the happens-before relation. Combinations are also possible, e.g., SC-preserving compilation [15] or binary instrumentation, combined with a formalization of the relaxed ISA memory model (e.g., [4]).

From a more conceptual perspective, two opposing directions seem promising. One direction would be towards fully formal verification, by embedding our analysis into a bounded model checker or static analyzer. Challenges here would be disambiguating memory references and finding a way to join the graphs when program paths join, without being too lossy. The other direction would be to try to simplify the approach to be fast enough to be used as a runtime checker. This would require a much smaller and simpler approximation of the happens-before relation and a much faster check for races, as well as exploiting known techniques for pre-analyzing the code to be checked and possible hardware support. Another interesting direction would be automatic synthesis/optimization of synchronization code. With a data race checker, one could exhaustively explore inserting/deleting synchronization operators. SAT/SMT-style heuristics might make such an approach practical.

ACKNOWLEDGMENTS

The authors would like to thank Sam Bayless for insightful comments early in this work.

REFERENCES

[1] Martin Abadi, Cormac Flanagan, and Stephen N. Freund. 2006. Types for safe locking: Static race detection for Java. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 28, 2 (2006), 207–255. https://doi.org/10.1145/1119479.1119480

[2] Sarita V. Adve and Kourosh Gharachorloo. 1996. Shared memory consistency models: A tutorial. *Computer* 29, 12 (1996), 66–76. https://doi.org/10.1109/2.546611

[3] Sarita V. Adve, Mark D. Hill, Barton P. Miller, and Robert H. B. Netzer. 1991. Detecting data races in weak memory systems. In *18th ACM/IEEE International Symposium on Computer Architecture (ISCA)*. 234–243. https://doi.org/10.1145/115953.115976

[4] Jade Alglave, Anthony C. J. Fox, Samin Ishtiaq, Magnus O. Myreen, Susmit Sarkar, Peter Sewell, and Francesco Zappa Nardelli. 2009. The semantics of power and ARM multiprocessor machine code. In *POPL 2009 Workshop on Declarative Aspects of Multicore Programming*. 13–24. https://doi.org/10.1145/1481839.1481842

[5] Laura Effinger-Dean, Brandon Lucia, Luis Ceze, Dan Grossman, and Hans-J. Boehm. 2012. IFRit: Interference-free regions for dynamic data-race detection. In *ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'12)*. 467–484. https://doi.org/10.1145/2384616.2384650

[6] Dawson Engler and Ken Ashcraft. 2003. RacerX: Effective, static detection of race conditions and deadlocks. *ACM SIGOPS Operating Systems Review* 37, 5 (2003), 237–252. https://doi.org/10.1145/1165389.945468

[7] Cormac Flanagan and Stephen N. Freund. 2000. Type-based race detection for Java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'00). ACM SIGPLAN Notices* 35, 5, 219–232. https://doi.org/10.1145/349299.349328

[8] Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: Efficient and precise dynamic race detection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'09). ACM SIGPLAN Notices* 44, 6, 121–133. https://doi.org/10.1145/1543135.1542490

[9] Davide Giri, Paolo Mantovani, and Luca P. Carloni. 2018. Accelerators and coherence: An SoC perspective. *IEEE Micro* 38, 6 (2018), 36–45. https://doi.org/10.1109/MM.2018.2877288

[10] John L. Hennessy and David A. Patterson. 2019. A new golden age for computer architecture. *Commun. ACM* 62, 2 (2019), 48–60. https://doi.org/10.1145/3282307

[11] Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (1978), 558–565. https://doi.org/10.1145/359545.359563

[12] Guangpu Li, Shan Lu, Madanlal Musuvathi, Suman Nath, and Rohan Padhye. 2019. Efficient scalable thread-safety-violation detection: Finding thousands of concurrency bugs during testing. In *27th ACM Symposium on Operating System Principles (SOSP'19)*. ACM, 162–180. https://doi.org/10.1145/3341301.3359638

[13] Yatin A. Manerkar, Daniel Lustig, and Margaret Martonosi. 2020. RealityCheck: Bringing Modularity, Hierarchy, and Abstraction to Automated Microarchitectural Memory Consistency Verification. arXiv:2003.04892 [cs.DC]

[14] Daniel Marino, Madanlal Musuvathi, and Satish Narayanasamy. 2009. LiteRace: Effective sampling for lightweight data-race detection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'09)*. ACM, 134–143. https://doi.org/10.1145/1542476.1542491

[15] Daniel Marino, Abhayendra Singh, Todd Millstein, Madanlal Musuvathi, and Satish Narayanasamy. 2011. A case for an SC-preserving compiler. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'11)*. 199–210. https://doi.org/10.1145/1993498.1993522

[16] Robert O'Callahan and Jong-Deok Choi. 2003. Hybrid dynamic data race detection. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'03)*. 167–178. https://doi.org/10.1145/781498.781528

[17] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. 2011. LOCKSMITH: Practical static race detection for C. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 33, 1 (2011), 3. https://doi.org/10.1145/1889997.1890000

[18] Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. 2012. Scalable and precise dynamic datarace detection for structured parallelism. In *33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'12)*. 531–542. https://doi.org/10.1145/2345156.2254127

[19] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)* 15, 4 (1997), 391–411. https://doi.org/10.1145/265924.265927

[20] Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: Data race detection in practice. In *Workshop on Binary Instrumentation and Applications (WBIA'09)*. ACM, 62–71. https://doi.org/10.1145/1791194.1791203

[21] Aaron Severance and Guy G. F. Lemieux. 2013. Embedded supercomputing in FPGAs with the VectorBlox MXP matrix processor. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'13)*. IEEE/ACM/IFIP, 1–10. https://doi.org/10.1109/CODES-ISSS.2013.6658993

[22] Inderpreet Singh, Arrvindh Shriraman, Wilson W. L. Fung, Mike O'Connor, and Tor M. Aamodt. 2013. Cache coherence for GPU architectures. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 578–590. https://doi.org/10.1109/HPCA.2013.6522351

[23] Nicholas Sterling. 1993. WARLOCK–A static data race analysis tool. In *USENIX Winter Technical Conference*. USENIX Association, 97–106.

[24] Caroline Trippel, Yatin A. Manerkar, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. 2017. TriCheck: Memory model verification at the trisection of software, hardware, and ISA. In *22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*. 119–133. https://doi.org/10.1145/3037697.3037719

[25] Ana Lucia Varbanescu and Jie Shen. 2016. Heterogeneous computing with accelerators: An overview with examples. In *2016 Forum on Specification and Design Languages (FDL)*. IEEE, 1–8. https://doi.org/10.1109/FDL.2016.7880387

[26] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. 2007. RELAY: Static race detection on millions of lines of code. In *6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC-FSE'07)*. ACM, 205–214. https://doi.org/10.1145/1287624.1287654

[27] Mai Zheng, Vignesh T. Ravi, Feng Qin, and Gagan Agrawal. 2011. GRace: A low-overhead mechanism for detecting data races in GPU programs. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'11)*. 135–146. https://doi.org/10.1145/2038037.1941574

Received 2 December 2021; revised 25 March 2022; accepted 2 May 2022