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

May Young
Department of Computer Science
University of British Columbia
Canada
youngmay@cs.ubc.ca

Alan J. Hu
Department of Computer Science
University of British Columbia
Canada
ajh@cs.ubc.ca

Guy G. F. Lemieux
Department of Electrical and Computer Engineering
University of British Columbia
Canada
lemieux@ece.ubc.ca

Abstract

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 2 of the 11 examples.

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

LCTES '21, June 22, 2021, Virtual, Canada

ACM ISBN 978-1-4503-8472-8/21/06.
https://doi.org/10.1145/3461648.3463856

Keywords: Data Race, Hardware Accelerator, Memory Coherence, Caching

ACM Reference Format:
May Young, Alan J. Hu, and Guy G. F. Lemieux. 2021. Cache Abstraction for Data Race Detection in Heterogeneous Systems with Non-coherent Accelerators. In *Proceedings of the 22nd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '21), June 22, 2021, Virtual, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3461648.3463856

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, 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 Sec. 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, both CUDA and OpenCL 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. Non-coherent 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 Fig. 1, for a simple, heterogeneous system consisting of a single scalar CPU and a single vector accelerator (Fig. 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 synchronization happens because the `vector_` variables are specified as uncached. 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 Sec. 4.4).

3 Theoretical Framework

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

[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 );
```

Figure 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 3 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 3 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 3 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 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.

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

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

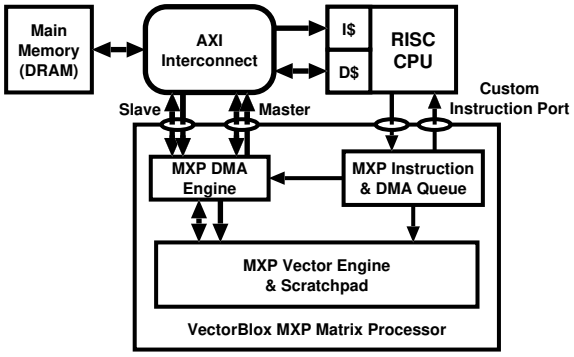


Figure 2. Example Heterogenous System with Accelerator. The code in Fig. 1 was written for an embedded system with a VectorBlox MXP [21] accelerator (with non-coherent scratchpad) and scalar CPU with caches (configurable with VectorBlox ORCA (RISC-V), Altera Nios II, or Xilinx MicroBlaze soft cores, 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.

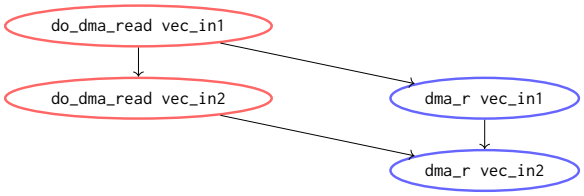


Figure 3. Example Graph of Happens-Before Relation. The `do_dma_read` nodes are CPU operations requesting that the accelerator perform a DMA read, and the `dma_r` nodes are when the accelerator actually does the read.

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 Sec. 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. Fig. 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.

3.2 Idioms for Shared Memory Access

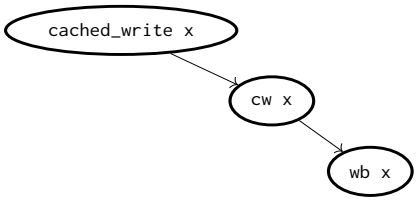
Data race detection reduces to building the happens-before graph (or an efficient abstraction) and checking for two unordered nodes that access the same memory location, of which at least one is a write. In the classical literature on data race detection, the nodes correspond to memory reads and writes, which are assumed to happen atomically. In a heterogeneous system, with different types of memory accesses, we must model memory operations more precisely.

Fortunately, we've found that main 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 generating 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., Fig. 3). Program order edges connect consecutive operations performed by a single thread or by an in-order accelerator. Synchronization/barrier instructions relate 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.)

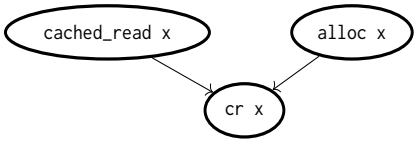
Caches are modeled similarly to accelerators: just as the `do_dma_write` CPU operation generates a causal edge to the corresponding `dma_w` operation on the accelerator, the `cached_write` CPU operation generates a causal edge to the corresponding `cw` node in the cache, which denotes the cache accepting the written data. The cache, in turn, would have a causal edge to a `wb` writeback node, because the data will eventually be written back to memory, and it is this `wb` operation that accesses main memory and must be checked for data races (Fig. 4a). 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 (Fig. 4b). 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,

Table 1. Operations Tracked for the VectorBlox Happens-Before Graph. All operations (except sync) are parameterized with the addresses affected.

Type of Node	Meaning
cached_read	CPU tries to read data from cache.
cr	Cache returns data to CPU.
alloc	Cache allocates cache line and reads data from main memory.
cached_write	CPU tries to write data to cache.
cw	Cache accepts data from CPU.
wb	Cache writes back data to main memory.
cache_flush	CPU tells cache to remove data, performing wb if dirty.
uncached_read	CPU reads data from main memory, bypassing cache.
uncached_write	CPU writes data to main memory, bypassing cache.
do_dma_read	CPU asks accelerator to read from main memory.
dma_r	Accelerator reads data from main memory.
do_dma_write	CPU asks accelerator to write to main memory.
dma_w	Accelerator writes data to main memory.
sync	CPU waits for outstanding DMA operations to complete.



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



(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.

Figure 4. Basic Modeling of Cached Writes and Reads in Happens-Before Graph.

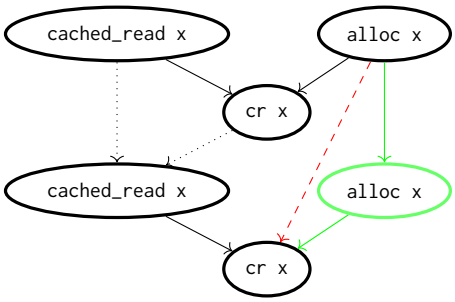


Figure 5. Did the second `alloc` happen? If so, we would have the solid green edges in the graph. If not, we would have the dashed red edge in the graph, instead. Theorem 3.1 says that it is safe to consider only the first case. The dotted lines are shorthand that there may be other instructions in between.

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.

3.3 Safely Abstracting Cached Reads and Writes

As noted earlier, the real problem for data race analysis with mixed coherent and non-coherent memory accesses is the caches and the unpredictable memory traffic they can generate. To create a broadly applicable data race analysis, we must 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 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.

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 `alloc` didn't happen, because the cache line was already in the cache and might not have been evicted (e.g., Fig. 5).

There are 4 cases to consider. The first case, a cached read followed later by a cached write to the same address, requires no special handling.

The second case is a cached read followed by another cached read (to the same address) (Fig. 5). Because we don't

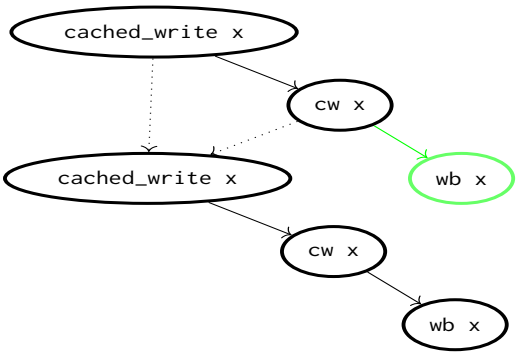


Figure 6. Did the first writeback happen? If so, the the first wb node would be in the graph (the solid green edge and wb node); if not, it wouldn't. Theorem 3.2 says that it is safe to consider only the first case. The dotted lines are a reminder that there may be other instructions in between.

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 number of graphs to analyze. Fortunately, we establish the following theorem:

Theorem 3.1. *It is safe (i.e., data-race preserving) to build only the graph with both* alloc *nodes.*

Proof. 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. □

A cached write followed by a cached write produces an analogous situation (Fig. 6). Did the first write's wb happen?

Theorem 3.2. *It is safe to build only the graph with both* wb *nodes.*

Proof. Similar to the proof 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. □

The most interesting case is a cached write followed by a cached read (Fig. 7). Did the writeback happen before the cached read occurs. If so, then the cached read needs an alloc node; if not, then the cache line is still dirty, and the cr node happens before the wb node. 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.

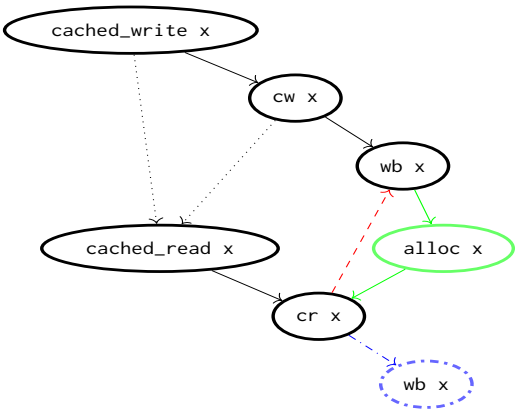


Figure 7. Did the cache line get written back? If so, we would have the solid green edges in the graph. If not, we would have the dashed red edges in the graph. The dashdotted blue edges are part of a safe abstraction. The dotted lines are a reminder that there may be other instructions in between.

Theorem 3.3. *It is safe to build the graph which assumes the writeback happened, but also add another copy of the* wb *node with an edge from the* cr *node.*

Proof. 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. □

Together, these theorems allow us to abstract away cache details like associativity, eviction and replacement policies, and pre-fetching, yet still be guaranteed to detect any possible races.

3.4 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

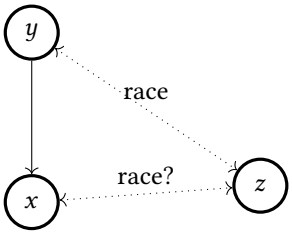


Figure 8. Illustration of Lemma 3.4. Node x is a write node newly added to the active frontier, and node y is a memory operation to the same address that was already in the active frontier. If we delete y from the active frontier, might we miss a race between y and a node z added later? The answer is that we will still detect a race (between x and z instead of between y and z): $z \not\mapsto x$ because x was added before z, and $x \not\mapsto z$ or else $y \rightarrow x \rightarrow z$, which contradicts that y and z are in a race.

natural and the same as in classical data race analysis. The italicized part means that an `alloc` node is created only when the corresponding `cached_read` node is created. This late creation is necessary because when analyzing the prefix of a trace, in principle, any address might be pre-fetched into the cache. However, if we were to pre-emptively create these `alloc` nodes, we would flag many spurious data races that have no actual relevance (because the racy value loaded into the cache would be overwritten before being used).

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. To help keep track of program order, it's also convenient to mark the latest operation in each program-order thread. When we add a new node to the graph, we update the set of marked nodes. Then, any node that is not either marked or reachable from a marked node via happens-before edges is deleted.

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. (Fig. 8 illustrates the intuition behind this proof.) First, if neither $x \rightarrow y$ nor $y \rightarrow x$, 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. Now, is it possible that $x \rightarrow y$? The answer is no, by the as-soon-as-possible property: the edges from $x \rightarrow y$ 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 analysis, we encounter a memory operation z that's in a race with node y that we unmarked in this step. By the same argument that $x \not\mapsto y$, we know that $z \not\mapsto x$. On the other hand, if $x \rightarrow z$, then we have $y \rightarrow x \rightarrow z$, which contradicts the fact that y and z are in a race. Hence, $x \not\mapsto z$ and $z \not\mapsto x$, which means x and z are in a race. Therefore, we will still detect a race, even if we delete 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. Suppose that later in the analysis, we encounter a memory operation z (which must be a write) that's in a race with a node y that we unmarked in this step. By the same arguments as above, we know that $z \not\mapsto x$ (because x was already in the graph) and $x \not\mapsto z$ (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. 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. □

Theorem 3.7. *If there's a data race in the full happens-before graph, then the online algorithm will find a data race in the active frontier.*

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, keeping the nodes that happen after the marked nodes avoids spurious reports of data races:

Theorem 3.8. *If the online algorithm flags a data race, then the same race exists in the full happens-before graph.*

Proof. (Fig. 9 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, $z \not\mapsto y$ and $y \not\mapsto z$ in the active frontier, and we need to prove that $z \not\mapsto y$ and $y \not\mapsto z$ in the full happens-before graph. By the same as-soon-as-possible argument we've used already, we know that $z \not\mapsto y$ in the full graph, because if $z \rightarrow y$, then y couldn't have been created before

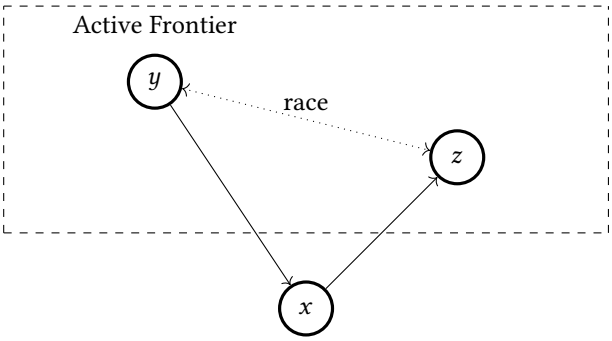


Figure 9. Illustration of Theorem 3.8. Node z is a newly added memory operation, and the analysis detects a race between z and existing node y in the active frontier. Could it be that y and z are *not* in a race if we considered the full happens-before graph? The answer is no. It's easy to see that $z \not\to y$ 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.

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. □

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 that could have happened (even if it didn't) 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 Fig. 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.[4] 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.[5]

[4] Instrumenting at the source-code level does mean that our proof-of-concept implementation assumes sequential consistency. 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]).

[5] 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.

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. 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, to produce a greater variety of buggy examples, we introduced bugs by removing necessary synchronization (e.g., sync statements).

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 9 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. Fig. 10 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 bugs were unremarkable. The tool was able to detect the injected bugs easily in all cases. 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.

In all experiments, 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. (As for false negatives, our analysis is sound.)

Remarkably, the tool found subtle, previously unknown races in two out of the eleven examples (in their original form, without injected bugs). One example is actually the vector addition example `vbw_vec_add` in Fig. 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`, a cached vector that eventually has its values written back when the test zeroes this array, and `vector_in1`, an uncached vector in main memory that is accessed by the accelerator via DMA. However, although they are two different variables, when executed in the VectorBlox simulator, 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 writebacks 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.)

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

Table 2. Test Programs. 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. The asterisks (*) indicate examples where our analysis discovered previously unknown data races.

Test Name	Test Description
vbw_libfixmath	Square root and division
vbw_mtx_fir	2D FIR filter using matrices
vbw_mtx_median_argb32	Median filter with 32-bit data type
vbw_mtx_median	Median filter with 8-bit data type
vbw_mtx_motest *	Motion estimation
vbw_mtx_sobel	Sobel filter
vbw_mtx_xp	Matrix transpose
vbw_vec_add *	Vector addition
vbw_vec_fft	FFT
vbw_vec_fir	FIR filter using vectors
vbw_vec_power	Vector exponentiation

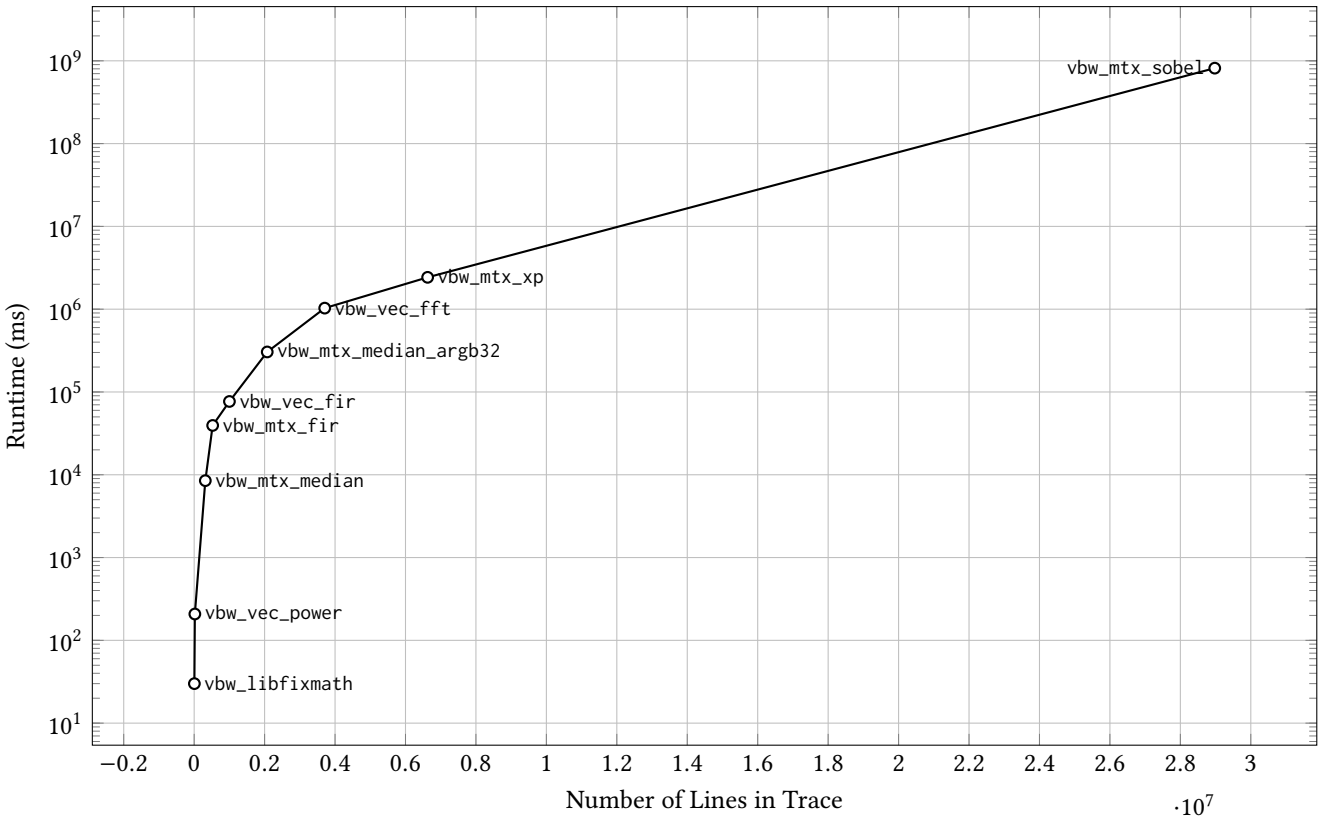


Figure 10. Analysis Runtime of Examples Without Data Races. (All examples with data races terminated quickly.) As noted in the text, our implementation is a straightforward tracking of the happens-before graph, so we make no claims of stellar performance. Nevertheless, this semi-log plot shows runtime 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.

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. This should be possible for any race detector that reasons about the happens-before graph. The result would be greatly improved performance and scalability over our implementation, and the capability to detect an emerging class of data races for the state-of-the-art race detector.

As raised in footnotes 3 and 4, 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. This work was supported by Discovery Grants from the Natural Sciences and Engineering Research Council of Canada (NSERC).

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 Lightweigth 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