Dynamic Race Detection for Non-Coherent Accelerators

by

May Young

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL
STUDIES
(Computer Science)

The University of British Columbia
(Vancouver)

May 2020

The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the thesis entitled:

Dynamic Race Detection for Non-Coherent Accelerators

submitted by **May Young** in partial fulfillment of the requirements for the degree of **Master of Science** in **Computer Science**.

Examining Committee:

Alan Hu, Computer Science
Supervisor

Guy Lemieux, Electrical and Computer Engineering
Supervisory Committee Member

Abstract

Modern System-on-Chip (SoC) designs are increasingly complex and heterogeneous, featuring specialized hardware accelerators and processor cores that access shared memory. Non-coherent accelerators are one common memory coherence model. The advantage of non-coherence in accelerators is that it cut costs on extra hardware that would have been used for memory coherence. However, the disadvantage is that the programmer must know where and when to synchronize between the accelerator and the processor. Getting this synchronization correct can be difficult.

We propose a novel approach to find data races in software for non-coherent accelerators in heterogeneous systems. The intuition underlying our approach is that a sufficiently precise abstraction of the hardware's behaviour is straightforward to derive, based on common idioms for memory access.

To demonstrate this, we derived simple rules and axioms to model the interaction between a processor and a massively parallel accelerator provided by a commercial FPGA-based accelerator vendor. We implemented these rules in a prototype tool for dynamic race detection, and performed experiments on eleven examples provided by the vendor. The tool is able to detect previously unknown races in two of the eleven examples, and additional races if we introduce bugs in the synchronization in the code.

Lay Summary

Modern systems-on-chip (SoCs) are integrated circuits that combine various computer components onto a single chip. Their designs are increasingly complex and heterogenous, where specialized hardware called accelerators and processors access shared memory. In non-coherent and heterogeneous systems, the programmer must know where and when to synchronize between the accelerator and the processor. Getting this synchronization correct is extremely difficult. Doing it incorrectly can result in dangerous bugs known as data races.

We propose a novel approach to analyze code for data races in non-coherent accelerators. We derived simple rules to model the interaction between a processor and an accelerator provided by a commercial vendor. We implemented these rules in a prototype tool, and performed experiments on eleven examples provided by the vendor. The tool detected previously unknown races in two of the eleven examples, and additional races if we introduce synchronization bugs in the code.

Preface

The work presented in this thesis was performed in collaboration with my supervisor, Dr. Alan Hu, and also with Dr. Guy Lemieux, who is the CEO of VectorBlox and a Professor in the Department of Electrical and Computer Engineering. None of the text of the dissertation is taken directly from previously published or collaborative articles.

The theoretical framework in Chapter 3 was designed by Alan Hu and me, and was influenced by input and feedback from Guy Lemieux and Sam Bayless. The proofs of the theorems in Section 3.4 were done by Alan Hu and me. The dynamic race detection tool in Chapter 4 and the experiments in Chapter 5 were primarily implemented and performed by me. The vector power experiment in Chapter 5 was completed by Guy Lemieux and me.

Table of Contents

Abstract . . . . . **iii**

Lay Summary . . . . . **iv**

Preface . . . . . **v**

Table of Contents . . . . . **vi**

List of Tables . . . . . **viii**

List of Figures . . . . . **ix**

Glossary . . . . . **x**

Acknowledgments . . . . . **xi**

1 Introduction **1**

2 Background **4**

2.1 Data Races 4

2.2 Non-Coherent Accelerators 5

3 Theory **8**

3.1 Simple Example 9

3.2 Examples of Basic Rules 15

3.3 Full Ruleset for a Non-Coherent Accelerator 18

3.4 Theorems and Proofs 24

4 Implementation 32
4.1 VectorBlox Architecture 33
4.2 Instrumentation and Execution 34
4.3 Trace Analysis 36

5 Evaluation 37
5.1 Experimental Setup 37
5.2 Time Required for Analysis 37
5.3 Experimental Results 39
5.4 Injection of Bugs in Examples 43

6 Related Work 46

7 Conclusions 48

Bibliography 50

List of Tables

Table 3.1 Node semantics 12

Table 5.1 Benchmark examples 38

List of Figures

Figure 2.1 Common data race example 5
Figure 2.2 VectorBlox architecture 6

Figure 3.1 A simple program 10
Figure 3.2 Graph of memory operations 11
Figure 3.3 Steps to build a graph 15
Figure 3.4 Arrow from wb node to alloc node 24
Figure 3.5 Theorem 3.4.3 26
Figure 3.6 Propagate writebacks 30

Figure 4.1 Workflow 33
Figure 4.2 Instrumented simple program 35
Figure 4.3 Trace file 35

Figure 5.1 Analysis runtime including the largest example 39
Figure 5.2 Vector addition graph showing a race 42
Figure 5.3 Sync in vbw_vec_power_t example 45
Figure 5.4 Missing sync in vbw_vec_power_t example 45

Glossary

API Application Programming Interface

CPU Central Processing Unit

DMA Direct Memory Access

FFT Fast Fourier Transform

FIR Finite Impulse Response

FIFO First-In, First-Out

SOC System-on-Chip

Acknowledgments

I would like to thank my supervisor, Alan Hu, for his guidance and advice. I would also like to thank Guy Lemieux for introducing the problem, giving me the opportunity to present my thesis in the industry, and being the second reader. Finally, I would like to thank my family, friends, and lab-mates who supported me throughout my Master's.

Chapter 1

Introduction

Modern System-on-Chip (SoC) designs are increasingly complex and heterogeneous, featuring specialized hardware accelerators and processor cores. This is largely due to the demise of Dennard scaling and today's growing demands of performance and energy efficiency, which are leading towards specialization and accelerator-rich architectures.

Heterogeneous computing architectures generally consist of at least two components: a processor and an accelerator, which improves overall performance [17]. In order to obtain the performance boost, the processor offloads some computation for an application to the accelerator. During the computation, the processor does not need to wait for the results; rather, it can continue executing other instructions in parallel, or even contribute to the processing as well. Taking advantage of *both* the processor and the accelerator simultaneously exploits the heterogeneity of the platform, resulting in performance improvement and efficient use of the platform in question.

Accelerators and processors commonly communicate with each other through shared memory. This obviously introduces the problem of memory coherence, since processors or accelerators may access stale data. In particular, non-coherent accelerators read from and write to main memory via Direct Memory Access (DMA), bypassing the processor's cache(s). Thus, the processor must wait for all DMA transfers to complete before accessing the data in main memory. Non-coherent accelerators have the advantages of using less complex designs and fewer

resources to build, compared to both hardware and software cache coherence protocols.

However, using non-coherent accelerators puts more burden on the programmer to ensure that the program they are writing is race-free. For example, consider an architecture consisting of one general-purpose Central Processing Unit (CPU) with its own cache and an accelerator that cannot communicate with the CPU or the CPU cache. The CPU, the CPU cache, and the accelerator are each connected to main memory. Then, the conservative programmer would flush the entire cache before each DMA read and before each DMA write in a program. In contrast, the aggressive programmer would flush only the dirty data from the cache before each DMA read, and flush any valid data before each DMA write. Therefore, the aggressive programmer must know what is in the cache before they can safely execute a DMA operation (read/write). Of course, there may be a type of programmer that falls in between these two sides of the spectrum.

Two problems can arise from programming in a non-coherent, heterogeneous environment: (1) there can be unnecessary synchronization operations that the programmer included because they are not confident of what is in the cache at a certain time; or (2) there can be missing synchronization operations because they failed to properly analyze the code. This results in either inefficiency or potentially incorrect behaviour.

In this thesis, we propose a novel approach to analyze code in software for non-coherent accelerators in heterogeneous systems. We design and implement a prototype tool that analyzes a trace of the memory operations in a single execution of a program and detects races — if any — in it. We use a semi-automatic process to instrument the program to print out each memory operation into a trace text file when it is compiled and executed. The benefits of our approach over existing memory coherence schemes are that it is generalizable to any non-coherent architecture, does not require additional hardware or new instructions, and is sound with respect to an execution. A limitation of our approach is that it verifies only that single execution.

Overall, the major contributions of this thesis include:

- A novel approach to find data races in software for non-coherent accelerators

in heterogeneous systems.

- Proofs of theorems that show the correctness and soundness (to an extent) of our solution.
- A prototype of a dynamic race detection tool, which analyzed eleven open-source examples provided by a commercial FPGA-based accelerator vendor.

The rest of the thesis is organized as follows: Chapter 2 provides background knowledge needed to understand our solution; Chapter 3 describes our theoretical framework; Chapter 4 presents and explains the implementation of the dynamic race detection tool; Chapter 5 reports the examples and results used in the experiments; Chapter 6 reviews related work; and Chapter 7 concludes.

Chapter 2

Background

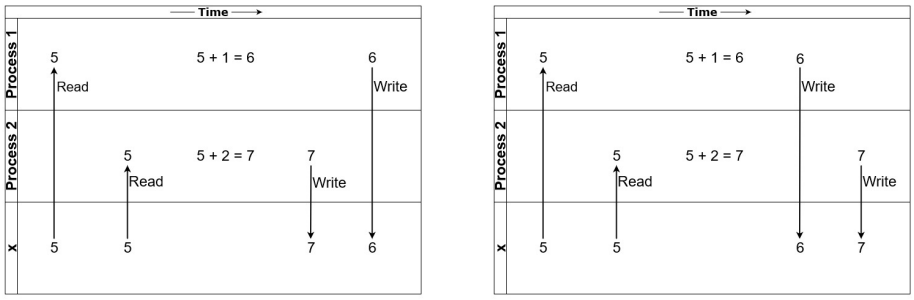
2.1 Data Races

With more than one thread or hardware component being able to directly access a shared resource such as main memory, the problem of managing concurrent accesses arises. Thus, errors in synchronization can cause data races, which negatively impacts correctness. We formally define a data race as below.

Definition 2.1.1 (Data Race)**.** A **data race** occurs when there are two or more possibly unordered operations that access the same memory location, where at least one of these operations is a write, and there is no *happened-before* relation (explained further below) that forces an ordering between these operations.

One way to detect data races is to utilize Lamport's *happened-before* relation [9], denoted by $\rightarrow$. This relation is a strict partial ordering of events in a system such that: (1) If a and b are events in the same process, and a comes before b, then $a \rightarrow b$; (2) If a is the sending of a message and b is the receipt of the same message, then $a \rightarrow b$; and (3) If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$.

To detect data races, memory operations can be treated as events. For example, a program first performs a cached write a, followed by an uncached read b. The cached write happens before the uncached read due to program order. If we were to put these operations into a simple graph, there would be a node representing the cached write and another node representing the uncached read. An arrow would



(a) Process 1 writes the value of x after Process 2, so the final value of x is 6.

(b) Process 2 writes the value of x after Process 1, so the final value of x is 7.

Figure 2.1: This shows a data race between two different processes in regards to who writes the value of the shared variable x last. Both (a) and (b) assume that Process 1 always reads x before Process 2.

be drawn from the cached write node to the uncached read node, where the arrow symbolizes that the write *a* happens before the read *b*.

There is a data race between two nodes in a graph if they access the same memory location, at least one is a write, and there is no happened-before relation that forces an ordering. A common example of a data race involves two different processes where each reads a shared variable x, increments it, and writes the new value back to x. Consider the example in Figure 2.1 where Process 1 adds 1 to x and Process 2 adds 2. If x was initially 5, without any synchronization in place, the final value of x could be 6 or 7 (Figure 2.1). Sometimes, it could be 6 (Figure 2.1a). Other times, it could be 7 (Figure 2.1b). Thus, it is a race between Process 1 and Process 2 to change the value of x; this is a data race. In this example, assume that the correct result is 6, which means that, when these two processes are executed, at times, x contains the correct value; other times, it contains the wrong value.

2.2 Non-Coherent Accelerators

The main idea behind non-coherent accelerators is that they access shared memory through DMA, without having to go through the processor's cache(s). This frees up the processor to continue executing in parallel with the accelerator. It also

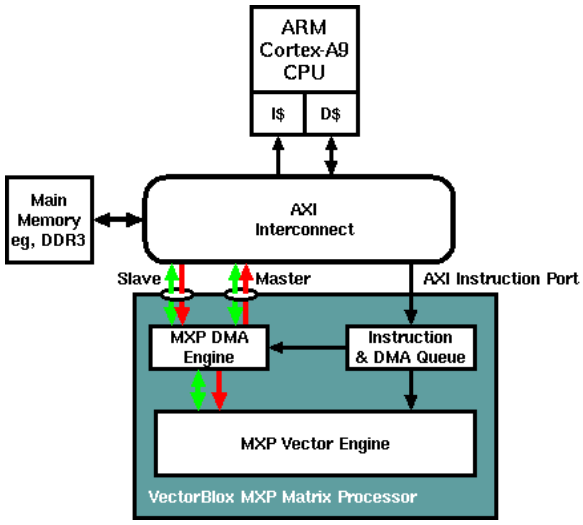


Figure 2.2: Example of an accelerator architecture taken from VectorBlox [1]. This is a VectorBlox MXP [15] system with ARM Cortex-A9 in Zynq-7000 FPGAs.

gives the accelerator much faster access to memory. A typical architecture is one that is used at VectorBlox, a commercial FPGA-based accelerator vendor that we collaborated with and whose examples we used. Figure 2.2 shows an ARM-based architecture from VectorBlox that contains one processor and one accelerator on a single FPGA board. The CPU accesses shared main memory through its caches. On the other hand, the vector engine accesses main memory via DMA and only when it receives such instructions from the CPU. In other words, the CPU offloads some computation to the accelerator by placing the instructions associated with this computation in a DMA First-In, First-Out (FIFO) queue, as seen in Figure 2.2, from which the accelerator (i.e., the vector engine) grabs them to do its work.

According to [11], "a memory is *coherent* if the value returned by a read operation is always the same as the value written by the most recent write operation to the same address". An accelerator architecture has more than one access path to shared memory. Figure 2.2 exemplifies this as two different entities — a processor and an accelerator — each have their own memory access path.

Giri et al. studied three accelerator cache-coherence models from an SoC perspective: non-coherent, coherent with the last-level cache (LLC-coherent), and

fully-coherent [8]. Non-coherent accelerators access shared memory via DMA, bypassing the cache hierarchy. Therefore, the cache hierarchy must flush the accelerator data region before accelerator execution. Fully-coherent accelerators are coherent with the private caches of the processors, commonly through the MESI or MOESI protocol, and do not require any cache flushing prior to accelerator execution. LLC-coherent accelerators are an intermediate model, as they are coherent with the LLC (via DMA requests) but not coherent with the private caches of the processors (hence, cache flushing is needed before accelerator execution). One finding by Giri et al. concluded that non-coherent accelerators are more effective for large workloads, which takes advantage of the higher throughput that is achieved from larger DMA bursts.

In addition, the non-coherent model has less hardware overhead than one with some type of memory coherence protocol in place. No extra hardware required also means a more cost-efficient model. However, a programmer must know when and where to synchronize between the accelerator and the processor. Getting this synchronization correct can be difficult.

Chapter 3

Theory

In order to detect races (which are defined in Definition 2.1.1), we construct a graph of all memory operations in a program, where a node in the graph represents a memory operation (or other relevant operation). An arrow from a node x to another node y indicates a happens-before relation; that is, $x \rightarrow y$. The graph is used to track enforced ordering of memory operations. The overall task of our analysis is simply to check the graph for two unordered nodes that access the same memory location, of which at least one is a write.

Our theoretical framework exists to make this task scalable as well as provably sound, i.e., it cannot miss a possible race in a program execution. Other important objectives are that our analysis not raise too many false alarms, and that it not require excessively detailed hardware modeling.

A necessary condition for scalability is that our analysis be done "on-the-fly", i.e., creating the graph node-by-node according to the program execution and detecting races as soon as possible, instead of trying to post-process an enormous execution trace after-the-fact. However, a challenge arises because memory operations do not have an obvious time at which they might occur. For example, a cache line might fill from main memory at some unspecified time before a cached read that would fill in that same line occurs. In fact, if the cache does prefetching, the allocate may happen without any CPU load instruction at all. Similarly, writebacks from the cache might happen at a not precisely specified time after the corresponding program instruction. Therefore, two problems might occur with on-

the-fly analysis: (1) false races could be flagged if nodes are added to the graph too early, because the ordering becomes evident only after the current nodes are created; and (2) races might not be detected if the conflicting nodes are added to the graph too late, after other nodes that enforce an ordering (which retroactively close a window of time during which a race could have happened). To prevent these two situations from occurring, we enforce the following invariant:

Definition 3.0.1. A node **exists** in the graph iff a memory operation can happen *and* the value being read or written by the operation is visible to the CPU or the accelerator. (In other words, the operation corresponding to a node *could* happen as soon as the node exists in the graph, but not before that point in time. Additionally, the result of that operation must be observable by the CPU or accelerator.)

For example, since we don't want to model the details of a cache's detailed pre-fetching and allocation policies, we must conservatively assume that a cache line might pre-fetch from memory at any point in time. But if we created nodes for all of these pre-fetch operations that *might* happen, we would create countless false race detections between these mythical pre-fetches and any write operation to the same address. Instead, we wait until we encouter an operation where the CPU does a cached read, and then retroactively create a node for the cache line allocation that must have happened before the cached read can happen. We prove in this chapter that these sorts of modeling decisions are sound.

What exactly are the nodes being created? We track cached reads, cached writes, uncached reads, uncached writes, DMA reads, DMA writes, flush instructions explicitly written by a programmer in a program, and sync operations. The aforementioned cached/uncached operations relate to the cpu-cache (i.e., the CPU's data cache).

3.1 Simple Example

Perhaps it would be easier to introduce our theory with an example: Figure 3.1 shows a simple program that performs a Fast Fourier transformation (FFT) on input microphone data. The flush of the array `a`, which stores the data, from the CPU cache at line 13 ensures that the accelerator's DMA read on line 15 gets the correct values from main memory. Furthermore, the sync operation at line 19 guarantees

```
1  int8_t main(void)
2  {
3      int8_t i;
4      int8_t N = 2;
5      int8_t a[N];
6      int8_t scratchpad[N]; // accelerator's internal memory
7
8      for (i = 0; i < N; i++)
9      {
10         a[i] = read_mic();                  // cached_write
11     }
12
13     flush(a);                               // cache_flusha
14
15     vbx_dma_to_vector(a, scratchpad, N);    // do_dma_read
16     test_vector_fft(scratchpad, N);
17     vbx_dma_to_host(scratchpad, a, N);      // do_dma_write
18
19     vbx_sync();                             // sync
20
21     return a[0];                            // cached_read
22 }
```

Figure 3.1: A simple program containing several types of memory operations that we capture in a graph. The CPU reads in data from a microphone (lines 8-11), offloads the Fast Fourier transformation computation to the accelerator (lines 15-18), and finally reads the first value of the returned data (line 22). The accelerator copies the mic data into its internal (scratchpad) memory (line 15), performs work on it (lines 16-17), and then copies the transformed data back into main memory for the CPU to access (line 18).

that all DMA operations have finished, so the CPU can safely access the operated-on data, such as the read on line 21.

Assuming the same typical architecture described in Section 2.2, the CPU issues and executes the operations specified in an input program (e.g., cached_write, do_dma_read, etc.), which can then cause the cache or the accelerator to perform an appropriate operation (e.g., a writeback or a DMA read, respectively). These operations are defined in Table 3.1.

Suppose that the hexadecimals in Figure 3.2 are the memory addresses — or more precisely, the address range — of the shared resource among the CPU and the accelerator, that is, array `a`. Figure 3.2 shows a cached write (in the address range, 0xc0f45c70 to 0xc0f45c71, shortened to 0xc70 to 0xc71 onwards), as the first node in the graph since it is the first memory operation in the program in Figure 3.1.

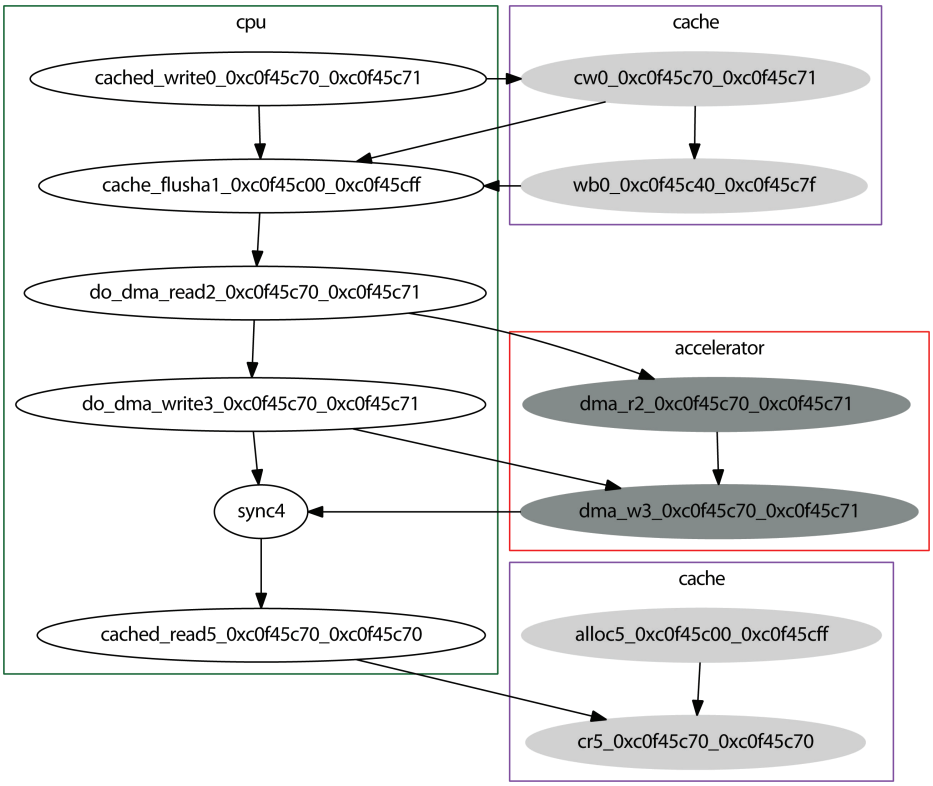


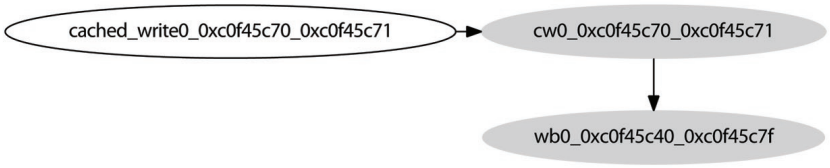
Figure 3.2: Graph of all memory operations from the program in Figure 3.1. White nodes are operations issued and executed by the CPU; light grey nodes are those executed by the CPU cache; and dark grey nodes are those by the accelerator

This happens before the next operation (a cache flush), hence, an arrow from the cached_write node to the cache_flusha node. Additionally, the CPU cache actually writes the value at addresses 0xc70 to 0xc71 into the cache, which generates a new cw node (the top right light grey node). The cache will eventually write back the value to main memory at some unspecified time but after the cache brought the value into the cache; thus, cw $\rightarrow$ writeback wb. Finally, wb $\rightarrow$ cache flush because the writeback must occur before or at the same time as the cache flush. If it did not happen before the flush, the cache flush would force the pending writeback data to be written back to main memory as part of the flush.

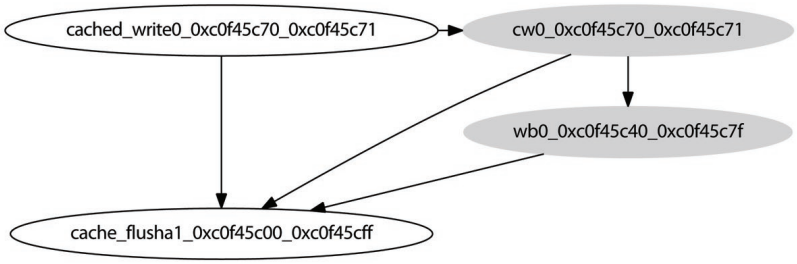
These nodes and arrows were built using rules that we have painstakingly devised, described later in this section (Section 3.2 and Section 3.3). We constructed the rest of the graph in a similar way. The steps are visualized in Figure 3.3.

Table 3.1: Node semantics

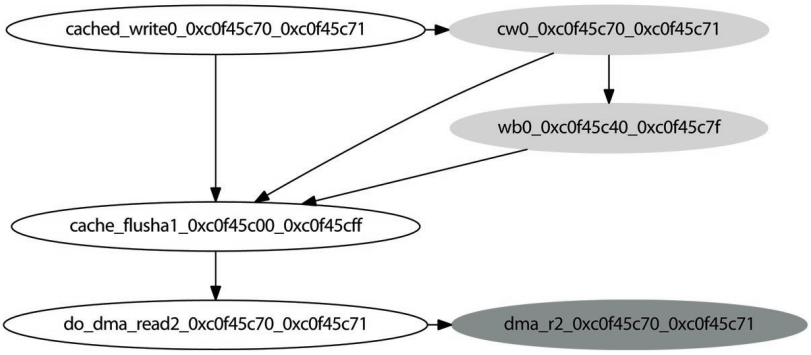
Type of Node	**Semantics**
cached_read	CPU reads data from cache
cr	Cache returns data to CPU
alloc	Cache reads data from shared memory
cached_write	CPU writes data to cache
cw	Cache writes data into cache
wb	Cache writes back data to shared memory
cache_flusha	CPU tells cache to remove data, performing wb if dirty
uncached_read	CPU reads data from shared memory, bypassing cache
uncached_write	CPU writes data to shared memory, bypassing cache
do_dma_read	CPU tells accelerator to read from shared memory
dma_r	Accelerator reads data from shared memory
do_dma_write	CPU tells accelerator to write to shared memory
dma_w	Accelerator writes data to shared memory
sync	CPU waits for all DMA operations to complete


cached_write0_0xc0f45c70_0xc0f45c71
cw0_0xc0f45c70_0xc0f45c71
wb0_0xc0f45c40_0xc0f45c7f

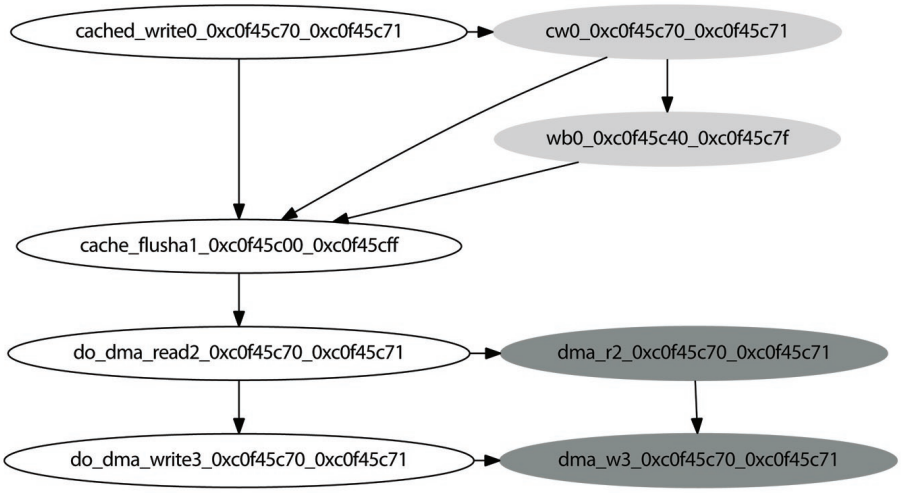
(a)


cached_write0_0xc0f45c70_0xc0f45c71
cw0_0xc0f45c70_0xc0f45c71
wb0_0xc0f45c40_0xc0f45c7f
cache_flusha1_0xc0f45c00_0xc0f45cff

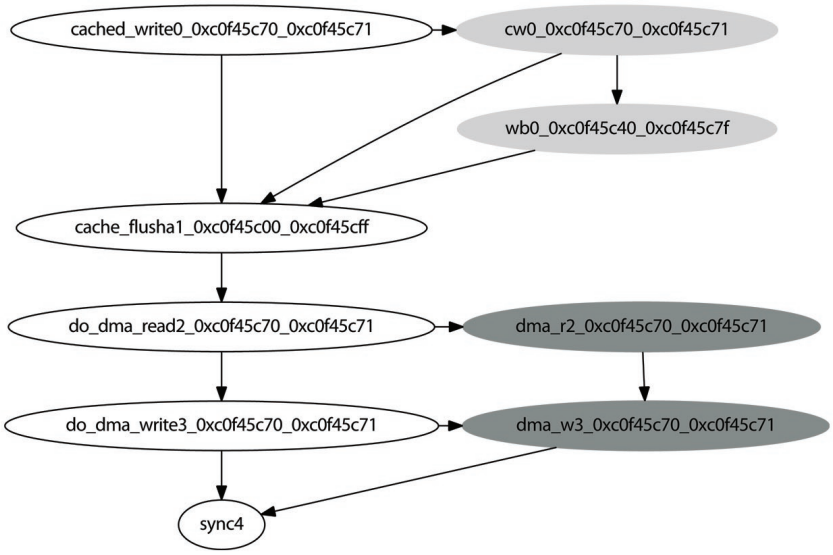
(b)


cached_write0_0xc0f45c70_0xc0f45c71
cw0_0xc0f45c70_0xc0f45c71
wb0_0xc0f45c40_0xc0f45c7f
cache_flusha1_0xc0f45c00_0xc0f45cff
do_dma_read2_0xc0f45c70_0xc0f45c71
dma_r2_0xc0f45c70_0xc0f45c71

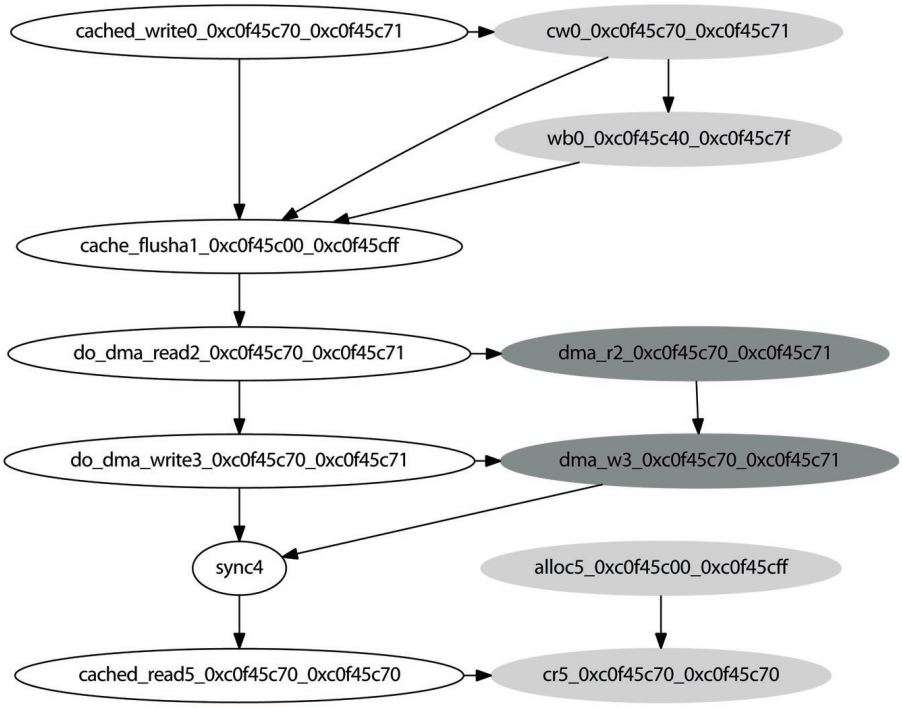
(c)


cached_write0_0xc0f45c70_0xc0f45c71
cw0_0xc0f45c70_0xc0f45c71
wb0_0xc0f45c40_0xc0f45c7f
cache_flusha1_0xc0f45c00_0xc0f45cff
do_dma_read2_0xc0f45c70_0xc0f45c71
dma_r2_0xc0f45c70_0xc0f45c71
do_dma_write3_0xc0f45c70_0xc0f45c71
dma_w3_0xc0f45c70_0xc0f45c71

(d)


cached_write0_0xc0f45c70_0xc0f45c71
cw0_0xc0f45c70_0xc0f45c71
wb0_0xc0f45c40_0xc0f45c7f
cache_flusha1_0xc0f45c00_0xc0f45cff
do_dma_read2_0xc0f45c70_0xc0f45c71
dma_r2_0xc0f45c70_0xc0f45c71
do_dma_write3_0xc0f45c70_0xc0f45c71
dma_w3_0xc0f45c70_0xc0f45c71
sync4

(e)



(f)

Figure 3.3: Steps to build a graph one node at a time for the program in Figure 3.1. Notice that new nodes are generated by an instruction, such as in (a), (c), (d), and (f).

3.2 Examples of Basic Rules

We create a node and draw a happens-before arrow in the graph according to our rules. A node is stored as a triple $(name, addrRange, neighbours)$, where $addrRange$ is an inclusive address range $[low, high]$.

As we parse a program line by line, we are only concerned with instructions associated with accessing memory and case-split depending on the type of memory operation. In each of these cases, we always construct a new node *currNode* with a given name (e.g., "cached_read") which is the type of memory operation, and its address range. If a previous node *prevNode* exists — in other words, if there is

another memory operation before *currNode* in the program — then we draw an arrow from *prevNode* to *currNode*. This is summarized in pseudocode as:

```
currNode = new Node(op, addrRange);
if (prevNode)
{
  prevNode->neighbours.append(currNode);
}
```

Firstly, handling uncached reads and writes are the simplest. After creating the current node and possibly connecting it to the previous node, the only thing to do consists of checking if there is a race from *currNode* to the last relevant writeback (wb) node or to the last DMA operation node, where "relevant" means a dangling node whose address range overlaps with that of *currNode*. Note that a dangling node is one that has no children in the graph.

```
if (op == "uncached_read" ||
    op == "uncached_write")
{
  hasRace(lastWbs, currNode);
  hasRace(lastDmaOps, currNode);
}
```

Secondly, we will explain the more complicated processing of cached reads and writes. Again, we first create the cached read/write node and draw an arrow from the previous node to it. For the cached read operation, we generate a new node dubbed "cr" that symbolizes the cached read executed by the cache itself, as well as a new predecessor node called "alloc" that represents the cache allocation done by the cache.

The CPU and the CPU cache are treated as separate entities and, therefore, two different nodes (one for each entity) are created for an operation executed by the CPU that involves the cache. For instance, as seen in Table 3.1, a cached read has a "cached_read" node that is executed by the CPU, as well as a "cr" node and an "alloc" node that are executed by the cache. This is the reason for the different coloured nodes and groups drawn in Figure 3.2, where white nodes are CPU executed and light grey nodes are cache executed.

We draw an arrow from the cache-executed "cr" or "cw" node to the next CPU operation, if it exists. This is to symbolize that control goes back to the CPU from the cache.

Cache allocation means fetching data from shared memory, and writebacks write cache data into shared memory. Thus, they are both accessing a shared resource which the accelerator also accesses via DMA. Therefore, conflicts can occur between "alloc" and the last DMA operation, or between "wb" and the last DMA operation, if there is no sync occurring between them and if they access any overlapping addresses. Note that although we are not explicitly considering cache prefetching, it is safe to model a cache allocation that is triggered by a cached read the same as a prefetched cache allocation.

```
if (op == "alloc" ||
    op == "wb")
{
  // Assume currNode is the "alloc" or "wb" node
  hasRace(lastDmaOp, currNode);
}
```

Thirdly, cache flush nodes do not spawn any new nodes and are obviously only applicable to cached writes, as cached writes may alter data values and cache flushes forcefully write cache data into shared memory. Thus, all dangling writebacks before a cache flush that have not been flushed already and that are in the same address range have an arrow pointing to the flush node. This means that the data in the flushed address range are no longer dirty in the cache.

Another type of node that does not generate any new nodes is the sync node. The last DMA operation that has not yet been synchronized with the CPU code has an arrow that points from it to the first sync node that happens after it. This is because, upon executing a sync instruction, the CPU will wait for all previously issued DMA operations to complete before continuing executing any further instructions.

Finally, we will describe DMA nodes. Recall that Direct Memory Access (DMA) is the mechanism by which data transfers between shared memory and the accelerator's internal memory. We first create a do_dma_read or do_dma_write node, which signifies that the CPU issues a DMA request to the accelerator. We then generate a new node called "dma_r" or "dma_w", respectively. Unlike "cr" and "cw" nodes, "dma_r" and "dma_w" nodes do not have an arrow pointing to the next CPU operation; if they did, it would indicate that there are no data races between the accelerator and the CPU. Oftentimes, a program contains a burst of

several consecutive DMA requests if the processor communicates with the accelerator and vice versa. In this case, if there is no sync between two DMA nodes ("dma_r" and "dma_w"), then there is an arrow from the earlier DMA node to the later one, respecting the original program order.

DMA operations can conflict with relevant dangling writebacks due to the same reasoning as how "wb" conflicts with the last DMA operation.

```
if (op == "dma_r" )
{
  // Assume currNode is the "dma_r" node
  hasRace(lastWbs, currNode);
}
if (op == "dma_w" )
{
  // Assume currNode is the "dma_w" node
  hasRace(lastWbs, currNode);
  hasRace(lastAllocs, currNode);
}
```

For a summary of the semantics of each type of node, please see Table 3.1.

Now that we have covered the basics of how to construct nodes in a graph of memory operations, consider again the simple program in Figure 3.1. Going through its trace line by line means that we build the nodes of the corresponding graph on the fly, one by one. The process of building this graph is shown in Figure 3.3.

3.3 Full Ruleset for a Non-Coherent Accelerator

There are subtle details in our theoretical framework that are not mentioned in the previous section (Section 3.2), which will be covered here. Below is the full ruleset that our approach and our race detection tool follow when analyzing a program intended for a non-coherent accelerator to find potential data races. To facilitate describing the rules, let *CPUop* be the following set of CPU operations: {cached_read, cached_write, cache_flusha, do_dma_read, do_dma_write, sync}. Also, *DMAop* is defined as {dma_r, dma_w}.

Several rules below use the concept of a *bloated address range*. Recall that each node has an inclusive address range $[low, high]$. The CPU cache reads and writes data from and to shared memory in chunks, where the size of the chunk

depends on the cache. The size of a read from shared memory is generally a multiple of the cache line size, which means that reads are *cache line granularity*. A cache flush is also performed at cache line granularity. The size of a write to shared memory is often a multiple of the cache line size as well but may differ depending on the cache, and is thereby called the *writeback granularity*. Therefore, to reflect the actual address range that the CPU cache touches in shared memory, the address range of "alloc" and "cache_flusha" nodes are modified so that *low* and *high* are multiples of the cache line size, where these multiples still cover the range $[low, high]$. Specifically, assuming that the bloated address range is $[bloated\_low, bloated\_high]$, $bloated\_low$ is the largest address that is a multiple of the desired granularity equal to or below *low* ; $bloated\_high$ is the smallest multiple of the desired granularity that is larger than *high*, minus one. In addition, the address range of "wb" nodes is likewise modified so that *low* and *high* are multiples of the writeback granularity. For example, if an address range is $[10, 59]$ and the granularity is 64, then the bloated address range would be $[0, 63]$.

1. If "uncached_read", generate an uncached_read node with the specified address range.

   (a) If there was a previous *CPUop*, draw an arrow from it to the uncached_read node.

   (b) If there exist any previous dangling writebacks (wb nodes) whose address range overlaps with this uncached_read node's address range, then check if there is a path from each of those wb nodes to this uncached_read node, or a path from this uncached_read node to those wb nodes. If there is no path either way — in other words, there is no path both from a relevant wb node to this uncached_read node and vice versa[1] — flag that there is a race and stop the analysis[2].

[1] With the current ruleset, a path would not exist from the uncached_read node to a previous wb node. However, to the best of our knowledge, the vice versa check is a conservative approach to cover any possible cases (perhaps, if more rules are constructed in the future) where such a path could exist. The reasoning behind performing the vice versa check, though it is currently trivial because there can be no such path, applies to all other vice versa checks in Section 3.3.

[2] In some processors, they may check uncached reads against dirty data in the cache. In such processors, there is no race. Here, we choose to be agnostic and conservative, and report a race.

(c) Check a chain of past DMA writes (dma_w nodes) back to the last known sync or where there was a dma_r or dma_w that *happened-before*. If any of these dma_w nodes have an address range that overlaps with this uncached_read node's address range, then check if there is a path from that dma_w node to this uncached_read node, or vice versa. If there is no path either way, flag that there is a race and stop the analysis. No check is necessary if the last DMA operation is a DMA read (dma_r node) because read/read is not a race.

2. If "uncached_write", generate an uncached_write node with the specified address range.

   (a) If there was a previous *CPUop*, draw an arrow from it to the uncached_-write node.

   (b) If there exist any previous dangling writebacks (wb nodes) whose address range overlaps with this uncached_write node's address range, then check if there is a path from each of those wb nodes to this uncached_write node, or vice versa. If there is no path either way — in other words, there is no path both from a relevant wb node to this uncached_write node and vice versa — flag that there is a race and stop the analysis[3].

   (c) Check a chain of past *DMAop* (dma_r or dma_w nodes) back to the last known sync or where there was a dma_r or dma_w that *happened-before*. If any of these *DMAop* nodes have an address range that overlaps with this uncached_write node's address range, then check if there is a path from that *DMAop* node to this uncached_write node, or vice versa. If there is no path either way, flag that there is a race and stop the analysis.

3. If "cached_read", generate a cached_read node with the specified address range.

[3] Similar to uncached reads, we assume that the processor will bypass the cache. This is a safe assumption because simple processors do this. Again, we choose to be conservative and report a race.

(a) If there was a previous *CPUop*, draw an arrow from it to the cached_-read node.

(b) Generate a cr node with the same address range. Draw an arrow from the cached_read node to this cr node. When the next *CPUop* is known in the future, draw an arrow from the cr node to it.

(c) If the read is aligned (the start address is a multiple of the cache line size), generate an alloc node with a bloated address range, $alloc_1$. Draw an arrow from this alloc node to the cr node from Step 3b.

(d) If the read is unaligned and the specified address range fits in one cache line, generate an alloc node with a bloated address range, $alloc_1$. Draw an arrow from this alloc node to the cr node from Step 3b.

(e) If the read is unaligned and the specified address range does not fit in one cache line, generate two alloc nodes, each with a bloated address range, $alloc_1$ and $alloc_2$. Draw an arrow from each alloc node to the cr node from Step 3b. Note: Only one of {3f}, {3g, 3h}, or {3i} sets of steps can happen at one time. For whichever set of steps occurs, perform the same set of steps for both $alloc_1$ and $alloc_2$.

(f) If an alloc's address range is the first cached operation after a flush with which its address range overlaps, draw an arrow from the cr node from Step 3b to that alloc node. Note that cache flushes, like allocs, are also cache line granularity.

(g) If there is a previous dangling wb node whose address range overlaps with that of alloc, draw an arrow from that wb to this alloc.

(h) If there is a previous dangling wb node (wb) whose address range overlaps with that of alloc, generate a wb node (wb') with the same address range as that of the dangling wb node (wb). Draw an arrow from the cr node from Step 3b (i.e., the parent of the alloc node) to this generated wb node (wb'). The writeback node needs to be propagated because this writeback could happen before or after the alloc node associated with this cached_read (see also Theorem 3.4.6).

(i) If there is a previous alloc node whose address range overlaps with that of alloc, draw an arrow from that alloc to this alloc.

(j) Check a chain of past DMA writes (dma_w nodes) back to the last known sync or where there was a dma_r or dma_w that *happened-before*. If any of these dma_w nodes have an address range that overlaps with the address range of the alloc node(s) ($alloc_1$ and, if it exists from Step 3e, $alloc_2$), then check if there is a path from that dma_w node to $alloc_1$ (and $alloc_2$), or vice versa. If there is no path either way, flag that there is a race and stop the analysis.

4. If "cached_write", generate a cached_write node with the specified address range.

   (a) If there was a previous *CPUop*, draw an arrow from it to the cached_-write node.

   (b) Generate a cw node with the same address range. Draw an arrow from the cached_write node to this cw node. When the next *CPUop* is known in the future, draw an arrow from the cw node to it.

   (c) If the write is aligned (the start address is a multiple of the writeback granularity), generate a wb node with a bloated address range, wb_1. Draw an arrow from the cw node from Step 4b to this wb node.

   (d) If the write is unaligned and the specified address range fits in one writeback (i.e., the size of the writeback granularity), generate a wb node with a bloated address range, wb_1. Draw an arrow from the cw node from Step 4b to this wb node.

   (e) If the write is unaligned and the specified address range does not fit in one writeback granularity, generate two wb nodes, each with a bloated address range, wb_1 and wb_2. Draw an arrow from the cw node to each wb node.

   (f) If there is a previous dangling wb node whose address range overlaps with that of any of the wb node(s) generated in this Step 4 (wb_1 and wb_2), draw an arrow from that wb to this overlapping wb.

   (g) Check a chain of past *DMAop* (dma_r or dma_w nodes) back to the last known sync or where there was a dma_r or dma_w that *happened-before*. If any of these *DMAop* nodes have an address range that over-

laps with the address range of the wb node(s) (wb_1 and, if it exists from Step 4e, wb_2), then check if there is a path from that *DMAop* node to wb_1 (and wb_2), or vice versa. If there is no path either way, flag that there is a race and stop the analysis.

5. If "do_dma_read", generate a do_dma_read node with the specified address range.

   (a) Generate a dma_r node with the same address range. Draw an arrow from the do_dma_read node to this dma_r node. If there was a previous *DMAop*, draw an arrow from it to the dma_r node.

   (b) Check if there is a path from any previous dangling writebacks (wb nodes) whose address range overlaps with that of this dma_r to this dma_r node, and vice versa. If there is no path either way, flag that there is a race and stop the analysis.

6. If "do_dma_write", generate a do_dma_write node with the specified address range.

   (a) Generate a dma_w node with the same address range. Draw an arrow from the do_dma_write node to this dma_w node. If there was a previous *DMAop*, draw an arrow from it to the dma_w node.

   (b) If there exist any previous dangling writebacks (wb nodes) whose address range overlaps with this dma_w node's address range, then check if there is a path from each of those wb nodes to this dma_w node, or vice versa. If there is no path either way, flag that there is a race and stop the analysis.

   (c) If there exist any previous floating allocs (alloc nodes) whose address range overlaps with this dma_w node's address range, then check if there is a path from each of those alloc nodes to this dma_w node, or vice versa. If there is no path either way, flag that there is a race and stop the analysis.

7. If "sync", generate a sync node (with no address range).

(a) If there was a previous *CPUop*, draw an arrow from it to the sync node.

(b) Draw an arrow from the last *DMAop* to this sync node. This ends a *DMAop* chain, and none of the *DMAops* in this chain are to be considered any longer.

8. If "cache_flusha", generate a flush node with a bloated address range.

   (a) If there was a previous *CPUop*, draw an arrow from it to the flush node.

   (b) Draw an arrow from any dangling wb nodes whose address range overlaps with that of flush.

3.4 Theorems and Proofs

The following theorems justify the correctness of our approach. Each of the following theorems tie into the rules described in Section 3.3; which rules a theorem supports are made explicit under each theorem in "Related rule(s)". Assume that the nodes discussed in each theorem access the same or overlapping memory range.

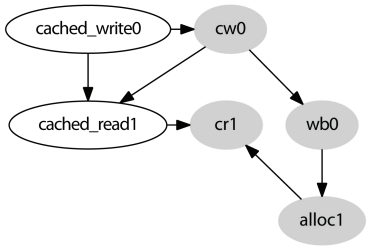


Figure 3.4: Assume each operation in the graph accesses the same part of memory. Adding an arrow from a "wb" node to an "alloc" node does not add more order, and adds more possible races. There is a missing propagated "wb" node dangling from "alloc1" not shown here, as this is due to another theorem (Theorem 3.4.6). For more details, please see Figure 3.6.

Theorem 3.4.1. Adding an arrow from a "wb" node to an "alloc" node does not add more order, and adds more possible races. The number of possible races is positively monotonically increasing.

Related rule(s): 3g

Proof. Note that adding an arrow from a "wb" node to an "alloc" node signifies that the writeback *may* happen before the allocation. However, it may not have happened and could have happened after the allocation (i.e., the cached read); this case is handled in Theorem 3.4.6. This means that there are strictly more writes and reads, which means there are strictly more races possible. For example, in Figure 3.4, the arrow from the "wb0" node to the "alloc1" node signifies that "wb0" may happen before but certainly not after"alloc1".

The problem that this theorem is addressing is how to make the write visible, which occurs when the value written is transferred from the CPU cache to shared memory. There are two situations that can transpire depending on when the write-back occurs. First, it could be that the writeback does not happen (until later); thus, the allocation also does not happen since the value can be read directly from the cache. The generated nodes in light grey are "*maybes*"; they *may* happen. In this case, to make the write visible, the "wb" node must be propagated after the cached read and is left dangling (i.e., unordered). As an aside, this is a point to consider in Theorem 3.4.6's proof. Second, it could that the writeback *does* occur at that moment in the program execution, resulting in an allocation for the succeeding cached read operation. Then, the write has already been made visible since it has been written back, and there seems to be no need to propagate the writeback in this scenario. The proof of Theorem 3.4.6 shows otherwise. □

Theorem 3.4.2. Adding an arrow from an "alloc" node to another "alloc" node does not add more order, and adds more possible races. The number of possible races is positively monotonically increasing.

Related rule(s): 3i

Proof. Again, note that adding an arrow from an "alloc" node to another "alloc" node signifies that two allocations could occur. However, in reality, only one would probably occur. Nonetheless, our model takes the conservative approach and follows the rule that every cached read node has an associated "alloc" node in order to make handling cached reads consistent. Having an arrow from an "alloc" node to another "alloc" node does not add new order because it follows cache order. Since

there are more “alloc” nodes (reads) than necessary, there can only be strictly more races possible. □

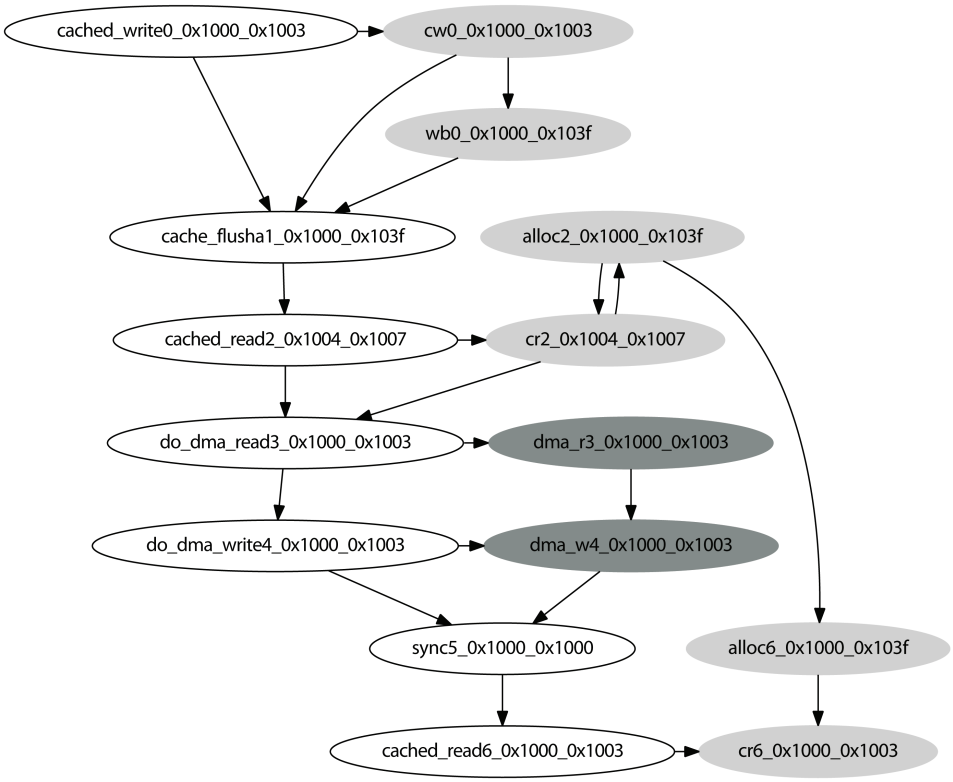


Figure 3.5: Handling cached memory operations appropriately after a CPU cache flush, for instance, by following the rule outlined in Theorem 3.4.3, allows for the race detection tool to avoid false warnings of a potential race possibly after every cache flush.

Theorem 3.4.3. Assume there is no spontaneous prefetching done by the CPU cache. Adding an arrow from a “cr” node to its “alloc” node after a cache flush, if a cached read is the first cached memory operation after the flush, and another arrow from the first cached memory operation after the flush to the next “alloc” node, if the memory range it accesses overlaps with that of the first cached memory operation after the flush, removes erroneous races that would otherwise be detected by the race detection tool after a CPU cache flush.

Related rule(s): 3f

Proof. Since the CPU cache cannot do anything after a cache flush until a cached operation occurs, drawing an arrow from the first cached memory operation after a flush to that memory operation's "alloc" node (if it is a cached read) and to the next "alloc" node no matter what its address range follow cache order.

Figure 3.5 shows an arrow from "cr2" to "alloc2" as "cached_read2" is the first cached memory operation after a flush ("cache_flusha1"). Therefore, the "alloc2" node is not a floating node. Semantically, this means that the cache does not do any work after a flush before there is an instruction to do so.

Furthermore, in Figure 3.5, the second cached read operation ("cached_read6") accesses a memory region that overlaps with the first cached read. Hence, the race detection tool draws an arrow from "alloc2" to "alloc6". Without the Theorem 3.4.3 rule, the tool would erroneously flag a race at the "cached_read6" operation because "alloc6" would be unordered with respect to "dma_w4" in Figure 3.5. □

Theorem 3.4.4. Loop invariant: We have enough information to detect all possible races as memory operations are added one at a time.

Related rule(s): 1b, 1c, 2b, 2c, 3j, 4g, 5b, 6b, 6c

Proof. The data race detection tool keeps track of only the nodes in the *active frontier*, which provides enough information detect all possible races. The *active frontier* is a set of nodes that are unordered in the graph, which implies that they are the only nodes that could be part of a data race. Removing nodes from the active frontier cannot eliminate any races, because they are only removed if they become ordered during the program analysis as the graph is built one memory operation at a time.

Recall that a race is defined in terms of an execution. Our analysis does not create more edges than in a graph of all the memory operations in a given program, referred henceforth as the "full graph". If there is a race in the full graph, as long as the tool creates the two nodes that are involved in the race, then there is a race in the partial graph. (A partial graph consists of only the nodes in the active frontier; it is a subgraph of the full graph.) This always happens since the tool builds the graph

one node at a time and does not add any extra edges compared to the full graph. In particular, when the tool creates the two racy nodes, it is when they matter, in other words, when they can execute under our abstraction model. This works because a node follows an instruction.

The proof for the other direction — if the partial graph has a race, then the full graph has a race — is discussed in the proof for Theorem 3.4.5. □

Theorem 3.4.5. An active frontier is sufficient to be able to check for races and thus, we can ignore all previous memory operations.

Proof. In order to prove Theorem 3.4.5, let's consider the two following statements:

- Any race that could happen in the full, unbounded graph can be detected with only the active frontier.
- Conversely, any race in the active frontier is a race in the full, unbounded graph.

If nodes are deleted from the analysis graph, then both the mistake of missing a race in the full graph and the mistake of flagging a race that is not present in the full graph must not happen. Theorem 3.4.4 has already shown that, if a race exists in the full execution (full graph) and since the tool is guaranteed to build nodes eventually and one at a time, creating fewer edges than in the full graph, then a race exists in the partial execution (partial graph).

Consequently, the only case that would pose a problem in being able to detect a data race with the active frontier is if one node is deleted (i.e., removed from the active frontier), before the other is built by the tool. This leads to a case analysis. Firstly, there are several scenarios where if the operation to be deleted first is a writeback (wb), as listed below:

1. wb $\rightarrow$ cache flush: wb must happen before the flush deletes wb. If the flush comes before another conflicting node, then this is not a race. On the other hand, if the flush comes after the conflicting node, the tool would detect a race because wb has not yet been deleted by the flush.

   Related rule(s): 8b

2. $wb_0 \rightarrow wb_1$: In this case, wb_1 is deleted. A conflicting node cannot be an alloc or a wb because arrows are drawn from wb_0. The conflicting node can only be a node that does not participate in cache behaviour. If the conflicting node is before wb_1, then the tool would have flagged the race already. If the conflicting node is after wb_1, then there could be a race or no race with wb_1. If there is a race involving wb_1, then our approach would have preserved a race. If there is no race involving wb_1, then there is also no race with wb_0 because the racy node is guaranteed to be either before or after wb_1; thus, a race did not escape detection.

   The only way that the tool would miss catching a race in this case is if the conflicting node is after the second cached_write (associated with wb_1) but is also somehow before wb_1; however, this scenario is impossible with our rules, which are fully defined in Section 3.3.

   Related rule(s): 4f

3. wb $\rightarrow$ alloc: wb is propagated further in this case (see Theorem 3.4.6). The racy node that conflicts with wb would not be a cached operation because cached operations are always in order. Recall that alloc is generated with a cached_read. Then, if wb happens before the cached_read in program order, the tool would catch the race before deleting wb. If wb happens after the cached_read and the alloc conflicts with the dangling wb, then a race would have been preserved. Otherwise, if the alloc does not conflict with the propagated, dangling wb, then it is also not a race with the deleted wb. It is impossible for the conflicting node to occur between wb and alloc due to our rules; there is no rule that would lead to this situation.

   Related rule(s): 3g

□

Theorem 3.4.6. Propagate a new "wb" node after a "cr" node (as its child) if there is a relevant dangling writeback before that "cr", where "relevant" means that the "wb" node and the "alloc" node associated with "cr" access overlapping memory addresses. The "wb" and "alloc" nodes are used to check for relevancy because

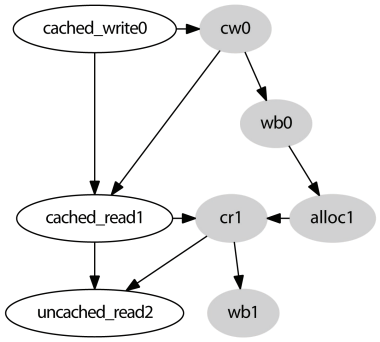


Figure 3.6: Assume that each node accesses overlapping memory addresses. wb1 is propagated as a child of cr1 since there is dangling writeback (wb0) that accesses the same or overlapping memory region as cached_read1. The tool would miss a race if this writeback was not propagated.

they both use a bloated address range, which help to detect issues similar to false sharing (see Section 5.3).

Related rule(s): 3h

Proof. With this rule stated in Theorem 3.4.6, there are two writeback nodes that signify the same writeback operation in the graph; there are two of them due to propagation of the writeback. This does not pose any problems because still, only one of them happens; the tool does not know when it happens — whether it happens before or after the alloc associated with the cached read node. Bear in mind that if the writeback does not happen, then the allocation also does not happen.

Figure 3.6 emphasizes the need for this rule. When the tool arrives at the uncached read node (uncached_read2) during analysis, the tool should flag a potential race with the writeback from cached_write0 because it is unknown whether the writeback has already happened or not at the time of this uncached read. Therefore, the tool must propagate the dangling writeback by creating a new "wb" node after the cr1 node. Now, with that propagated node, the tool is able to flag a potential race between the uncached read (the uncached_read2 node) and the writeback (the wb1 node). □

Observe that Theorem 3.4.1, Theorem 3.4.2, Theorem 3.4.3, and Theorem 3.4.6

conservatively capture any possible race, therefore, making our approach sound.

Chapter 4

Implementation

Dynamic race detection aims to find data races in a program while it is executing. It analyzes one execution of the program at a time. Dynamic race detection can detect data races more easily than if done statically. Static race detection runs before the program is executed. It is not as scalable as the dynamic approach, as it must consider all the possible paths in the control flow of the program. Additionally, it is difficult for static race detectors to perform alias analysis, let alone avoiding false positives in results. On the other hand, dynamic race detectors know exactly the memory locations of dynamically allocated variables and when they are accessed.

Hence, the current implementation performs dynamic race detection on a given program in C; the workflow is shown in Figure 4.1. The program must be instrumented to keep track of every memory operation, compiled, and then executed. The instrumentation causes the program to print the specific memory operations that were done in that execution into a trace file. Then, our tool reads through the trace file to detect whether there are any races or not in the recorded execution of the program.

Firstly, Section 4.1 introduces the VectorBlox architecture which the implementation is tailored for. Next, Section 4.2 describes what happens in the transition from a program to its trace, i.e., the first arrow of the workflow in Figure 4.1. Section 4.3 details how the following step — the trace analysis — is implemented in the race detection tool.

Figure 4.1: The workflow starts from a given input program that is instrumented to print out every memory operation and then executed so that the analysis is done on a single execution. The outputted memory operations are aggregated if necessary into a final trace file that is read by our dynamic race detection tool, which will inform the user of potential races in that one execution.

4.1 VectorBlox Architecture

All devices see a single global address space. The processor possesses an instruction cache and a data cache, the latter of which we will denote as *cpu-cache*. We consider only the data cache because only the values in the data cache that are flushed to main memory can conflict with values originating from the accelerator. Therefore, we are only concerned with addresses used to access external DRAM.

We assume that there exist only physical addresses (no virtual addresses). Also, VectorBlox has no prefetching mechanism for the CPU cache. Nonetheless, it is safe to model a cache allocation that is triggered by a cached read the same as a prefetched cache allocation; there is no distinction between how we treat them regardless of whether or not it is a prefetch.

The CPU component typically executes the following scalar instructions: load (`ld`), store (`st`), and flush (`flush`). For each of these instructions, the CPU operates on a given address, e.g., `ld addr`. At VectorBlox, the processor issues `ld` and `st` in-order and synchronously before the next instruction. A `flush addr` instruction probes the CPU cache for a specific address and evicts/writebacks if it is present and dirty. This flush may touch other addresses as well if other addresses are in the same cache line as the address that is being flushed.

DMA instructions consist of read (`rd addr-range`), write (`wr addr-range`), and sync (`sync`), where `addr-range` is a (start-addr, length) pair. The DMA `rd` and `wr` instructions are issued and executed in-order, one at a time until

the range is exhausted, but are also executed asynchronously (i.e., they are concurrent with future instructions). As for the `sync` operation, it is synchronous blocking and only returns after all previously issued DMA operations have completed.

4.2 Instrumentation and Execution

Dynamic race detection needs a trace of all operations that could be involved in a race and that were run during an execution of a program. In this case, the potentially racy operations are all memory operations.

For the workflow demonstrated in this thesis, a user manually modifies the code so that, for every memory operation, there is a print statement that prints out the type of memory operation and the address range that was operated on. This instrumentation could be automated instead but this is orthogonal to the thesis. Figure 4.2 illustrates an instrumented program (the same program from Figure 3.1). Compiling and executing this instrumented program produces a trace file, which displays the memory address range of memory operations.

For efficiency purposes, some processing of the raw trace file consolidates consecutive memory accesses into one line into a new version of the trace. For example, on line 10 in Figure 4.2, the program populates the entire *a* array through cached writes. However, the `printf` statement prints out the address of each element one at a time, resulting in an unnecessarily verbose trace, especially if the size of the array (*N*) is a large number:

```
cached_write 0x7ffd97898fd0-0x7ffd97898fd0
cached_write 0x7ffd97898fd1-0x7ffd97898fd1
...
cached_write 0x7ffd97898fd9-0x7ffd97898fd9
```

I wrote a Python program that condenses the trace file, if possible. For example, the above lines can be combined into one line since they are the same memory operation and the memory addresses are consecutive. This one line becomes:

```
cached_write cached_write 0x7ffd97898fd0-0x7ffd97898fd9
```

Figure 4.3 shows the final, condensed trace file for our running example.

This covers the first part of the workflow. The next section describes how we use the outputted trace file to detect potential races.

```c
int8_t main(void)
{
    int8_t i;
    int N = 10;
    int8_t a[N];
    int8_t scratchpad[N]; // accelerator's internal memory

    for (i = 0; i < N; i++)
    {
        a[i] = read_mic();
        printf("cached_write %p-%p\n", &a[i], &a[i] + sizeof(int8_t) - 1);
    }

    vbx_dcache_flush(a, N);
    printf("cache_flusha %p-%p\n", a, a + (N*sizeof(int8_t)) - 1);

    vbx_dma_to_vector(a, scratchpad, N);
    printf("do_dma_read %p-%p\n", a, a + (N*sizeof(int8_t)) - 1);
    test_vector_fft(scratchpad, N);
    vbx_dma_to_host(scratchpad, a, N);
    printf("do_dma_write %p-%p\n", a, a + (N*sizeof(int8_t)) - 1);

    vbx_sync();
    printf("sync\n");

    printf("cached_read %p-%p\n", &a[0], &a[0] + sizeof(int8_t) - 1);
    return a[0]; // cached_read
}
```

Figure 4.2: A simple program (similar to Figure 3.1 but with a larger N) that has been instrumented to print out every memory operation, including the type of operation and the memory address range being manipulated. `vbx_dma_to_vector` function has the destination as the first parameter and the source as the second parameter.

```
cached_write 0x7ffd97898fd0-0x7ffd97898fd9
cache_flusha 0x7ffd97898fd0-0x7ffd97898fd9
do_dma_read 0x7ffd97898fd0-0x7ffd97898fd9
do_dma_write 0x7ffd97898fd0-0x7ffd97898fd9
sync
cached_read 0x7ffd97898fd0-0x7ffd97898fd0
```

Figure 4.3: Consolidated trace file generated from the program in Figure 4.2, where the hexadecimals are the memory address range of the variable that is being accessed.

4.3 Trace Analysis

The race detection tool, written in C++, implements the rules delineated in Section 3.3, building each node on the fly. It parses a trace file line by line in order to build these nodes and uses map data structures to record nodes in the active frontier[1]. When nodes in the active frontier become ordered in the graph, they can be safely removed from the active frontier because either they cannot be racy or the same race would affect a newer node in the active frontier. Therefore, the graph that the tool builds can be pruned; in other words, the tool only tracks the nodes in the active frontier, ignoring all other operations. This reduces the amount of memory used for the analysis step.

[1]Recall from Section 3.4 that the active frontier consists of the nodes that could be involved in a race in the future as of the point where the tool has come so far in analyzing the program. For example, the newest write to a memory location is obviously part of the active frontier, whereas writebacks that already have been flushed are not. Dangling writebacks and floating allocs are also part of the active frontier, where a dangling writeback is a writeback node that the tool created at the earliest point in time that the writeback could occur but it has not occurred yet for certain (the writeback will take place at some later point in time), and similarly, a floating alloc node is created retroactively when the node that requires the allocated cache line is created.

Chapter 5

Evaluation

Evaluation involved analyzing real-world examples for races using the dynamic race detection tool implemented for this thesis, which applies the rules defined in Section 3.3, and validating the effectiveness of the tool.

5.1 Experimental Setup

To analyze real-world examples, I ran my dynamic race detection tool on eleven VectorBlox open-source examples[1] provided on GitHub. Select examples in the GitHub repository were chosen to be used in the experiments if they were non-trivial and were written in C. All the examples were executed using a VectorBlox simulator, assuming a single core with its own cache and one vector accelerator. The examples included basic math computations (such as vector addition, etc.), signal processing algorithms, and image processing algorithms (Table 5.1).

5.2 Time Required for Analysis

The shortest runtime took 30 milliseconds (ms) for 6,000 lines in the trace file (vbw_libfixmath example), and the longest runtime took approximately 815 million ms, or around 9 days, for almost 29 million lines in the trace (vbw_mtx_sobel example). The analysis for ten out of the eleven examples completed in less than

[1] Retrieved on November 21, 2019. Hash version c7a4894206.
https://github.com/VectorBlox/mxp/tree/master/examples/software/bmark

Table 5.1: Benchmark examples

Test Name	Test Description
vbw_libfixmath	Square root and division
vbw_mtx_fir_t	2D Finite Impulse Response (FIR) filter using matrices
vbw_mtx_median_argb32	Median filter (using bubble sort) with 32-bit data type
vbw_mtx_median_t	Median filter (using bubble sort) with 8-bit data type
vbw_mtx_motest*	Motion estimation
vbw_mtx_sobel	Sobel filter (e.g., used in edge detection algorithms)
vbw_mtx_xp_t	Matrix transpose
vbw_vec_add_t*	Vector addition
vbw_vec_fft	Fast Fourier Transform (FFT)
vbw_vec_fir_t	FIR filter using vectors
vbw_vec_power_t	Vector power

*This example contained a race.

45 minutes each. Figure 5.1 provides a visual outlook of the analysis runtime for the examples that contained no races, as the tool stops at the first race detected.

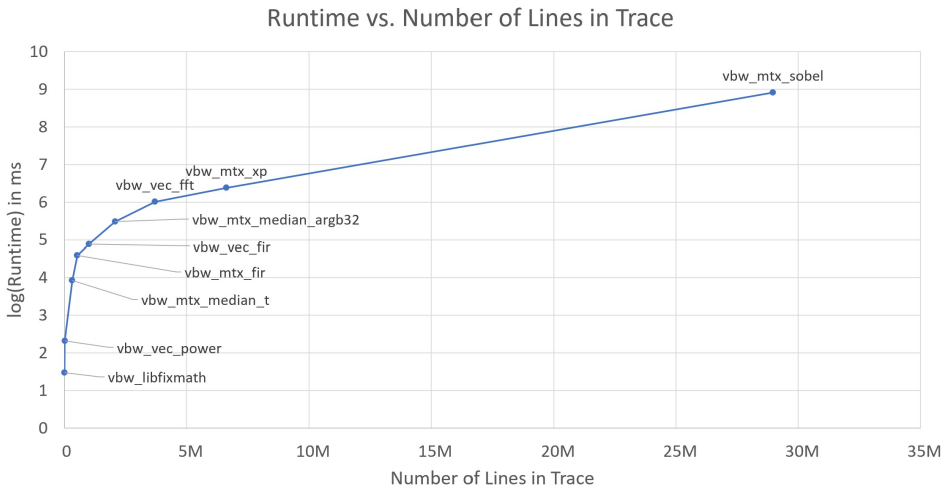


Figure 5.1: This semi-log graph shows the log of the runtime in milliseconds (ms) of each example that completed, i.e., had no races, with respect to the number of lines in the respective trace.

5.3 Experimental Results

The tool found subtle races in two out of the eleven examples evaluated. One example is the vector addition example called "vbw_vec_add_t" in the GitHub repository, which is a test to check that the vector accelerator works correctly. Excerpts of this code from the GitHub repository will be shown below as an example. Firstly, the test allocates various variables:

```
// Assume N is the number of elements in an array.

vbx_mm_t *scalar_in1 = malloc( N*sizeof(vbx_mm_t) );
vbx_mm_t *scalar_in2 = malloc( N*sizeof(vbx_mm_t) );
vbx_mm_t *scalar_out = malloc( N*sizeof(vbx_mm_t) );
```

`scalar_x` variables are named so because they will be manipulated by a scalar processor. They are allocated using the standard library's `malloc`, which means that they are CPU cached variables; hence, any access to them are cached reads or writes. Next, `vector_x` variables are allocated using the VectorBlox Application Programming Interface (API), `vbx_shared_malloc`, whose semantics

mean that they will be treated as uncached variables; they reside in shared memory.

```
vbx_mm_t *vector_in1 = vbx_shared_malloc( N*sizeof(vbx_mm_t) );
vbx_mm_t *vector_in2 = vbx_shared_malloc( N*sizeof(vbx_mm_t) );
vbx_mm_t *vector_out = vbx_shared_malloc( N*sizeof(vbx_mm_t) );
```

`vector_x` variables will be mainly manipulated by a vector accelerator. Finally, `v_x` variables are allocated in the scratchpad, the accelerator's internal memory, but they are not considered by the tool because they are internal only to the accelerator.

```
vbx_sp_t *v_in1 = vbx_sp_malloc( N*sizeof(vbx_sp_t) );
vbx_sp_t *v_in2 = vbx_sp_malloc( N*sizeof(vbx_sp_t) );
vbx_sp_t *v_out = vbx_sp_malloc( N*sizeof(vbx_sp_t) );
```

The vector addition test consists of two tests, performing the same vector addition computation twice, first by a scalar processor and then by a vector accelerator. The first test, which is done by the processor, has two input arrays, `scalar_in1` and `scalar_in2`, whose values are added together and put into an output array, `scalar_out`. After this test is completed, the next test executes, this time carried out by the vector accelerator and where `vector_in1` and `vector_in2` are two input arrays and `vector_out` is the output array that contains the results.

The vector addition test starts by zeroing out the output arrays, `scalar_out` and `vector_out`.

```
VBX_T(test_zero_array)( scalar_out, N );
VBX_T(test_zero_array)( vector_out, N );
```

Then, it initializes the `scalar_in1` array. Afterwards, the values in the cached variable, `scalar_in1`, are copied into the uncached variable, `vector_in1`, in a loop.

```
VBX_T(test_init_array)( scalar_in1, N, 1 );
VBX_T(test_copy_array)( vector_in1, scalar_in1, N );
```

Similarly, the test initializes `scalar_in2` before copying its values into `vector_in2`.

```
VBX_T(test_init_array)( scalar_in2, N, 1 );
VBX_T(test_copy_array)( vector_in2, scalar_in2, N );
```

The program calls the helper function `test_scalar` to perform the test by the CPU and then prints out the output array.

```
scalar_time = test_scalar( scalar_out, scalar_in1, scalar_in2, N );
VBX_T(test_print_array)( scalar_out, PRINT_LENGTH);
```

To check that the vector accelerator can execute instructions correctly, the accelerator first needs to obtain the input values via DMA. The CPU tells the accelerator to do a DMA read through the VectorBlox API call, `vbx_dma_to_vector`. The code below shows DMA reads of `vector_in1` and `vector_in2` in main memory.

```
vbx_dma_to_vector( v_in1, (void *)vector_in1, N*sizeof(vbx_sp_t) );
vbx_dma_to_vector( v_in2, (void *)vector_in2, N*sizeof(vbx_sp_t) );
```

The accelerator executes the vector addition in its scratchpad in the call, `test_vector`, and then DMA writes the results into the uncached variable, `vector_out`, with a call to `vbx_dma_to_host`.

```
test_vector( v_out, v_in1, v_in2, N, scalar_time );
vbx_dma_to_host( (void *)vector_out, v_out, N*sizeof(vbx_sp_t) );
```

On the next line, the VectorBlox synchronization function, `vbx_sync`, instructs the CPU to wait until all previously issued DMA requests have completed. Due to this sync, there is no race at the next CPU operation, which is to print the uncached array, `vector_out`.

```
vbx_sync();
VBX_T(test_print_array)( vector_out, PRINT_LENGTH );
```

The last line in the vector addition test verifies that the results computed by the scalar processor and the vector accelerator were equal, thereby confirming that the accelerator works.

```
errors += VBX_T(test_verify_array)( scalar_out, vector_out, N );
```

The tool detects a potential race at the first DMA read instruction in the "vbw_vec_add_t" example. Specifically, the race occurs between a cached write and a DMA read. This is made apparent in the graph generated by the tool (Figure 5.2), where the race is between the first writeback node (labelled "wb0") and

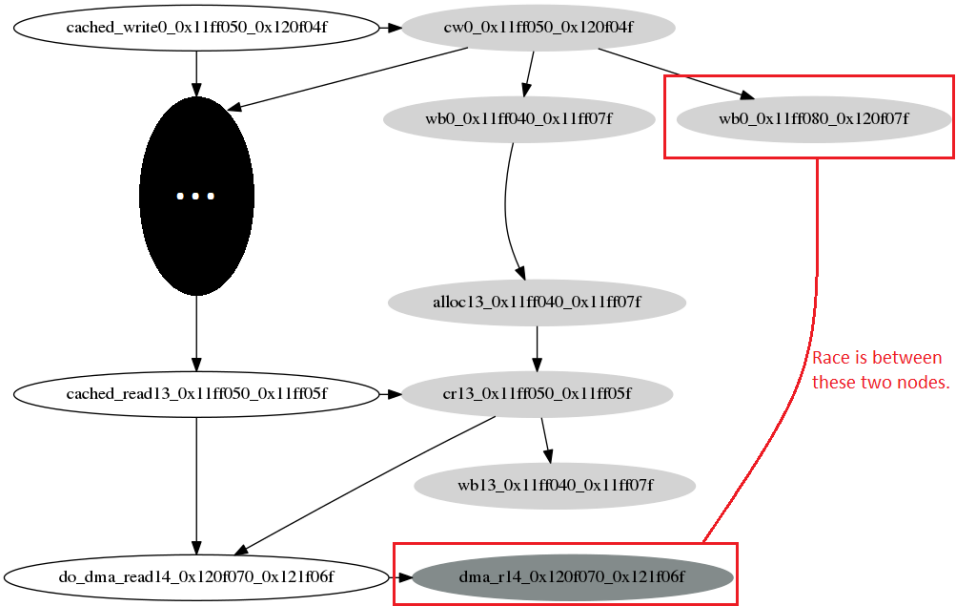


Figure 5.2: This is part of the graph that is generated by our tool as it is analyzing the vector addition example. The large black oval in the middle of the graph signifies that there are several other nodes that are in between the nodes shown but are not important in illustrating the race present.

the first DMA read ("dma_r14"). In the graph, there is a dangling writeback node (wb0_0x11ff080_0x120f07f) that is unordered with respect to the first DMA read node (dma_r14_0x120f070_0x121f06f). Their memory address ranges — in particular, from 0x120f070 to 0x120f06f — overlap with each other. As only one of these accesses is a write (wb0), this is a write-read case. Hence, this situation fulfills the criteria of a data race stated in Definition 2.1.1.

In the code, the dangling writeback node and the DMA read node correspond to the following two lines:

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

The conflicting variables are `scalar_out`, a cached variable that eventually has its values written back when the test zeroes this array, and `vector_in1`, an uncached variable in main memory that is accessed by the accelerator. Although

they are two different data structures, our dynamic tool saw that they are mapped to the same cache line due to them being allocated to consecutive memory locations. Therefore, due to writeback granularity, the `scalar_out` writebacks can overwrite the first few bytes of `vector_in1` in main memory. The DMA read of `vector_in1` may thus read wrong data values. This is similar to false sharing (see Theorem 3.4.6); however, this is a coherence error rather than a performance issue.

The "vbw_mtx_motest" example contained the second race. The tool flagged a potential race between a dangling writeback node from a cached write to `scalar_x_input` and a DMA write to the uncached variable, `vector_result`. Specifically, the race is between wb160616_0x1a29080_0x1a290bf and dma_w160722_0x1a25070_0x1a3506c. The guilty lines in the program are below; the `vbw_mtx_motest_byte` function does a DMA write to `vector_result` at the end.

```
init_motest( scalar_x_input, scalar_result );
...
error_rc = vbw_mtx_motest_byte( vector_result, vector_x_input, vector_x_input, &m );
```

Similar to the first race condition, `vector_result` and `scalar_x_input` are two different data structures but overlap the same cache line because `scalar_x_input` is the next variable allocated after `vector_result`. Again, this condition satisfies the data race definition, as this is a write-write case that accesses the memory address range, from 0x1a29080 to 0x1a290bf, simultaneously.

5.4 Injection of Bugs in Examples

To verify that the implemented tool was able to catch more types of races than what were discovered in real-world examples, I also performed experiments in which I introduced bugs in the VectorBlox code examples by removing necessary synchronization. Not surprisingly, my race detection tool was able to detect these bugs easily in all cases.

Interestingly, in at least one case, the VectorBlox simulator still showed the test as passing, even though I had eliminated necessary synchronization. Note that there exists other sync calls in the program but the particular sync that was removed turned out to be essential to avoiding potential future problems. This high-

lights how data races can produce elusive bugs, which are not caught in debugging, but emerge only late (and sporadically) in a production system — and hence the importance of data race detection tools such as I propose.

To demonstrate the impact of these bugs, I will now go through one example in detail, specifically, the VectorBlox example that calculated vector power. Figure 5.3 shows the operation immediately before the sync (`do_dma_write`) in the program and immediately after the sync (`uncached_read`). There is no race here as the `dma_w` node is ordered with respect to the sync node, and the sync node is ordered with respect to the `uncached_read` node as well. Since orders are transitive, the `dma_w` node is guaranteed to happen before the `uncached_read` node. On the other hand, in Figure 5.4, where the sync has been removed, the `dma_w` node is now unordered with respect to the `uncached_read` node because there is no happens-before relation (or path from) the `dma_w` node to the `uncached_read` node. Because the DMA write and the uncached read access overlapping memory regions, there is a potential race.

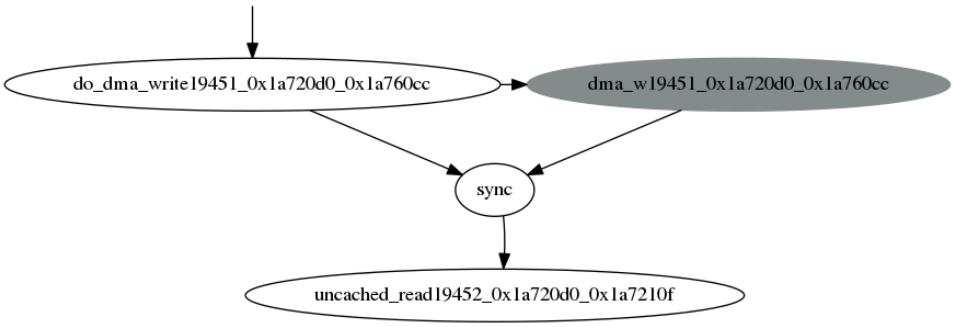


Figure 5.3: The vbw_vec_power_t VectorBlox example's original program order which includes synchronization.

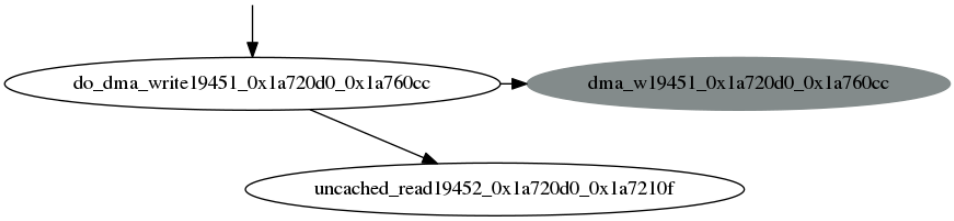


Figure 5.4: Removed a sync operation in the vbw_vec_power_t VectorBlox example, thereby introducing a bug in the program. On the VectorBlox simulator, the race does not happen, so the code behaves correctly, and the test "passes". However, there is no guarantee that any given implementation will happen to order the two operations in the same way that the simulator did. Most importantly, the race detection tool is able to detect this data race.

Chapter 6

Related Work

As we use Lamport's happens-before relation, an obvious related work is the Lamport timestamps algorithm [9] and the more advanced vector clock method [5], which builds on Lamport's work. These two algorithms are used to determine ordering of events in distributed systems.

Lamport's logical clocks are functions that assign a number to an event in a process and do not necessarily have a relation to physical time. Essentially, each process has a counter and increments it before any event occurs in that process. This counter value is included in any message sent by the process. A receiving process updates the counter to be the maximum of its current counter and the timestamp received in the message, and then increments that number by one to consider the message received. Furthermore, Lamport's timestamps satisfies the condition: if event $a \rightarrow$ event b, then $C(a) < C(b)$, where $C(x)$ is the logical clock of an event x. However, the converse is not true. The vector clock method addresses this limitation and satisfies that condition both ways.

Our solution differs from the Lamport timestamp and vector clock algorithms, as we rely on program order to create partial ordering in the graph produced. Causality between events is clear in a program because the CPU is the "master" of the system, telling itself, its cache, and the accelerator what to do.

Multithreaded and concurrent programs have generated much research in both static and dynamic race detection tools. For example, RELAY [18], Warlock [16], RacerX [4], and Locksmith [12] statically check for races by performing lockset

analysis. rccjava [6],[2] (which uses extended static checking [3]) is another static tool but uses theorem provers and type checkers. Although static detection tools offer scalable solutions, they generally work with abstracted versions of a program and thus, can produce a large number of false alarms.

Eraser [13] and FastTrack [7] are examples of dynamic race detectors. Eraser tracks the set of locks held by each shared variable in a thread but is prone to false alarms. FastTrack exploits lightweight vector clocks and performs comparably to Eraser but is more precise and never reports false alarms. FastTrack claims that the majority of data in multithreaded programs is either thread local, lock protected, or read shared. Thus, it uses an adaptive representation for the happens-before relation that requires only constant space for these common cases, without any loss of precision or correctness. In contrast to a vector clock-based race detector that records the clock of the most recent write to each variable x by each thread t, FastTrack records the clock and thread identifier of only the very last write to x, where this information is dubbed an *epoch*. The authors of FastTrack assert that all writes to x are totally ordered by the happens-before relation, assuming no races have been detected so far. Hence, the full generality of vector clocks is not needed in this case. Similarly, since reads on thread-local and lock-protected data are totally ordered — assuming no races have been detected — FastTrack records only the epoch of the last read to these types of data. FastTrack adaptively switches from epochs to vector clocks (and vice-versa) in order to guarantee no loss of precision.

To the best of our knowledge, there is no race detection approach in a non-coherent and heterogeneous context, for which our work is targeted.

Chapter 7

Conclusions

In conclusion, this thesis presents a novel approach to detect data races in software for non-coherent accelerators in heterogenous systems, and proves its correctness and soundness. The approach generalizes to other hardware architectures and code examples, not only by VectorBlox, because it is mostly architecture-independent. It is mostly but not completely architecture-independent because the model still includes a CPU cache and needs to know the size of the cache line and that of the writeback granularity. Nonetheless, the model does not require any details of how the cache works, which protocol it uses, etc., as our solution abstracts the hardware's behaviour. The approach is also generalizable in the sense that the derived simple rules work for any non-coherent accelerator.

Furthermore, I have built a dynamic race detection tool that successfully found two subtle races in real-world examples. Limitations of the current tool are due to the fact that it performs dynamic race detection (as opposed to static), as well as the fact that it works with an abstraction of the hardware. Dynamic race detection is less sound than static race detection, which means that it can miss races (false negatives), because it does not execute all possible paths in a program. However, our analysis is sound with respect to the given exeuction, in that it is able to find all potential races in that execution. On the other hand, the soundness comes with a loss of precision. We are conservatively abstracting away some details of the hardware's behaviour, which can lead to flagging false alarms (false positives), since the abstraction is an over-approximation and builds upon several assumptions. For

instance, one assumption is that a writeback can happen arbitrarily late but this may not be true due to hardware details.

Future work include implementing a static version of the race detection tool and automatically instrumenting an input program. For example, the static race detection tool can use the LLVM framework [10] to help with the instrumentation and analysis of a program. It would also be interesting to examine ThreadSanitizer [14], a data race detector for C/C++, to see if our solution can work alongside or within it, or to compare against it.

Bibliography

[1] VectorBlox MXP Programming Guide for Xilinx. URL http://vectorblox.github.io/mxp/mxp_guide_xilinx.html. Accessed 2019-09-27. → page 6

[2] M. Abadi, C. Flanagan, and S. N. Freund. Types for safe locking: Static race detection for Java. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(2):207–255, 2006. → page 47

[3] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. *Research Report 159, Compaq SRC*, 1998. → page 47

[4] D. Engler and K. Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 237–252. ACM, 2003. → page 46

[5] C. Fidge. Logical time in distributed computing systems. *Computer*, 24(8): 28–33, 1991. → page 46

[6] C. Flanagan and S. N. Freund. Type-based race detection for Java. In *ACM SIGPLAN Notices*, volume 35, pages 219–232. ACM, 2000. → page 47

[7] C. Flanagan and S. N. Freund. FastTrack: efficient and precise dynamic race detection. In *ACM SIGPLAN Notices*, volume 44, pages 121–133. ACM, 2009. → page 47

[8] D. Giri, P. Mantovani, and L. P. Carloni. Accelerators and coherence: An SoC perspective. *IEEE Micro*, 38(6):36–45, 2018. → page 7

[9] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978. → pages 4, 46

[10] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization (CGO)*, pages 75–86. IEEE, 2004. → page 49

[11] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems (TOCS)*, 7(4):321–359, 1989. → page 6

[12] P. Pratikakis, J. S. Foster, and M. Hicks. LOCKSMITH: Practical static race detection for C. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 33(1):3, 2011. → page 46

[13] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)*, 15(4):391–411, 1997. → page 47

[14] K. Serebryany and T. Iskhodzhanov. ThreadSanitizer: data race detection in practice. In *Workshop on Binary Instrumentation and Applications (WBIA)*, pages 62–71. ACM, 2009. → page 49

[15] A. Severance and G. G. F. Lemieux. Embedded supercomputing in FPGAs with the VectorBlox MXP matrix processor. In *2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 1–10. IEEE, 2013. → page 6

[16] N. Sterling. WARLOCK–a static data race analysis tool. In *USENIX Winter*, pages 97–106, 1993. → page 46

[17] A. L. Varbanescu and J. Shen. Heterogeneous computing with accelerators: an overview with examples. In *2016 Forum on Specification and Design Languages (FDL)*, pages 1–8. IEEE, 2016. → page 1

[18] J. W. Voung, R. Jhala, and S. Lerner. RELAY: static race detection on millions of lines of code. In *ESEC/FSE*, pages 205–214. ACM, 2007. → page 46