Imperial College of Science, Technology and Medicine
Department of Computing

Memory Models for Heterogeneous Systems

Dan Iorga

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of the University of London and
the Diploma of Imperial College, August 2022

Declaration

This thesis and the work it presents are my own except where otherwise acknowledged

Dan Iorga

Abstract

Heterogeneous systems, in which a CPU and an accelerator can execute together while sharing memory, are becoming popular in several computing sectors. Nowadays, programmers can split their computation into multiple specialised threads that can take advantage of each specialised component. FPGAs are popular accelerators with configurable logic for various tasks, and hardware manufacturers are developing platforms with tightly integrated multicore CPUs and FPGAs. In such tightly integrated platforms, the CPU threads and the FPGA threads access shared memory locations in a fine-grained manner. However, architectural optimisations will lead to instructions being observed out of order by different cores. The programmers must consider these reorderings for correct program executions.

Memory models can aid in reasoning about these complex systems since they can be used to explore guarantees regarding the systems' behaviours. These models are helpful for low-level programmers, compiler writers, and designers of analysis tools. Memory models are specified according to two main paradigms: *operational* and *axiomatic*. An operational model is an abstract representation of the actual machine, described by *states* that represent idealised components such as buffers and queues, and the legal *transitions* between these states. Axiomatic models define relations between memory accesses to constrain the allowed and disallowed behaviours.

This dissertation makes the following main contributions: an operational model of a CPU/FPGA system, an axiomatic one and an exploration of simulation techniques for operational models. The operational model is implemented in C and validated using all the behaviours described in the available documentation. We will see how the ambiguities from the documentation can be clarified by running tests on the hardware and consulting with the designers. Finally, to demonstrate the model's utility, we reason about a producer/consumer buffer implemented across the CPU and the FPGA.

The simulation of axiomatic models can be orders of magnitude faster than operational models. For this reason, we also provide an axiomatic version of the memory model. This model allows us to generate small concurrent programs to reveal whether a specific memory model behaviour can occur. However, synthesising a single test for the FPGA requires significant time and prevents us from directly running many tests. To overcome this issue, we develop a soft-core

processor that allows us to quickly run large numbers of such tests and gain higher confidence in the accuracy of our models.

The simulation of the operational model faces a path-explosion problem that limits the exploration of large models. Observing that program analysis tools tackle a similar path-explosion problem, we investigate the idea of reducing the decision problem of "whether a given memory model allows a given behaviour" to the decision problem of "whether a given C program is safe", which can be handled by a variety of off-the-shelf tools. Using this approach, we can simulate our model more deeply and gain more confidence in its accuracy.

Acknowledgements

I am grateful to all the people who have made my PhD experience great. First, I would like to thank my advisers, Alastair Donaldson and John Wickerson, who have been incredibly supportive and provided excellent guidance.

Tyler Sorensen has also been a great mentor throughout this time. He has helped me navigate the research life, provided me with career advice, and joined me for drinks. His PhD students will have a great life in Santa Cruz!

The first years of my PhD have been more enjoyable due to all the friends I made in the HIPEDS cohort. Thank you Johannes Wiebe, George Theodorakis, George Rizos, Jukka Soikkeli, Alexandros Tasos for all the times you joined me for drinks. Also, thank you guys for joining me for swims, runs and gym sessions. I am also happy to have had Filip Paszkiewicz as a housemate with whom I always complained about university life.

I am grateful to the members of the CAS group in EEE who welcomed me into their group when I had to temporarily move from my DoC office. The office environment was great alongside Nadesh Ramanathan, Yann Herklotz, Jianyi Cheng, Samuel Coward, Diederik Vink, Aditya Rajagopal, Ian McInerney, Benjamin Briggs, Benjamin Chua. I enjoyed it so much that I did not want to go back when my DoC office was available.

I am incredibly grateful to my parents and relatives for all the support during this time. I would not have been able to do anything without them.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The end of Dennard scaling in the early 2000s led to CPU designers resorting to duplicating processor cores to make computational gains, exploiting additional transistors that became available year on year thanks to Moore's law [Rup15]. Now, with the future of Moore's law looking uncertain [HP19], this *homogeneous* approach to parallelism is under threat. System designers and application developers must look to *heterogeneous* systems, comprising multiple architecturally distinct computational units, for performance and energy efficiency.

Heterogeneous systems consist of different specialised computing elements that can work concurrently to solve complex tasks. These complex tasks must be partitioned into small sub-tasks that take advantage of the particularities of each distinct computational unit. These distinct specialised computational units can solve sub-tasks more efficiently and energy efficient than a general-purpose processor. As a result, computer architects have been designing systems that integrate general-purpose computation with different types of specialised accelerators.

One promising category of accelerators is field-programmable gate arrays (FPGAs) [Moo17]. FPGAs are composed of configurable logic and memory blocks. This structure allows for many different configurations, which provides the potential to create custom designs that can offer increased parallelism and efficient access to irregular data patterns. Furthermore, FPGAs only need to power the circuitry that is needed to perform the computation of interest and all the circuitry that is not needed is eliminated, resulting in much less energy usage [Xil23]. Generally,

FPGAs are used when a manufacturer wants to avoid the prohibitive cost of manufacturing an application-specific integrated circuit or maintain the flexibility to change the design of the circuit in the future.

A recent trend in heterogeneous systems is to combine a homogeneous multicore CPU with an FPGA. These combined CPU/FPGA systems are of special interest because the FPGA component can be configured to represent one or more processing elements customised for a particular computationally-intensive sub-task. At the same time, the general application can be written to run on the general-purpose CPU. This combination of CPU and FPGA devices has provided significant performance gains in several domains, including video processing [AKK+16], neural networks [GSQ+18], and image filtering [DS13].

Until recently, data movement in CPU/FPGA systems has been *coarse-grained*: large amounts of data are transferred back and forth between the memory spaces of the FPGA and CPU via special `memcpy`-like API calls. However, recent devices – including Intel's Xeon+FPGA system [OSC+11, Int19], the IBM CAPI [SBJS15], the Xilinx Alveo [Xil18] and the Enzian platform [CRS+22]– offer a *fine-grained* shared-memory interface between the CPU and FPGA. This shared-memory interface enables synchronisation idioms where data is exchanged in arbitrary (potentially small) amounts, such as *work stealing*, which has been shown to enable significant speedups in difficult-to-accelerate applications (e.g. [RWWC16, TPO10, FBL+16]).

Combined CPU/FPGA systems with fine-grained shared memory have the potential to accelerate irregular applications in an energy-efficient manner but present significant programmability challenges. They inherit the well-known challenges associated with concurrent programming on homogeneous shared-memory systems and present new challenges due to complex interactions between heterogeneous processing elements that each have distinct memory semantics. Furthermore, fine-grained CPU/FPGA systems are new, and applications that exploit them are only just emerging, so this an opportune time to examine their semantics rigorously and lay solid foundations for compiler writers and low-level application developers.

Writing correct and efficient programs for such systems requires a formal specification of memory semantics, called a *memory consistency model* [AG96b]. The study of memory models

has helped programmers understand concurrent programs for CPUs [OSS09, SSA+11] and GPUs [ABD+15]. However, it is challenging to develop memory models for complex systems and validate their accuracy. Such memory models must accurately describe the interaction between threads that run on different hardware and must account for the idiosyncrasies of each type of thread. Different types of threads with different types of operations can interact with each other leading to some behaviours that are often not intuitive. This thesis shows how this type of heterogeneous system can be modelled and how we can gain confidence that our modelling decisions resulted in an accurate description of the underlying system.

The increased complexity of these systems also leads to more *simulation challenges*. These challenges stem from the many different components with non-deterministic behaviours that generate multiple paths through the execution of the model. These paths must be understood and sometimes explored to guarantee a system's behaviour. However, exploring all these paths must be done efficiently, so that simulation time does not become infeasible. Moreover, there are many cases where just exploring some of these paths can provide the user with sufficient information to understand the system's behaviour. In this thesis, we also explore multiple tools a memory model developer can use to understand a memory model's behaviour.

We provide a detailed formal case study of the memory semantics of Intel's CPU/FPGA systems. These combine a multicore Xeon CPU with an Intel FPGA, and allow them to share main memory [Int19]. We refer to this class of systems as X+F (Xeon+FPGA) throughout.[1]

1.1 Contributions

Operational memory model In chapter 3 we present a formal semantics for the X+F memory system in operational form. To understand the variety of complex behaviours that the system can exhibit, we have studied the available X+F documentation in detail and empirically investigated the memory semantics of the real system that integrates a Broadwell Xeon CPU with an Arria 10 FPGA. The *operational* semantics describes the X+F memory system using

[1] Other works have called these systems HARP e.g. [MKN+18], but we could find no official documentation from Intel using this terminology. Our naming scheme is consistent with recent work [CCF+19].

an abstract machine. We have implemented the operational semantics in C, which is suitable for analysis with the CBMC model checker [CKL04]. This model allows an engineer to explore the possible behaviours of a given memory model litmus test, and supports the generation of counterexamples that can be understood with respect to the abstract machine.

To demonstrate the utility of our formal model, we use it to reason about a producer/consumer queue linking the CPU and the FPGA. First, we investigate various design choices for the queue, using our model to argue why they provide correct synchronisation, and we compare their performance. Then, guided by our model, we develop *lossy* versions of the queue that omit some synchronisation, risking loss or reordering of elements as a result, but in a well-defined manner that our formal model describes. Finally, we present experimental results exploring the performance/quality trade-off associated with these queue variants, which is relevant in application domains where some loss is tolerable, such as image processing and machine learning. We also show that lossy behaviour is exacerbated when the FPGA is configured to contain additional processing elements that stress the shared-memory system; these "enemy" components can serve as a debugging aid to help shake out bugs arising from missing synchronisation.

Axiomatic memory model In Chapter 4 we provide the *axiomatic* semantics that declaratively characterises the executions permitted by the memory system. The axiomatic semantics has been mechanised in the Alloy modelling language [Jac12]. The Alloy Analyzer can then automatically generate allowed or disallowed executions, subject to user-provided constraints on the desired number of events and actors that should feature in a generated execution. We have used the Alloy description of our axiomatic semantics to generate a set of *disallowed* executions that feature only *critical* events (i.e. removing any event from an execution would make the execution allowed). Using a back-end that converts an execution into a corresponding C program, we have used these executions and the CBMC model checker to validate our operational model both 'from above' and 'from below'; that is, every disallowed execution generated from the axiomatic model is also disallowed by the operational model, and removing any event from such an execution causes it to become allowed by the operational model. This back-and-forth process is a compelling demonstration of the value of developing operational and axiomatic

models in concert, which we hope will inspire other researchers to follow suit.

Having gained confidence in the accuracy of our models via this cross-checking process, we proceeded to run tests against hardware both to check that execution results disallowed by the model are indeed not observed (increasing confidence that our model is sound), and to see how often unusual-but-allowed executions are observed in practice. Since synthesising an FPGA design from Verilog takes several hours, performing synthesis on an execution-by-execution basis was out of the question. Instead, we present the design of a soft-core processor customised to execute litmus tests described using a simple instruction set. The processor is synthesised once, after which the CPU can send a series of tests to the FPGA for execution, allowing us to process hundreds of tests in a matter of hours, rather than weeks.

Simulating operational models Simulating the operational memory model is required to decide if a potentially unwanted state of the system can be reached. However, this simulation is the bottleneck in our previous experiments; therefore, in Chapter 5 we search for alternatives. A similar problem is tackled by a wide range of off-the-shelf tools to decide whether a program can reach a particular state and are available for several languages such as C, C++, Java or Python. This leads to the following idea: instead of implementing a bespoke memory model simulator, why not implement the simulator logic as a computer program that takes a particular test scenario as input? Determining whether the test scenario is allowed would then boil down to determining whether a particular state of the program that encodes the memory model is reachable when executed on an input describing the scenario of interest, and off-the-shelf reachability analysis tools for the language of interest could be leveraged to answer this question. We investigate the idea of reducing to C and then leveraging existing tools with respect to three diverse analysis tools for C: a SAT-based model checker, CBMC [CKL04]; a dynamic symbolic execution engine, KLEE [CDE08], and a coverage-guided fuzzer, libFuzzer [Ser22]. Of these, CBMC is a fully symbolic analyser, libFuzzer is a fully dynamic analyser, and KLEE mixes symbolic and dynamic analysis.

In summary, this thesis makes three main contributions:

- an operational model of a heterogeneous CPU/FPGA system alongside a case-study on how our model affects a producer/consumer queue (Chapter 3)
- an axiomatic model of the same system alongside the techniques to validate the models (Chapter 4)
- an experience report on methods that can be used to efficiently simulate operational memory models (Chapter 5)

The work is a compelling demonstration of the combined power of formal techniques and program analysis technology in bringing rigorous to an emerging computing domain.

1.2 Publications

The material presented in this thesis has either been published in conference articles or is currently under submission. Organised by chapter, these publications are as follows:

- The contents of Chapter 3 and Chapter 4 have been previously published in OOPSLA 2021 [IDSW21b], published as a full-length article in the Proceedings of the ACM Programming Languages journal. These chapters include the operational model, the axiomatic model, the technique to validate them and the case study for the producer/consumer queue.
- The contents of Chapter 5 is based on our work that is currently under review at TSE 2022 [IWD]. This includes the techniques we used to simulate operational memory models that allowed us to fix some infidelities in the original model.

In addition to this work, the author of this thesis led the following publication work at RTAS 2020 [ISWD20]. This paper deals with testing multicore systems to assess their suitability for real-time applications.

Chapter 2

Background

This chapter provides a brief introduction to weak memory (Section 2.1) and an introduction to the heterogeneous system that we have modelled (Section 2.2).

2.1 Weak memory

Sequential consistency states that the result of any execution of a multicore system is the same as if the operations of all cores were executed in some sequential order. As a result, the operations of each process appear in this sequence in the order specified by its program. Modern architectures do not implement sequential consistency (SC) as defined by Lamport [Lam79]. Therefore, programmers cannot expect their programs to access memory in the order in which loads and stores appear in their source code without additional synchronisation. Accessing main

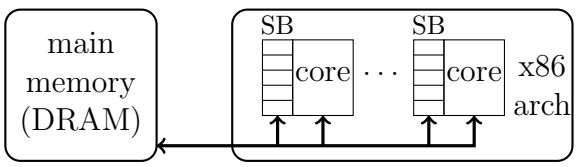


Figure 2.1: A pictorial representation of the x86 memory model. The store buffers (SB) present in each core can cause weak memory behaviours. Each core will first write to its store buffer before committing to main memory.

init: $x = y = 0$	
1 $x \leftarrow 1$	1 $y \leftarrow 1$
2 $r_0 \leftarrow y$	2 $r_1 \leftarrow x$
Allowed: $r_0 = 0$ and $r_1 = 0$	

Figure 2.2: Store buffering using two threads

memory has high latency, and optimisations are required to hide this latency. Optimisations that cause **weak memory** effects are present in all modern architectures [AG96a]. These weak memory effects are only observable when multiple threads access the same data in shared memory, and the reorderings that can result from this are often quite subtle.

2.1.1 Memory models

A **memory model** is an abstract representation of the system that defines the memory-related behaviours a system permits. Memory models allow reasoning about the correctness of multi-threaded programs and are generally used by the compiler to understand what optimisations can be safely applied [Gha95, HKV98]. A memory model stipulates synchronisation operations and when changes to a variable should be made visible to other threads.

As an example, let us consider the operational memory model of the x86 architecture [OSS09], illustrated in Figure 2.1. Each core has its own *store buffer*. When a core issues a write to memory, the write temporarily resides in the store buffer, and all writes in a store buffer are bulk-transferred to main memory periodically. When loading from a location, a core will first check whether a write to that location is pending in its store buffer. If so, it will return the value associated with that write, thus avoiding the expensive operation of reading from the main memory. As a result, each core has a different view of the system's memory: a core may be able to observe the writes that it has issued before other cores can observe them. These different views of main memory can lead to unintuitive behaviours, where cores observe memory operations as having occurred in an order that is not *sequentially consistent*: it does not correspond to any interleaving of instructions executed by individual threads. Programmers can recover sequential consistency with the aid of special *fence* operations, which force store buffers

to be flushed, so that writes become visible to all cores. However, because fence operations are expensive, they should be used sparingly.

The weak memory effects differ from architecture to architecture. Therefore, each architecture has its memory model, and concurrent programs running on them must consider this to ensure correct and predictable execution. A memory model provides an abstraction of the reorderings allowed by the hardware that the compiler should understand so that programs are portable and hide away the complexity of each processor to the programmer.

Memory models usually are more conservative and describe all the potential reordering, even if they do not manifest in all versions of a specific architecture [SSA+11]. While adding unrequired fence operations can degrade performance, it is generally prefered to code that executes incorrectly.

Memory models are specified according to two main paradigms: *operational* and *axiomatic*. An operational model is an abstract representation of the actual machine, described by *states* that represent idealised components such as buffers and queues, and the legal *transitions* between these states. On multicore systems, there may be several available transitions from any given state, and hence one gets an exponential blow-up in the number of paths to explore. On the other hand, an axiomatic model defines relations between memory accesses to constrain the allowed and disallowed behaviours. Simulation of axiomatic models "can be orders of magnitude faster" than the simulation of operational models, but "operational models are often considered more intuitive" [AMT14], and there is a great deal of recent work that relies on operational models [PFD+17a, SFP+20, PPPK+19].

2.1.2 Litmus tests

Litmus tests are small concurrent programs designed to reveal whether a specific memory model behaviour can occur. A litmus test usually comprises a sequence of shared memory write and read operations, followed by an assertion over the values observed by reads.

Specific litmus tests have been designed to characterise particular architectural features that

might give rise to certain weak behaviours. For example, in Figure 2.2 the well-known *store-buffering* litmus test is illustrated. These litmus tests can reveal the write buffers of the x86 memory model seen in Figure 2.1. The test requires both writes to be buffered in order for the CPU cores to observe the old values of variables x and y. If this happens, the old values of the variables will be observed.

Typically a litmus test should be executed many times on a processor of interest to gain confidence as to whether or not the assertions associated with the litmus test hold because concurrent systems with weak memory models are highly non-deterministic. Moreover, it is possible that some weak behaviours only manifest when the system is under heavy stress [AMSS10, SD16, KSTM20, ABD$^+$15].

2.1.3 Simulating memory models

Constructing and using such models is facilitated by **simulators** that reveal which behaviours of a given program are allowed. An accurate memory model should describe the entire set of allowed behaviours on the hardware. Therefore, running code on the memory model should potentially reveal the same behaviours as running on the actual hardware.

However, there can be cases when the reorderings described by the memory model do not manifest on the actual hardware. This behaviour can be due to the conditions required by the reordering to occur only when certain very improbable conditions are encountered or due to the architect of the processor choosing not to implement a feature in a particular iteration of the processor that causes that reordering. As an example, load-buffering has never been observed on PowerPC machines even though theoretically it can occur[SSA$^+$11].

A litmus test can be run on an operational memory model and an axiomatic memory model [AMT14]. The operational and axiomatic models should be equivalent, meaning that both should accurately describe the same behaviours when simulating litmus tests. A memory model takes as input a litmus test describing the sequence of instructions for each concurrent component of the system and a description of a final state of interest. While extensive work has been done

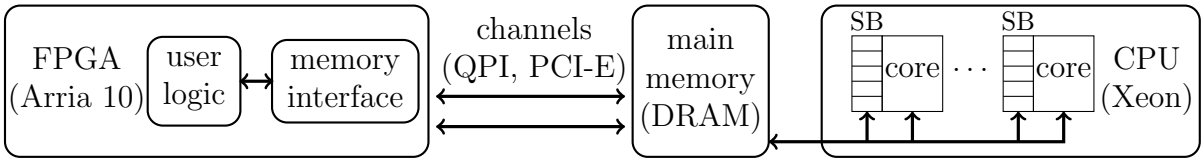


Figure 2.3: Overview of the X+F memory system. The FPGA (left) is composed of user logic and a memory interface. The memory interface uses channels mapped to hardware buses to reach the main memory. The Xeon CPU (right) communicates through coherent caches. Each CPU core contains a store buffer (SB), which allows write/read reorderings.

on simulating *axiomatic* memory models [KV21, AMT14, WBSC17], there has been little work on the simulation of *operational* models [ABC+22] . Operational models are often considered more intuitive but are challenging to simulate due to the vast number of paths through the model's transition system.

An operational memory model will facilitate searching for transitions of the system that might lead from an initial state of the litmus test to the final state of interest. A trace that leads to the final state indicates that this behaviour is *allowed*; if no such trace exists, it is *disallowed*. Nondeterminism arises due to the order in which the concurrent components issue the instructions and due to the internals of the memory system (such as flushing policies for buffers and caches). Once a trace that reaches a state of interest has been found, the programmer or memory model engineer can use the simulator to step through the trace in detail to understand its behaviour better.

An axiomatic memory model will analyse if the axioms describing the system allow for the outcome of the litmus test to be reached from the system's initial state. In contrast to operational models, axiomatic models do not provide a trace through the system that can explain how the final state of the test has been reached. However, these models are generally much faster to simulate.

2.2 The heterogeneous X+F system

We now provide an overview of the X+F memory system, which we depict in Figure 2.3. The system contains a multicore CPU (Xeon) and an FPGA (Altera 10). The FPGA is composed

of programable user logic and a fixed memory interface. The user logic can interact with the main memory via the fixed memory interface. The goal of the memory interface is to provide a layer of abstraction for low-level communication channels. On the other hand the CPU is connected to the main memory via well-know caches.

A typical heterogeneous program starts with the CPU allocating a region of shared memory and then communicating the address and size of that region to the FPGA via dedicated control registers. The FPGA can then be treated as an additional core, accessing the shared memory via read and write requests.

The FPGA thread The user logic on the FPGA does not have any notion of thread. Multiple threads can be implemented within the same logic and run in parallel, but it is up to the user to define how they communicate. Since the user's choices in implementing the multiple threads can have a profound impact on the memory model, we choose to have a single thread on the user logic. This assures us the the resulting memory model will be defined by the available hardware and not choices of the user.

As with any shared-memory system, the behaviour of loads and stores is governed by a *memory model*. Since most modern CPU architectures use a *relaxed* memory model, we can expect the same behaviour from this system. In the case of the X+F system, the behaviour of loads and stores by the CPU and the FPGA is defined by the Intel standard from [Int19]. The fixed logic provides a layer of abstraction for the actual bus interface, facilitating portability. To reason about the X+F system, the distinct ways in which different compute-units access shared memory must be considered: the CPU system has a traditional coherent cache hierarchy, while the FPGA must directly target low-level channels that correspond to hardware buses, as depicted in Figure 2.3. The specification for operations that target these channels is given in official documentation [Int19]. This system's most recent version has three channels: a cache-coherent Quick Path Interconnect (QPI) channel [1] and two Peripheral Component Interconnect Express (PCI-E) channels. By sending read and write requests along these channels, the CPU

[1] While this channel is described as coherent, our initial model treats all channels symmetrically, i.e. non-coherent. The QPI channel is referred to as Ultra Path Interconnect (UPI) in later versions.

init:	x = 0
	1 x ← 1
	2 r0 ← x
allowed?	r0 = 0

(a) Basic test

init:	x = 0
FPGA:	1 ch1: x ← 1
	2 await write resp.
	3 ch1: r0 ← x
disallowed:	r0 = 0

(b) Single-channel synchronisation

init:	x = 0
FPGA:	1 ch1: x ← 1
	2 ch1: fence
	3 await fence resp.
	4 ch2: r0 ← x
disallowed:	r0 = 0

(c) Multi-channel synchronisation

Figure 2.4: Litmus tests for write/read coherence on the FPGA. Programs (b) and (c) show two distinct ways to disallow the non-coherent behaviour described in (a).

and FPGA can concurrently access main memory (DRAM) in a fine-grained manner.

There are three primary sources of relaxed memory behaviours. First, the Xeon CPU implements the x86-TSO memory model [OSS09]. Each core has a store buffer (SB), which may allow writes to be reordered with subsequent reads. Second, memory accesses initiated by the FPGA can be reordered before they are sent to the communication channels. Third, those memory accesses might be sent along different channels with different latencies. This multitude of relaxed behaviours can be attributed to two architectural features of the X+F: (1) writes and reads initiated by the FPGA can be reordered before they are sent to the communication channels, and (2) they might be sent on different channels that have different latencies and drain asynchronously. Furthermore, since the CPU memory is significantly larger than the local memory of the FPGA, applications can effectively utilise it as a shared memory between the CPU and FPGA.

We use examples to illustrate the relaxed nature of the X+F memory model, showing that not even single-address consistency (coherency) is guaranteed for the FPGA (Section 2.2.1), and discussing more complicated CPU/FPGA interactions using standard litmus tests, instantiated for the X+F system, where one thread is on the CPU, and the other is on the FPGA (Section 2.2.2).

2.2.1 FPGA Coherency

The write/read litmus test of Figure 2.4a contains two memory instructions: a write to a location `x` and then a read from `x`. The test asks whether the read can observe the (stale) initial value of 0. A memory interface that allows this behaviour violates *coherence*, which is a property provided by all mainstream shared-memory CPU architectures. However, if the memory instructions are compiled to a sequential FPGA circuit that uses the Core Cache Interface (CCI-P) interface to memory, the behaviour *is* allowed. This is documented [Int19, page 41], and observable in practice: we ran 1M iterations of Figure 2.4a under heavy memory traffic and observed non-coherent behaviour in around 0.1% of them.

One of two CCI-P interface features must be used to disallow this extremely weak behaviour.

Single-Channel Synchronisation First, FPGA-issued memory instructions can specify an explicit *channel* (cf. Figure 2.3). For instance, in Figure 2.4b, instructions 1 and 3 target channel 1, as indicated by "`ch1:`". However, targeting the same channel is not enough to restore coherence: although channels are strictly ordered, CCI-P allows accesses to be re-ordered *before reaching* a channel. Thus, the interface provides *response* events, which can be waited on. For instance, instruction 2 in Figure 2.4b is a write response that indicates that the write to `x` has reached the channel. The read instruction (instruction 3) will then be inserted into the channel *after* the write, disallowing the non-coherent behaviour.

Multi-Channel Synchronisation The other mechanism for ensuring coherence is illustrated in Figure 2.4c, in which the write to and read from `x` do *not* target the same channel. A write response guarantees that the value has been committed to the target channel, but different channels are allowed to flush asynchronously and in any order. Instead, Figure 2.4c uses a fence for synchronisation. Once the fence *response* is observed, all writes must have reached the main memory, so subsequent reads from different channels are guaranteed to observe up-to-date values.

init:	x = y = 0		
FPGA:	1 ch1: y ← 1 2 await write resp. 3 ch1: r0 ← x	CPU:	1 x ← 1 2 fence 3 r1 ← y
disallowed:	r0 = 0 and r1 = 0		

Figure 2.5: Heterogeneous variant of the store buffering (SB) test. The left instruction stream corresponds to an FPGA circuit while the right instruction stream represents a CPU program. Each device needs its own synchronisation variant to disallow the relaxed behaviour.

2.2.2 CPU/FPGA Synchronisation

The previous example showed that sequential streams of shared-memory accesses on the FPGA can allow counterintuitive behaviours. Now, we complicate things further by adding a CPU thread. This is a challenge because the FPGA and the CPU implement distinct memory models and require different types of synchronisation depending on the desired orderings.

Store Buffering We begin with the classic store buffering (SB) test. The heterogeneous variant is shown in Figure 2.5: an FPGA instruction stream is shown on the left and a CPU stream on the right. The FPGA stream has its specific synchronisation constructs: channel annotations and response-waiting. On the other hand, the CPU stream resembles standard tests in the literature, i.e. without channels and using traditional CPU fences.

In order to disallow the SB weak behaviour, the write/read ordering between instructions 1 and 3 on both the CPU and the FPGA must be enforced. On the CPU side of the X+F system, we must reason about the TSO memory model. Recall from Figure 2.3 that each CPU thread contains a store buffer, which can allow reads to overtake writes at run-time. To disallow this, we can place a write/read fence (e.g. `MFENCE` in x86) to flush the store buffer. The FPGA stream must also enforce ordering and, as in Section 2.2.1, there are two ways to do this, depending on whether memory instructions target the same or different channels. In this example, we show the single-channel variant, where the memory operations can be ordered simply by waiting for the write response before issuing the read instruction. Without this wait, there is no guarantee that the write and read will be inserted into the channel in the issue order. Recall that the write response guarantees that the write operation has been inserted into the channel.

init:	x = y = 0	
FPGA:	1 ch1: x ← 1 2 all: write fence 3 ch2: y ← 1	CPU: 1 r0 ← y 2 r1 ← x
disallowed:	r0 = 1 and r1 = 0	

(a) FPGA producer, CPU consumer

init:	x = y = 0	
FPGA:	1 ch1: r0 ← y 2 await read resp. 3 ch2: r1 ← x	CPU: 1 x ← 1 2 y ← 1
disallowed:	r0 = 1 and r1 = 0	

(b) CPU producer, FPGA consumer

Figure 2.6: A heterogeneous message passing (MP) test. A producer writes a value (in x) and then a ready-flag (in y). The query asks if a consumer is allowed to observe a positive ready-flag but then read stale data.

Message Passing Now we move on to some heterogeneous variants of the classic message-passing (MP) litmus test, shown in Figure 2.6. In this test, one instruction stream (the producer) attempts to communicate a data value to another instruction stream (the consumer). Because streams execute asynchronously and in parallel, an auxiliary ready-flag message must be sent. The test asks whether the consumer can read a positive, ready flag but still observe stale data. On the producer side, ensuring that stale data cannot be observed requires ordering the two writes, i.e. the write of the data followed by the write of the ready flag. On the consumer, the two reads must be ordered, i.e. the read of the ready flag followed by the reading of the data. Unlike the SB test, the MP test is asymmetric: different types of synchronisation are required depending on the role of the instruction stream (producer or consumer) and the device.

Figure 2.6a shows a variant of the test where the FPGA is the consumer. The data is written to one channel (ch1 in the example), and synchronisation is achieved through another channel (ch2 in the example). This may happen, e.g. if the synchronisation is implemented in a library that does not constrain the client's channels. To prevent the writes from being reordered, a fence that synchronises across all channels is required [Int19, page 41]. The CPU side of synchronisation is much simpler: TSO preserves read/read order, so no additional instructions

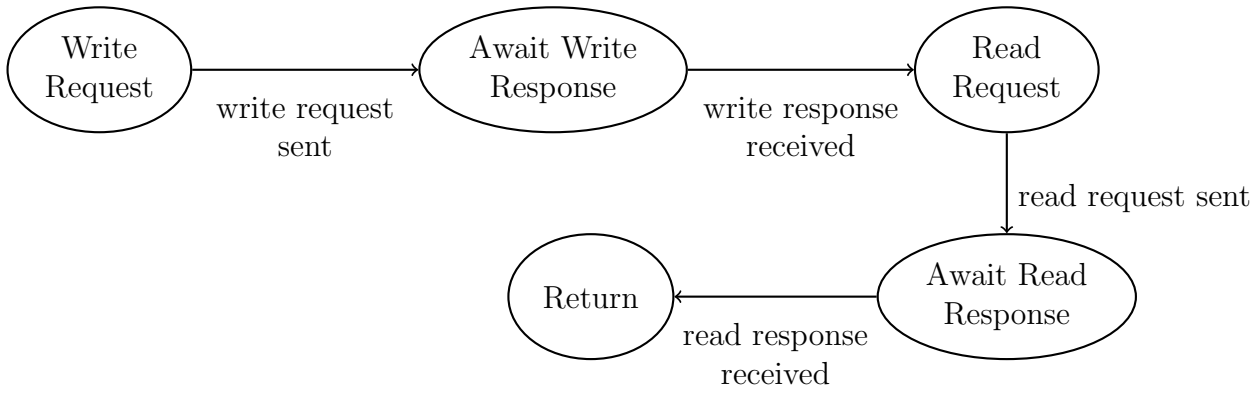


Figure 2.7: A state machine corresponding to the litmus test in Figure 2.4b. The FPGA will only exit the Write/Read Request states once it has managed to send the corresponding request. It will only exit the Await Write/Read Response once the corresponding response has been received.

.

are required. (A weaker CPU architecture, such as PowerPC or Arm, would require different reasoning.)

Figure 2.6b shows another variant of the test, where the FPGA is the consumer this time. Unlike the CPU consumer, the FPGA does allow read/read reorderings if synchronisation is not used. In this case, the FPGA needs to wait for the read response, which means that the read has been satisfied; no further synchronisation is required [Int19, page 41]. The CPU side does not require additional synchronisation because write/write order is preserved in TSO.

2.2.3 Implementing Litmus Tests on the FPGA

While the CPU threads in a litmus test are executed in a conventional instruction-by-instruction fashion, the FPGA 'thread' of a litmus test is handled differently: it is compiled to a sequential circuit implemented on the FPGA. The circuit takes the form of a state machine. As an example, Figure 2.7 illustrates the state machine corresponding to the litmus test in Figure 2.4b. The FPGA remains in the first state while it issues the write request. It then remains in the second state, constantly monitoring the memory interface until it receives the corresponding response. The next two states perform the read request/response similarly, after which the test is finished.

The FPGA will only exit a request state once it has sent the corresponding request. Likewise, the FPGA will only exit a response state once it has received a response. Of course, a response will only be issued after it has been requested, but the FPGA does not guarantee when it will be received. Therefore, if the user wants to observe a response, it will await it immediately after the request has been issued.

The amount of time spent in any state is not known. While in a request state, the interface can be occupied for an unknown number of clock cycles, forcing the FPGA to wait until this is no longer the case. Correspondingly, the FPGA will constantly monitor the interface in a response state until a response has been received.

As another example, the litmus test from Figure 2.4a can be obtained by removing the "Await Write Response" state from Figure 2.7. Likewise, the litmus test of Figure 2.4c can be obtained by replacing the "Await Write Response" state with a "Fence Request" state followed by an "Await Fence Response" state.

Chapter 3

Operational model

As the examples in Section 2.2 show, the low-level shared-memory interface on the X+F system is complex and nuanced. We could only determine these tests' outcomes through careful documentation-reading, discussion with the X+F engineers, and empirical testing.

We thus present an operational memory model (Section 3.1), describing the *actions* that the CPU and FPGA can take, the system *states* and the possible *transitions* between these states. Having presented the model, we then justify our key modelling decisions concerning available documentation about the X+F system (Section 3.2) and present our implementation of the model (Section 3.3). This model is also the basis for reasoning about the memory model idioms that underpin efficient implementations of synchronisation constructs (Section 3.4). We then present related works (Section 3.5) and conclude with a summary of our findings (Section 3.6).

3.1 A formalisation of the memory model

We now present a formal operational semantics for this heterogeneous shared-memory interface that faithfully accounts for these behaviours. We begin by describing the *actions* that the FPGA and the CPU can use to interact with the memory system (Section 3.1.1). We then describe the set of *states* in which the memory system can reside (Section 3.1.2), and the set of

transitions between states that the memory system can make in response to the FPGA's and CPU's actions (Section 3.1.3).

3.1.1 Actions

We model the CPU's and the FPGA's interactions with the memory system using *actions*. On the CPU, an action represents the execution of a memory-related instruction. On the FPGA, an action represents a request sent to the memory system or a response received from it. There are four types of requests that the FPGA can make to the memory system:

- `WrReq`(c, l, v, m) is a request to write value v to location l along channel c. The request is tagged with metadata m so that it can later be associated with its corresponding response.
- `RdReq`(c, l, m) is a request to read from location l along channel c, with metadata m as above.
- `FnReqOne`(c, m) is a request to perform a fence on channel c, with metadata m as above.
- `FnReqAll`(m) is a request to perform a fence on *all* channels, with metadata m as above.

There are four actions that can be received by the FPGA from the memory system:

- `WrRsp`(c, m) represents a response from the memory system indicating that an earlier write request with metadata m has entered its channel (though the write may not yet have propagated all the way to the main memory). The FIFO property of the channel ensures that all subsequent writes to the same channel will not get reordered. The c field is required because the original request might not have specified a channel.
- `RdRsp`(c, v, m) represents a response from the memory system containing, in v, the value requested by an earlier read request on channel c with metadata m.
- `FnRspOne`(c, m) represents a response from the memory system that a fence requested on channel c with metadata m has finished and that all writes on that channel requested prior to this fence have reached main memory.

- `FnRspAll`(m) represents a response from the memory system that a requested all-channel fence with metadata m has finished and that all writes requested prior to this fence (on any channel) have reached main memory.

Finally, there are three actions by which the CPU can interact with the memory system, each parameterised by thread-identifier t:

- `CPUWrite`(t, l, v) represents the execution of an instruction that writes value v to location l.
- `CPURead`(t, l, v) represents the execution of an instruction that reads value v from location l.
- `CPUFence`(t) represents the execution of a fence instruction.

3.1.2 States

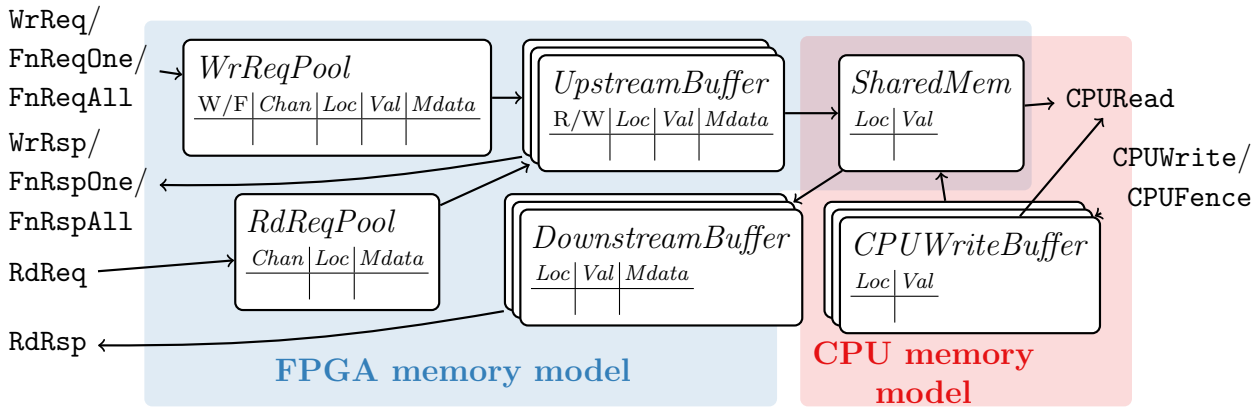
Figure 3.1 describes the states in which the system can reside.

Working through the definitions in Figure 3.1a, we see the locations, values, metadata tags, and channels that we have encountered already. We use the notation $X_\perp$ for the set X extended with an additional $\perp$ element, which represents a blank.

Read requests from the FPGA enter the system via the *read-request pool*, which is a list of records, each of which contains the channel that the request is to be sent along, the location to be read, and the metadata to identify the request. The *write-request pool* is similar, but since it can hold both write requests and fence requests, each record is additionally tagged as W or F. Fence requests leave the location and value fields blank, and an all-channel fence request also leaves the channel field blank. Requests reside in a pool before migrating to a *channel* (discussed next), and this is where some reordering is possible: migration from pool to channel is not always first-in-first-out.

$$
\begin{array}{rrcl}
l \in & \mathit{Loc} & \stackrel{\text{def}}{=} & \mathbb{N} \\
t \in & \mathit{Tid} & \stackrel{\text{def}}{=} & \{0, \ldots, T-1\} \\
v \in & \mathit{Val} & \stackrel{\text{def}}{=} & \mathbb{Z} \\
c \in & \mathit{Chan} & \stackrel{\text{def}}{=} & \{0, \ldots, N-1\} \\
m \in & \mathit{Mdata} & \stackrel{\text{def}}{=} & \mathbb{Z} \\
\mathit{RP} \in & \mathit{RdReqPool} & \stackrel{\text{def}}{=} & (\mathit{Chan} \times \mathit{Loc} \times \mathit{Mdata})\,\text{list} \\
\mathit{WP} \in & \mathit{WrReqPool} & \stackrel{\text{def}}{=} & (\{\mathrm{W}, \mathrm{F}\} \times \mathit{Chan}_\perp \times \mathit{Loc}_\perp \times \mathit{Val}_\perp \times \mathit{Mdata})\,\text{list} \\
 & \mathit{UpstreamBuffer} & \stackrel{\text{def}}{=} & (\{\mathrm{R}, \mathrm{W}\} \times \mathit{Loc} \times \mathit{Val}_\perp \times \mathit{Mdata})\,\text{list} \\
\mathit{UB} \in & \mathit{UpstreamBuffers} & \stackrel{\text{def}}{=} & \mathit{Chan} \to \mathit{UpstreamBuffer} \\
 & \mathit{DownstreamBuffer} & \stackrel{\text{def}}{=} & (\mathit{Loc} \times \mathit{Val} \times \mathit{Mdata})\,\text{list} \\
\mathit{DB} \in & \mathit{DownstreamBuffers} & \stackrel{\text{def}}{=} & \mathit{Chan} \to \mathit{DownstreamBuffer} \\
 & \mathit{FPGAState} & \stackrel{\text{def}}{=} & \mathit{WrReqPool} \times \mathit{RdReqPool} \times \mathit{UpstreamBuffers} \times \mathit{DownstreamBuffers} \\
 & \mathit{CPUWriteBuffer} & \stackrel{\text{def}}{=} & (\mathit{Loc} \times \mathit{Val})\,\text{list} \\
\mathit{WB} \in & \mathit{CPUWriteBuffers} & \stackrel{\text{def}}{=} & \mathit{Tid} \to \mathit{CPUWriteBuffer} \\
 & \mathit{CPUState} & \stackrel{\text{def}}{=} & \mathit{CPUWriteBuffers} \\
\mathit{SM} \in & \mathit{SharedMem} & \stackrel{\text{def}}{=} & \mathit{Loc} \rightharpoonup \mathit{Val} \\
 & \mathit{SyState} & \stackrel{\text{def}}{=} & \mathit{FPGAState} \times \mathit{SharedMem} \times \mathit{CPUState}
\end{array}
$$

(a) Formal definitions



(b) The system state, pictorially

Figure 3.1: The memory system state that combines the FPGA view and the CPU view

The model contains N channels that link the FPGA to the shared memory. (For the current version of X+F, N is 3, and in an earlier version, N was 1 [CCF+19]. Our model applies to any value of N.) Each channel is split into an *upstream buffer* heading towards shared memory and a *downstream buffer* heading towards the FPGA. Each upstream buffer is modelled as a list of records, each of which is tagged as being a read request (R) or a write request (W). Read requests leave the value field blank. Downstream buffers only hold read responses, so do not need tagging. The upstream and downstream buffers are both FIFO and no reordering can happen within them. Fences are not sent to the upstream buffer; rather, they guard entry to the upstream buffer.

The FPGA's view of the memory system thus consists of the write- and read-request pools, a set of upstream buffers and a set of downstream buffers, together with a shared memory that maps locations to values. Meanwhile, the CPU's view of the memory system consists of the same shared memory together with a write buffer per core, each holding writes destined for shared memory. The overall memory system that combines the FPGA view and the CPU view is depicted in Figure 3.1b.

3.1.3 Transitions

The state of the system can evolve by the CPU or FPGA performing one of the actions listed in Section 3.1.1 or by an internal action of the memory system. We write

$$\text{old state} \xrightarrow{a} \text{new state}$$

to denote a state transition that coincides with the sending or receiving of action a. Internal actions are labelled with τ. We divide transitions into those that only affect the FPGA's view of the memory system (Figure 3.2) and those that only affect the CPU's view (Figure 3.3). These two sets of transitions can be combined (Figure 3.4) to describe the evolution of the entire system state.

The FPGA-related transitions defined in Figure 3.2 can be understood as follows.

Write Request

$$\frac{}{(WP, RP, UB, DB, SM) \xrightarrow[\text{FPGA}]{\texttt{WrReq}(c,l,v,m)} (WP +\!\!+ (\mathrm{W}, c, l, v, m), RP, UB, DB, SM)}$$

Read Request

$$\frac{}{(WP, RP, UB, DB, SM) \xrightarrow[\text{FPGA}]{\texttt{RdReq}(c,l,m)} (WP, RP +\!\!+ (\mathrm{R}, c, l, m), UB, DB, SM)}$$

Fence Request One Channel

$$\frac{}{(WP, RP, UB, DB, SM) \xrightarrow[\text{FPGA}]{\texttt{FnReqOne}(c,m)} (WP +\!\!+ (\mathrm{F}, c, \bot, \bot, m), RP, UB, DB, SM)}$$

Fence Request All Channels

$$\frac{}{(WP, RP, UB, DB, SM) \xrightarrow[\text{FPGA}]{\texttt{FnReqAll}(m)} (WP +\!\!+ (\mathrm{F}, \bot, \bot, \bot, m), RP, UB, DB, SM)}$$

Flush Write Request to Upstream Buffer

$$\frac{WP = head +\!\!+ (\mathrm{W}, c, l, v, m) +\!\!+ tail \qquad (\mathrm{F}, c, \_, \_, \_) \notin head \qquad (\mathrm{F}, \bot, \_, \_, \_) \notin head}{(WP, RP, UB, DB, SM) \xrightarrow[\text{FPGA}]{\texttt{WrRsp}(c,m)} (head +\!\!+ tail, RP, UB[c := UB[c] +\!\!+ (\mathrm{W}, l, v, m)], DB, SM)}$$

Write to Memory

$$\frac{UB[c] = (\mathrm{W}, l, v, m) +\!\!+ tail}{(WP, RP, UB, DB, SM) \xrightarrow[\text{FPGA}]{\tau} (WP, RP, UB[c := tail], DB, SM[l := v])}$$

Fence Response One Channel

$$\frac{WP = (\mathrm{F}, c, \bot, \bot, m) +\!\!+ tail \qquad UB[c] = \emptyset}{(WP, RP, UB, DB, SM) \xrightarrow[\text{FPGA}]{\texttt{FnRspOne}(c,m)} (tail, RP, UB, DB, SM)}$$

Fence Response All Channels

$$\frac{WP = (\mathrm{F}, \bot, \bot, \bot, m) +\!\!+ tail \qquad \forall c \in Chan.\ UB[c] = \emptyset}{(WP, RP, UB, DB, SM) \xrightarrow[\text{FPGA}]{\texttt{FnRspAll}(m)} (tail, RP, UB, DB, SM)}$$

Flush Read Request to Upstream Buffer

$$\frac{RP = head +\!\!+ (\mathrm{R}, c, l, m) +\!\!+ tail}{(WP, RP, UB, DB, SM) \xrightarrow[\text{FPGA}]{\tau} (WP, head +\!\!+ tail, UB[c := UB[c] +\!\!+ (\mathrm{R}, l, m)], DB, SM)}$$

Read from Memory

$$\frac{UB[c] = (\mathrm{R}, l, m) +\!\!+ tail \qquad SM(l) = v}{(WP, RP, UB, DB, SM) \xrightarrow[\text{FPGA}]{\tau} (WP, RP, UB[c := tail], DB[c := DB[c] +\!\!+ (l, v, m)], SM)}$$

Read Response

$$\frac{DB[c] = (l, v, m) +\!\!+ tail}{(WP, RP, UB, DB, SM) \xrightarrow[\text{FPGA}]{\texttt{RdRsp}(c,l,v,m)} (WP, RP, UB, DB[c := tail], SM)}$$

Figure 3.2: Operational semantics, FPGA side.

CPU Write

$$\frac{}{(SM,\, WB) \xrightarrow[\text{CPU}]{\texttt{CPUWrite}(t,l,v)} (SM,\, WB[t := WB[t] +\!\!+ (l,v)])}$$

CPU Flush Write Buffer to Memory

$$\frac{WB[t] = (l,v) +\!\!+ tail}{(SM,\, WB) \xrightarrow[\text{CPU}]{\tau} (SM[l := v],\, WB[t := tail])}$$

CPU Fence

$$\frac{WB[t] = \emptyset}{(SM,\, WB) \xrightarrow[\text{CPU}]{\texttt{CPUFence}(t)} (SM,\, WB)}$$

CPU Read from Memory

$$\frac{SM(l) = v \qquad (l, \_) \notin WB[t]}{(SM,\, WB) \xrightarrow[\text{CPU}]{\texttt{CPURead}(t,l,v)} (SM,\, WB)}$$

CPU Read from Write Buffer

$$\frac{WB[t] = head +\!\!+ (l,v) +\!\!+ tail \qquad (l, \_) \notin tail}{(SM,\, WB) \xrightarrow[\text{CPU}]{\texttt{CPURead}(t,l,v)} (SM,\, WB)}$$

Figure 3.3: Operational semantics, CPU side (following [OSS09])

FPGA Step

$$\frac{(WP, RP, UB, DB, SM) \xrightarrow[\text{FPGA}]{a} (WP', RP', UB', DB', SM')}{(WP, RP, UB, DB, SM, WB) \xrightarrow{a} (WP', RP', UB', DB', SM', WB)}$$

CPU Step

$$\frac{(SM,\, WB) \xrightarrow[\text{CPU}]{a} (SM',\, WB')}{(WP, RP, UB, DB, SM,\, WB) \xrightarrow{a} (WP, RP, UB, DB, SM',\, WB')}$$

Figure 3.4: Operational semantics, FPGA and CPU combined

Write Request adds a new write entry to the write-request pool, and **Read Request** adds a new read entry to the read-request pool. **Fence Request One Channel** adds a new fence entry with a specified channel to the write-request pool, while **Fence Request All Channels** adds a new fence entry on all channels. **Flush Write Request to Upstream Buffer** says that if the write-request pool contains a write entry and there are no older fences on the same channel (or on all channels) in the pool, then the write entry can be removed and appended to the corresponding upstream buffer, issuing a write response back to the FPGA. Note that this rule allows writes in the pool to overtake one another if they are not separated by fences. **Write to Memory** says that a write entry can be removed from the head of an upstream buffer, whereupon shared memory is updated. **Fence Response One Channel** removes a single-channel fence from the head of the write-request pool providing the named channel is empty, issuing a fence response back to the FPGA. **Fence Response All Channels** similarly removes an *all-channel* fence providing *all* channels are empty. **Flush Read Request to Upstream Buffer** removes any read entry from the read-request pool and adds it to the tail of the corresponding upstream buffer. **Read from Memory** removes a read entry from the head of an upstream buffer and updates the corresponding downstream buffer with a new entry containing the value found in the shared memory. **Read Response** removes an entry from the head of a downstream buffer, issuing a corresponding read response to the FPGA.

Figure 3.3 defines the following CPU-related transitions. **CPU Write** adds an entry to the tail of the write buffer. **CPU Fence** blocks the CPU from completing a fence action until the write buffer is empty. **CPU Flush Write Buffer to Memory** removes the entry at the head of the write buffer and updates the shared memory with the corresponding value. **CPU Read from Memory** reads from shared memory the value of a location that is not in the write buffer, while **CPU Read from Write Buffer** reads the latest value of a location that is in the write buffer.

Finally, Figure 3.4 presents the **FPGA Step** and **CPU Step** rules, which describe how the overall system can evolve as a result of a step either on the FPGA side or on the CPU side.

3.2 Justifications for modelling decisions

The model presented above was informed by studying the CCI-P manual [Int19]. We now describe how the various modelling decisions we made – such as the use of read- and write-request pools, and the inclusion of both upstream and downstream buffers – are justified by reference to the text from the manual.

Read- and Write-request Pools Even though they are not explicitly mentioned in the manual, our model contains read- and write-request pools. These are motivated by the system behaviour when a single channel is used for reads and writes.

> "Memory may see two writes to the same [channel] in a different order from their execution, unless the second write request was generated after the first write response was received."
>
> [Int19, page 40]

> "Reads to the same [channel] may complete out of order; the last read response always returns the most recent data." [Int19, page 42]

This indicates the presence of a staging area before the writes are committed to their corresponding channel. Thus we deduce that this staging area is responsible for reordering write requests with other write requests and read requests with other read requests. From this, it is unclear whether there are separate pools for read and write requests, but since CCI-P exposes different interfaces for checking whether the memory system can accept read and write requests, we keep these staging areas separate. We think this also makes the model easier to read. There would be no semantic difference if they were combined since fences do not affect reads.

Upstream Buffers Once the read and write requests leave their corresponding pools it might be tempting to think that they arrive in the shared memory. However, these requests must first travel through the channels before reaching the shared memory.

> "A memory write response does NOT mean the data are globally observable across channels. A subsequent read on a different channel may return old data and a subsequent write on a different channel may retire ahead of the original write." [Int19, page 39]

No reordering within upstream buffers Once requests arrive in the upstream buffer, reordering can no longer occur.

> "All future writes on the same physical channel replace the data." [Int19, page 38]

The previous observations indicate that requests can only be reordered while they are in their corresponding pools and not once they have reached an upstream buffer. Responses are sent to the FPGA user logic once these have reached an upstream buffer, and it is only then that users have a guarantee of some order.

Downstream Buffers Figure 3.5 shows an allowed execution, adapted from the CCI-P manual [Int19, Table 37 on page 41], that motivated us to introduce the downstream buffers. A CPU core performs two write actions on the location `x` where the FPGA performs two read requests on `x` using two different channels (`ch1` and `ch2`). The first FPGA read observes the first value written by the CPU, while the second FPGA read observes the second value. This indicates that reads have reached the shared memory in program order. However, the responses arrive back to the FPGA in reverse order. This indicates that reordering occurred after the reads reached the shared memory. In our model, this reordering is performed in the downstream buffers.

Fences The **Flush Write Request to Upstream Buffer** rule is somewhat unclear since the manual does not clearly specify if writes can be prevented from being flushed to the upstream buffer by an older fence on *any* channel or the specific channel for which the write is destined. The CCI-P manual states that a fence

init: $\texttt{x} = 0$

CPU:	FPGA:
1 $\texttt{CPUWrite}(\texttt{x}, 1)$	1 $\texttt{RdReq}(\texttt{ch1}, \texttt{x}, \texttt{m1})$
2 $\texttt{CPUWrite}(\texttt{x}, 2)$	2 $\texttt{RdReq}(\texttt{ch2}, \texttt{x}, \texttt{m2})$
	3 $\texttt{RdRsp}(\texttt{ch2}, \textit{v1}, \texttt{m2})$
	4 $\texttt{RdRsp}(\texttt{ch1}, \textit{v2}, \texttt{m1})$

Allowed: $v1 = 2$ and $v2 = 1$

Figure 3.5: A litmus test that motivates downstream buffers

init: $\texttt{x} = \texttt{y} = 0$

CPU:	FPGA:
1 $\texttt{CPURead}(\texttt{y}, \textit{v1})$	1 $\texttt{WrReq}(\texttt{ch1}, \texttt{x}, 1, \texttt{m1})$
2 $\texttt{CPURead}(\texttt{x}, \textit{v2})$	2 $\texttt{WrRsp}(\texttt{ch1}, \texttt{m1})$
	3 $\texttt{FnReqOne}(\texttt{ch2}, \texttt{m2})$
	4 $\texttt{FnRspOne}(\texttt{ch2}, \texttt{m2})$
	5 $\texttt{WrReq}(\texttt{ch1}, \texttt{y}, 1, \texttt{m3})$
	6 $\texttt{WrRsp}(\texttt{ch1}, \texttt{m3})$

Allowed: $v1 = 1$ and $v2 = 0$

Figure 3.6: A litmus test to check our understanding of **Flush Write Request to Upstream Buffer**

> "guarantees that all [...] writes preceding the fence are committed to memory before any writes following [the fence] are processed." [Int19, page 39]

It is unclear whether the channel of a fence matters here. We know that the channel of a fence affects which writes are flushed to memory, as captured in the **Fence Response One Channel** rule; the question is whether single-channel fences impose ordering on accesses to different channels. The question is captured by the Message Passing (MP)-style litmus test in Figure 3.6, in which the FPGA writes to `x` then `y` over channel `ch1`, and between the writes is a fence on channel `ch2`. Without the fence, the weak outcome was observable on hardware, but enabling the fence prevented it, even with stress testing. Nevertheless, we choose to err on the side of caution and enforce ordering between writes only when the fence specifies the same channel as the writes.

3.3 CBMC Implementation and Litmus Tests

We have implemented our model in CBMC [CKL04] and validated it with all of the 17 traces (5 explicitly written out and 12 described in prose) that are given as examples in the CCI-P manual [Int19].

Our CBMC implementation comprises a C file that contains code corresponding to every rule by which the X+F system can take a step according to the semantics in Figures 3.2 and 3.3. The

Table 3.1: Simulation bounds for our CBMC executable model.

Parameter	Value
Number of simulation steps	30
Number of CPU threads	2
Size of upstream buffers	2
Size of downstream buffers	2
Size of write-request pool	4
Size of read-request pool	4
Size of CPU write buffer	2

state of the system is held in several variables and arrays. The premises of the operational rules are implemented using `assume` statements; any rule whose premises are met can be chosen non-deterministically. The effect of the rule is achieved by imperatively updating the state variables and arrays. Where non-deterministic choice is required by the semantic rules, corresponding features for requesting non-deterministic values in CBMC are used, combined with `assume` statements to limit the scope of non-determinism to an appropriate range.

Reasoning about a litmus test scenario with the CBMC implementation involves combining the model C file with a test-specific harness. This harness uses `assume` statements to encode the initial state of the trace and the sequence of actions that each of its threads must take. An `assert` statement is then used to check whether a particular final state is observable.

CBMC is invoked on the combined model and test harness by pointing the tool at the entry point for the test harness and specifying a suitable *loop-unwinding depth*. The tool symbolically unwinds the program up to this depth, producing a SAT formula that is satisfiable if and only if the unwound program contains an instance of the assertion associated with the test that can be violated; that is, all valid paths up to the unwinding depth are explored. In this case, the associated satisfying assignment provides a concrete trace witnessing the assertion violation. Furthermore, assertions are used to sanity check the correctness of the model. For example, we check that buffers are indeed empty after a fence finishes executing.

Simulation Bounds Naturally, the guarantees provided by this kind of simulation are limited by its bounded nature. Table 3.1 shows the parameters that must be bounded, together with the

specific bounds we have used when experimenting with our model. The total number of CPU threads has been chosen such that interesting weak behaviours are exhibited but simulation times are kept reasonable. Another important parameter is the total number of simulation steps, which describes the number of times each component has an opportunity to take a step. We empirically determined whether this value was high enough via a "smoke test" where we temporarily included `assert(false)` at certain points that should be reachable – if the assertion failed then we could be assured that the machine had taken enough steps to reach the program point. Furthermore, we invoked CBMC with the `--unwinding-assertions` option, whereby it checks that unwinding the program further does not lead to any more states being explored. In this mode, CBMC can *prove* that the program under test is free from assertion failures: if an insufficiently large unwinding depth for loops is used then an "unwinding assertion" fails, indicating that a higher bound is required for the proof to succeed. We used scripts to run CBMC repeatedly until sufficient unwinding bounds are found.

Since we do not have access to all of the microarchitectural features of the system, we cannot know the maximum number of requests that can be stuck in transit in the request pools, channels, and buffers. We have chosen numbers that are high enough to exhibit all the weak behaviours described in the manual but small enough for the SAT formulas generated by the bounded model checker to be solvable in a reasonable amount of time.

Confidence in the Model and Test Encoding We acknowledge that the trustworthiness of our findings using our CBMC-based implementation are subject to coding mistakes on our part in writing the model, amplified by the unsafe nature of the C programming language.

We chose CBMC because it is a widely used, robust and practical tool well suited for the system-level modelling work associated with X+F. It was straightforward for us to implement the model and associated tests using only very basic C features: integer variables and arrays. No dynamic memory allocation/deallocation or non-trivial use of pointers and pointer arithmetic was required. Furthermore, CBMC checks for many common sorts of undefined behaviour, including out-of-bounds array accesses, as a matter of course, so our implementation

is guaranteed to be free from these (assuming CBMC works correctly).

To guard against implementation mistakes we made heavy use of assertions in the code associated with our model during its development, e.g. to check various invariants of the X+F system state. We also employed smoke testing in our litmus tests, confirming that each test always failed when its post-condition was replaced with `false`. We deemed this important due to our heavy use of `assume` statements in our modelling effort: careless use of multiple `assume` statements can easily lead to `false` being assumed, in turn leading to vacuous reasoning; our smoke tests demonstrated that our implementation is not subject to these errors.

Michael Adler, a senior Principal Engineer at Intel involved in the design of the X+F system, with whom we communicated frequently during this research, described our operational model as 'definitive' in an email, saying that they would point engineers to it as a reference.

3.4 Case study: reasoning about a producer/consumer queue

A common pattern in heterogeneous CPU/FPGA systems is to deploy a kernel on the FPGA and then stream work to it for processing. For example, one work [WHN19] employed such queues to implement graph algorithms (Dijkstra's single-source shortest path and breadth-first search) on the X+F system. These graph algorithms work by having the CPU decide the order in which the CPU nodes are going to be processed while the FPGA is responsible for the actual node processing. This algorithm takes advantage of the CPU's ability to effectively process highly sequential code and the FPGAs ability to process multiple nodes in a parallel manner.

Because the CPU/FPGA shared memory interface does not support atomic operations, the data structure used for synchronisation must not depend on them. A single-writer, single-reader producer/consumer queue can be used to achieve CPU/FPGA communication without requiring complex synchronisation primitives. The CPU can use such a queue to send work to the FPGA. The FPGA can either write its results to a designated area of shared memory, or

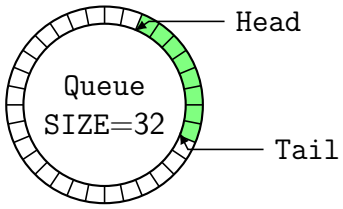


Figure 3.7: A producer/consumer queue

```
1 Head ← *HeadAddr
2 if ((Tail+1) % SIZE == Head) goto 1
3 Queue[Tail] ← new_value
4 *TailAddr ← (Tail+1) % SIZE
```

Listing 3.1: Enqueue

send them back to the CPU via a second queue.

We present a case study demonstrating that our formalisation of the X+F memory model allows rigorous reasoning about the memory operations that are required to correctly implement producer/consumer queues between the CPU and the FPGA. We present two ways to implement queues correctly and evaluate their relative performance. We also investigate what happens when *insufficient* synchronisation instructions are used: our memory model predicts that this should lead to a queue that loses messages or gets them the wrong order, and indeed we observe this behaviour on the hardware in practice.

We now describe how to implement a producer/consumer queue abstractly (Section 3.4.1), and various approaches to implementing it concretely for the X+F system, showing that the CBMC-based implementation of our formal model can help with reasoning about the correctness of the idioms that underpin this implementation. We then present performance results comparing the various X+F implementation options (Section 3.4.2), and study the performance/quality trade-off associated with various *incorrect* but – thanks to our model – still well-defined

```
1 Tail ← *TailAddr
2 if (Head == Tail) goto 2
3 ret_value ← Queue[Head]
4 *HeadAddr ← (Head+1) % SIZE
```

Listing 3.2: Dequeue

implementations (Section 3.4.3).

3.4.1 Implementation

A producer/consumer queue can be implemented using a circular array, as shown in Figure 3.7. The `SIZE` of the array limits the total number of elements that can be added.

Listings 3.1 and 3.2 present the pseudocode of the enqueue and dequeue operations. The producer (resp. consumer) must ensure that the queue is not full (resp. empty) before adding (resp. dequeuing) an element. To achieve this, a simple lock-free implementation will continuously read `Head` (or `Tail`), spinning until it can operate on the queue. With a single producer and consumer, only the producer updates `Tail` and only the consumer updates `Head`. Two important synchronisation behaviours must be preserved between the producer and the consumer. First, `Head` (or `Tail`) must be read (line 1) before accessing the queue (line 3). Second, they must only update their respective pointers (line 4) after accessing the queue (line 3) to ensure correct message-passing behaviour.

X+F Implementations Recall that there are two possible ways of synchronising between the CPU and FPGA on the X+F system: (1) using a single channel and waiting for responses or (2) using multiple channels and issuing fences. Therefore, we implemented two variants of the producer/consumer queue, using C++ for the CPU and Verilog for the FPGA, with associated litmus tests validated using CBMC as described above. The **single channel** variant uses a single channel for communication and waits for write responses whenever reordering could lead to incorrect behaviour. The **multiple channel** variant allows writes and reads to choose any available channel, which may lead to better performance under heavy traffic, but does require fences. In both cases the queue is designed to store 64-bit integer elements.

Listing 3.3 shows a single channel snippet of the code that enqueues `new_value` at the `Tail` of the queue. After writing the value to the queue (line 4), we wait for the corresponding response (line 5) to ensure that the `Tail` does not get updated before the queue has the value. Since we

```
1 RdReq(ch1, HeadAddr, mdata0)
2 RdRsp(Head, mdata0)
3 if ((Tail + 1) % SIZE == Head) goto 1
4 WrReq(ch1, Queue + Tail, new_value, mdata1)
5 WrRsp(mdata1)
6 WrReq(ch1,TailAddr, Head + 1, mdata2)
```

Listing 3.3: Enqueue (single channel)

```
1 RdReq(ch1, TailAddr, mdata0)
2 RdRsp(Tail, mdata0)
3 if (Tail == Head) goto 1
4 RdReq(ch1, Queue + Head, mdata1)
5 ret_value ← RdRsp(mdata1)
6 WrReq(ch1, HeadAddr, Head + 1, mdata2)
7 WrRsp(mdata2)
```

Listing 3.4: Dequeue (single channel)

do not get incorrect behaviour if the `Tail` write request (line 6) gets reordered with the next queue update, we do not have to wait for its response.

Listing 3.4 shows a single-channel snippet of the code that dequeues from the `Head`. When the `Head` of the queue is updated (line 7), we need to wait for a response so that the `Head` does not get reordered with the next `Head` update.

Listing 3.5 shows a multiple channel snippet of the code that enqueues value `new_value` at the `Tail` of the queue. Notice that instead of waiting for a response from the first write request (line 4), a fence operation on all channels is inserted (line 5). There is no need to wait for the response of the fence operation since we are only interested in providing a order between the two write operations. The response of the fence operation would have provided the additional

```
1 RdReq(⊥, HeadAddr, mdata0)
2 Tail ← RdRsp(mdata0)
3 if ((Tail + 1) % SIZE == Head) goto 1
4 WrReq(⊥, QUEUE + Tail, new_value, mdata0)
5 FnReqAll(mdata1)
6 WrReq(⊥, TailAddr, Tail + 1, mdata1)
```

Listing 3.5: Enqueue (multiple channels)

```
1 RdReq(⊥, TailAddr, mdata0)
2 Tail ← RdRsp(mdata0)
3 if (Tail == Head) goto 1
4 RdReq(⊥, QUEUE + Head, mdata1)
5 r1 = RdRsp(mdata1)
6 WrReq(⊥, HeadAddr, Head + 1, mdata2)
7 FnReqAll(mdata2)
```

Listing 3.6: Dequeue (multiple channels)

init:	Head = 0, Tail = 0, Queue = [0,0]
FPGA:	enqueue(42)
	enqueue(43)
assert:	Head = 0, Tail = 2, Queue = [42,43]

(a) FPGA double enqueue

init:	Head = 0, Tail = 2, Queue = [42,43]
FPGA:	r1 = dequeue()
	r2 = dequeue()
assert:	Head = 2, Tail = 2, r1 = 42, r2 = 43

(b) FPGA double dequeue

init:	Head = 0, Tail = 0, Queue = [0]	
FPGA: enqueue(42)		CPU: r1 = dequeue()
assert:	Head = 1, Tail = 1, r1 = 42	

(c) FPGA enqueue, CPU dequeue

init:	Head = 0, Tail = 0, Queue = [0]	
FPGA: r1 = dequeue()		CPU: enqueue(42)
assert:	Head = 1, Tail = 1, r1 = 42	

(d) FPGA dequeue, CPU enqueue

Figure 3.8: Four litmus tests for validating the queue

guarantee that all previous writes have reached a point where they are globally visible between all threads in the system.

Listing 3.6 shows a multiple channel snippet of the code that dequeues from the `Head`. We have the same synchronisation issues in this version as with the single-channel version, but since traffic can flow on any channel (written ⊥), we need to use fences on all channels (line 7) instead of just waiting for responses. Similarly to the enqueue case, there is no need to wait for the response of the fence operation since there is no need for the writes to become globally visible.

Validation Using Our Operational Model Before implementing the producer/consumer queue in C++ and Verilog, we identified four litmus tests that capture the key interactions upon which the correctness of the queue depends. These tests include two successive enqueues, two successive dequeues, an FPGA enqueue followed by a CPU dequeue, and a CPU enqueue followed by an FPGA dequeue. We used the CBMC-based mechanism of our operational memory model (Section 3.3) to confirm that these litmus tests are all guaranteed to behave correctly according to our model. The tests that validate the queue primitives can be summaries as following:

Successive FPGA enqueues Two successive enqueues are performed on the FPGA as seen in Figure 3.8a. The assertion checks if the queue contains the correct values and if the head and tail are at the correct position.

Successive FPGA dequeues The queue is initialised with 2 elements and the FPGA performs two dequeues as seen in Figure 3.8b. The assertion checks if the FPGA received the values in order and the head and tail have the correct value.

FPGA enqueue, CPU dequeue The FPGA does an enqueue and the CPU does a dequeue as seen in Figure 3.8c. The assertion checks if the CPU received the correct values and if the head and tail are at the correct position.

CPU enqueue, FPGA dequeue The FPGA does a dequeue and the CPU does an enquque as seen in Figure 3.8d. The assertion checks if the FPGA received the correct values and if the head and tail are at the correct position.

These tests do not establish the correctness of the queue since they do not provide a complete formal evaluation. While a complete formal evaluation of the correctness of these queues is possible using our memory model, these tests give us the confidence that the basic operations of the queues are correct.

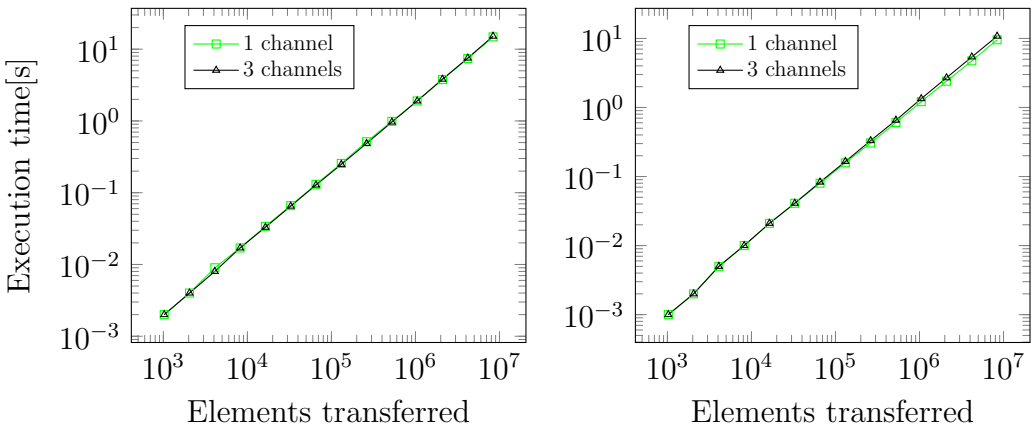
3.4.2 Performance comparison

Cloud providers widely deploy FPGAs as application-specific accelerators for customer use. Since most FPGAs contain more logic elements than a typical user can efficiently utilise, these providers multiplex their FPGAs among customers [VPK18, KLP+18]. Therefore we can expect that there will be multiple applications running on the FPGA that will use the same CCI-P interface. Therefore a realistic use case would be when the queue runs alongside an unknown amount of traffic. Moreover, it has been shown that some weak memory effects can only be exposed when the memory system is stressed with enough traffic [AMSS11, SD16].

To understand the effectiveness of the producer/consumer queue and its robustness to stress, we gained access to an X+F system though the Intel Academic Compute Environment [Int21]. This X+F system comprises a Broadwell Xeon CPU and an Arria 10 FPGA. While Intel provides a OpenCL SDK that can be used to run code for these category of devices, this SDK is based on an old version of OpenCL that does not provide access to all the fine-grain synchronisation that our memory model describes. Therefore, we use C++ to write code for the processor and SystemVerilog to write code for the FPGA.

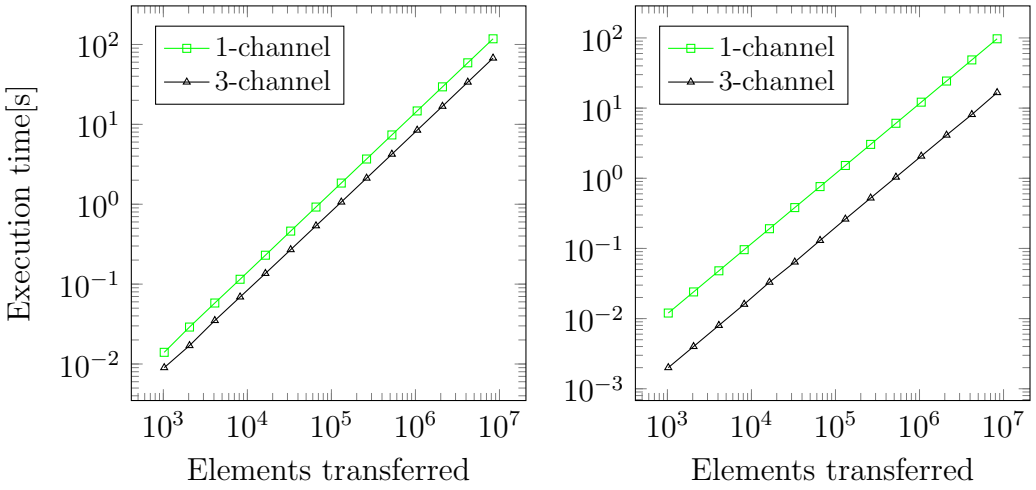
We implemented the two versions of the queue (synchronised using a a single channel and synchronised using multiple channels) and we make the CPU add the data and the FPGA was remove the data. We also implemented the mirror case where the FPGA is the one adding the data and the CPU is the one removing the data. We want to evaluate the effectiveness of these queues and therefore we measure their execution time with various number of elements being transferred. We repeat each experiment 10 times, but after observing that there was virtually no variance, we decided to report results just from the first.

Stressing the system We simulate traffic on the FPGA by sending write requests and read requests to the main memory at regular intervals. We call these requests *enemy* requests. The shorter the interval, the more traffic is created. Internally, we measure the length of the interval by using a counter that increments at every clock cycle. To avoid corrupting the state of our producer/consumer queue, we make sure to send requests to main memory addresses that are



(a) FPGA enqueue, without stress

(b) FPGA dequeue, without stress



(c) FPGA enqueue with stress

(d) FPGA dequeue with stress

Figure 3.9: Execution time of the two variants of the queue, transferring different number of elements in isolation and with heavy stress.

not utilised by any of the elements of the queue.

Figure 3.9 shows the execution time with a varying number of elements transferred. The number of elements transferred is always a power of two. We experiment with two scenarios: one where the FPGA continuously enqueues elements and the CPU continuously dequeues, and another where the roles are switched. In both cases, the `SIZE` of the buffer is 32. Afterwards, we repeat the same scenario but we add stress. For the same number of elements transferred, the FPGA can enqueue faster than it can dequeue. This can be explained by two factors: (1) FPGA main memory accesses are slower than CPU main memory accesses and (2) writes are more expensive than reads.

Without memory stress In the absence of enemy traffic, using one or multiple channels does not have significant impact on execution speed, whether the FPGA is enqueuing or dequeuing. Indeed, in Figure 3.9a, the lines indicating the execution time of the 1-channel and the 3-channels versions coincide. However, Figure 3.9b shows that using multiple channels is (slightly) slower. This decrease in execution time is marginal when the number of elements transferred is small, but becomes visible when more elements are transferred. We start to observe a decrease in execution time of 3% when 2^{16} elements are sent, reaching 12% when more than 2^{22} elements are sent.

To explain this, we consult the information related to channel utilisation that the CCI-P interface provides, and in this case, we can see that only one channel was used, regardless of the number of elements transferred. Allowing the use of multiple channels requires fences, but these operations are expensive and can cause a performance penalty when very few writes are present, as in the dequeue operation.

With memory stress Contention on shared resources of multicore systems caused by memory stressing can significantly impact execution time of completely independent processes [ISWD20, BY19, RGG+12]. We see the same effect on FPGAs, where adding stress significantly increases the execution time in all our experiments. In this case, the request pools are quickly filled and the FPGA is blocked until there is enough space to add a new request. In Figures 3.9c and 3.9d we can observe that the **multiple channel** variant has a consistently shorter execution time – about 60% of the execution time of the **single channel** variant. The decrease in execution time ranges from 35% (when just 2^{10} elements are transferred), up to 42% when more than 2^{18} elements are transferred. By examining the profiling information provided by CCI-P, we can see that this can be attributed to the fact that under heavy stress, queue traffic gets evenly distributed across all channels.

Summary From these experiments, we see that using a single channel can be the better option if we know that no unknown traffic will be present: if multiple channels are used unnecessarily, the queue will still only use a single channel but will require costly fences to ensure correctness.

However, when when there may be heavy traffic present, it is better to allow the CCI-P interface to redirect traffic onto multiple channels, and the cost of fences is justified.

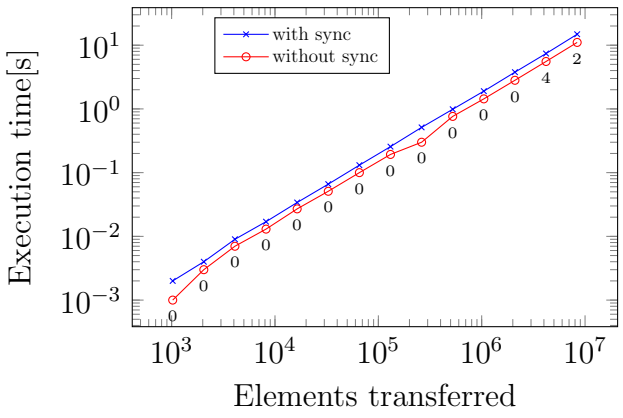
3.4.3 Exploring Incorrect Behaviour

The implementations described in Section 3.4.1, and the associated performance results in Section 3.4.2, are with respect to *correct* synchronisation, so that elements are dequeued from the producer/consumer queue in the same order in which they were enqueued, even in the presence of stressing traffic. We now explore the effects of *incorrect* synchronisation, showing that stressing traffic helps to expose incorrect synchronisation and get a quantitative handle on how unreliable an insufficiently-synchronised queue is in practice. The idea of eliminating synchronisation that is strictly necessary for correctness has been explored by previous work [Rin12]. Furthermore, while an insufficiently synchronised queue is unacceptable for use in a domain where the notion of correctness is binary, it is well-known that performance/quality tradeoffs are of interest in *approximate computing* domains, such as image processing and machine learning. A queue that performs more efficiently at the expense of losing elements with some probability may have useful application in such domains, and our formal memory model gives precise semantics to such a queue implementation.

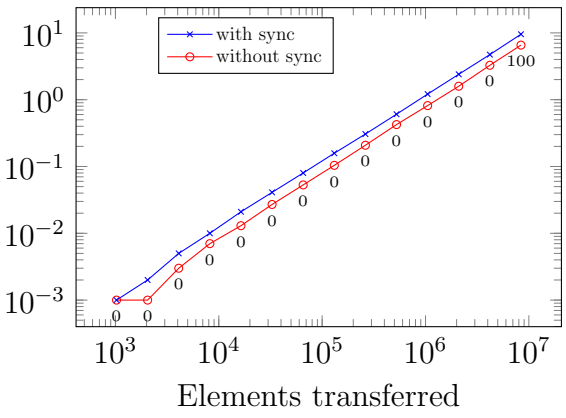
An improperly synchronised queue can result in lost messages, duplicate messages or reordered messages. In our experiments, any message that does not arrive in its correct position is considered a lost message. We call such a improperly synchornised queue, a *lossy* queue.

We conduct our case study by eliminating all write responses and fences and measuring the number of elements that are lost. The fences on the CPU side are kept to make reasoning about the queue easier. We run our experiments for different numbers of elemens transferred. Figure 3.10 shows a comparison of multiple versions of the fully synchronised queue with their coresponding version without synchronistion primitives.

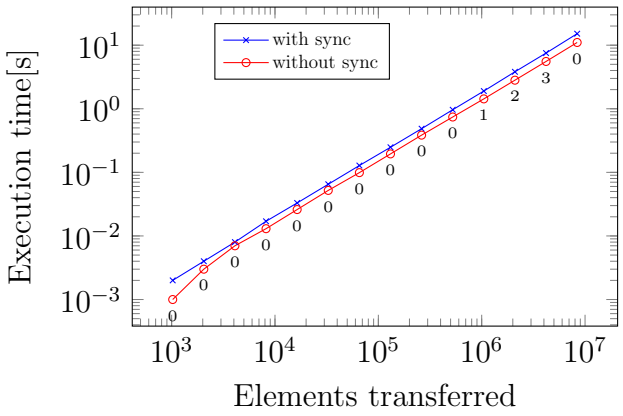
We first run the experiments in isolation (without stressing traffic) and observe that weak behaviour seldom occurs. The *lossy* versions of the enqueue tend to be about 20–27% faster



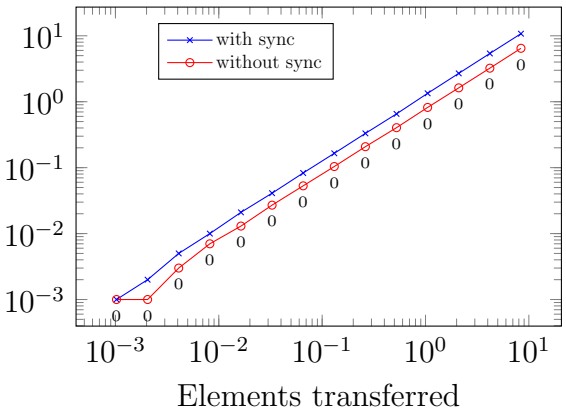
(a) FPGA enqueue, 1 channel, without stress



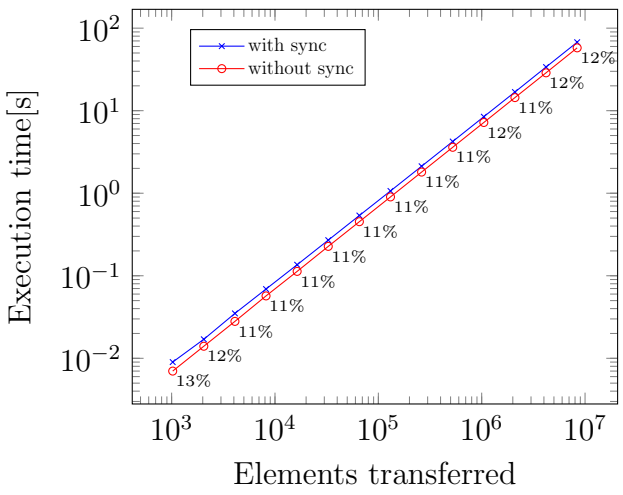
(b) FPGA dequeue, 1 channel, without stress



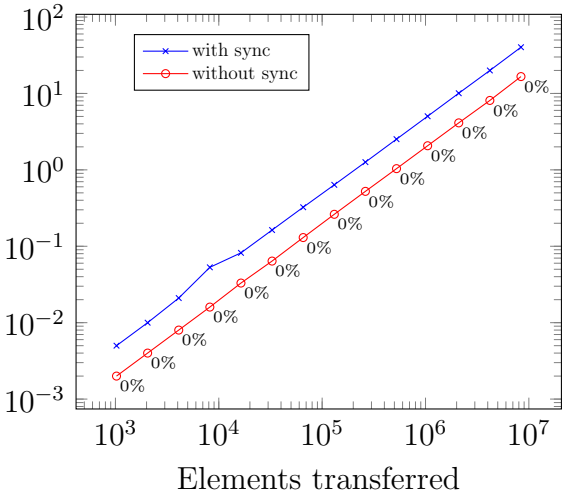
(c) FPGA enqueue, 3 channels, without stress



(d) FPGA dequeue, 3 channels, without stress



(e) FPGA enqueuem 3 channels, with stress



(f) FPGA dequeue, 3 channels, with stress

Figure 3.10: Comparing multiple versions of the the fully synchronised queue with the improperly synchronised versions. The numbers next to the data points give the number (percentage) of elements received incorrectly when synchronisation is omitted.

than the correctly synchronised versions, while the lossy dequeues are about 30–40% faster. We can see that without adding any kind of stress, the queue only manifests weak behaviour rarely and lost elements can be observed with large tranfers. This is true for both the versions of the queue that use a single channel and the versions of the queue that that uses multiple channels. An interesting behaviour can be observed in Figure 3.10b where elements are lost only when the 10^7 elements are transferred in the queue.

The stress exacerbates the weak behaviours in the enqueue. We experimented with many different configurations of the stress parameters but we only show the ones that were able to cause the most weak behaviour. Here we can observe a loss of about 11% of the elements. The performance benefits are clearer in this case, ranging from 14% to 22% faster for the enqueue and from 50% to 60% faster for the dequeue. This improved performance is of course irrelevant if the queue is intended for deployment in a scenario where no loss of messages can be tolerated. However, the reliability/performance trade-off may be of interest in approximate computing domains.

Summary A lack of synchronisation causes elements to be lost, but this behaviour can only be observed when significant stress is applied. There is a clear performance benefit of the lossy versions, which might be tempting if the application can tolerate data loss. However, we do not recommend this version of the queue unless we *clearly* know the data loss tolerance of the application.

3.5 Related work

CPU/FPGA Applications [ZCP16] have shown that implementing large-scale merge sort on an earlier version of the X+F can improve its throughput by 2.9× and 1.9× compared respectively to CPU-only and FPGA-only systems. [WHN19] and [ZP17] have shown that some graph algorithms are similarly well-suited to these platforms. In such platforms, the work is shared between the FPGA and the CPU threads and better results are obtained com-

pared to the CPU working alone. [WC17] have demonstrated similar results about K-means clustering applications using a different CPU/FPGA system called the Intel Cyclone V. More recently, some machine learning applications have improved their throughput when ported from a CPU/GPU implementation to a CPU/FPGA implementation [MKP20, GLL+19, GSQ+18].

FPGA Synchronisation Synchronisation primitives such as locks and barriers have been shown to be effective at enforcing orderings between FPGA threads [YFAE14]. Other works have shown how threads running on GPUs can be synchronised [SDB+16]. However, we are not aware of any work showing how to reason about the synchronisation of threads running on a CPU and on an FPGA. One work closely connected to ours [WC17] shows how OpenCL can be extended to support shared virtual memory (SVM) and its performance benefits. However we are not aware of any work showing how the memory model impacts this synchronisation.

Memory Modelling CPU memory models such as x86 [OSS09], POWER [SSA+11], Arm [PFD+17b], and RISC-V [PPPK+19] are now fairly well understood, as are some GPU memory models [ABD+15, LSG19]. However, these models do not apply to systems where threads are on different devices.

[LTPM15] provide a framework for translating between different memory consistency models. This is done with the aid of a format for specifying the semantics of memory orderings. Reasoning about the *combination* of two different memory models is not in the scope of that work, so it would not directly help with modelling a heterogeneous system like X+F.

Heterogeneous models between CPUs and GPUs have also been explored. [HHB+14] describe scoped memory models that arise principally in GPU computing, where threads are organised hierarchically into workgroups, and where it is desirable to be able to guarantee consistency at a particular level of this hierarchy. The X+F system does not have a scoped memory model, so the context of the [HHB+14] model is largely irrelevant to our setting. Furthermore, in their model, all compute units in the heterogeneous system are treated uniformly, whereas our model is sensitive to the respective idiosyncrasies of the CPU and the FPGA components. Also,

their model is pitched at the language level (OpenCL), rather than at the level of a particular architecture.

[ZTM+18] show how to reason about systems-on-chip by building what is called an *instruction-level abstraction* (ILA) [HZS+18] for each component. They do not address the challenge of coming up with (or validating) each ILA, so in this sense, our work can be seen as complementary to theirs. It is also not clear how to generalise their framework to deal with an FPGA memory model like ours, where reads and writes are split into requests and responses, and accesses are allocated to channels. We also note that none of the works mentioned above provide a means to generate test-cases for heterogeneous systems.

3.6 Summary

CPU/FPGA systems represent an attractive heterogeneous option, though they offer some unique programmability challenges. Therefore, we provided formal semantics in operational format for this system and mechanised them in C. The semantics can aid in reasoning about concurrent programs that run on the CPU and the FPGA. Moreover, the C implementation of the operational semantics provides a means by which executions can be explored through this system.

Furthermore, we have provided two correct versions of a popular producer/consumer synchronisation primitive and compared their performance in different scenarios. We have also shown how the lack of synchronisation can cause incorrect behaviours and how these behaviours can be exposed. Since these incorrect behaviours are rare, we consider our work the first step towards principled approximate computing on CPU/FPGA systems.

Chapter 4

Axiomatic model

While operational models are often considered more intuitive, a memory model based on axiomatic semantics can be orders of magnitude faster than one based on operational ones [AMT14]. Furthermore, axiomatic semantics can efficiently generate allowed and disallowed executions that can be used to test the model empirically. We develop an axiomatic memory model of the X+F system and aim to show that this is equivalent to the operational one. By automatically generating tests and running them on the actual hardware, we can also gain more confidence in the accuracy of our model.

In this chapter, we start with a formal description of the axiomatic semantics (Section 4.1) and show how sequential consistency can be recovered (Section 4.2); we then show the method used to test the model and gain confidence in its accuracy (Section 4.3). Afterwards we investigate the experimental results (Section 4.4) and conclude by presenting related works (Section 4.5) and a summary of our findings (Section 4.6).

4.1 A formalisation of the memory model

We now present an axiomatic formalisation of the X+F memory model. Axiomatic formali-

sations are attractive because they can be easily compared against each other [AMT14]. The more immediate advantage for us is the possibility of generating a suite of conformance tests automatically from the axioms (Section 4.3.1).

Notation We write r^* for the reflexive transitive closure and r^{-1} for the inverse of a binary relation r. Given binary relations r_1 and r_2 we define their join $r_1 ; r_2$ as $\{(x, z) \mid \exists y.\, (x, y) \in r_1 \wedge (y, z) \in r_2\}$. We write $[S]$ for the identity relation restricted to a set S, so $[S] ; r ; [T] = r \cap (S \times T)$. We use the convention that sets begin with an uppercase letter and relations begin with a lowercase letter.

4.1.1 Executions

We define an *execution* as a structure comprising a set of *events* plus several relations among those events. Each event represents one of the `WrReq`, `WrRsp`, `RdReq`, `RdRsp`, `FnReqOne`, `FnRspOne`, `FnReqAll`, `FnRspAll`, `CPUWrite`, `CPURead`, or `CPUFence` actions that we saw in Section 3.1.1. In what follows, given an execution X, we shall write `WrReq` for the set of events in X that represent `WrReq` actions, and so on. It is useful to define a few further subsets of events, so we write:

- `E` for the set of all events in the execution,
- `W` for `CPUWrite` $\cup$ `WrRsp`,
- `R` for `CPURead` $\cup$ `RdRsp`,[1]
- `Req` for `RdReq` $\cup$ `WrReq` $\cup$ `FnReqAll` $\cup$ `FnReqOne`,
- `Rsp` for `RdRsp` $\cup$ `WrRsp` $\cup$ `FnRspAll` $\cup$ `FnRspOne`,
- `CPU` for the events from the CPU, i.e. `CPUWrite` $\cup$ `CPURead` $\cup$ `CPUFence`, and
- `FPGA` for the events from the FPGA, i.e. `Req` $\cup$ `Rsp`.

Remark. It is unusual in axiomatic memory models to have separate events for requests and responses – usually there is just a single event for each read, write or fence. However, we find it necessary to track requests and responses explicitly in executions, as their relative order can affect whether an execution is allowed. For instance, consider the following executions:

init: x = 0	
FPGA:	1 RdReq(ch1, x, m1) 2 WrReq(ch1, x, 1, m2) 3 WrRsp(ch1, m2) 4 RdRsp(ch1, v, m1)
Allowed: $v = 0$	

init: x = 0	
FPGA:	1 WrReq(ch1, x, 1, m2) 2 WrRsp(ch1, m2) 3 RdReq(ch1, x, m1) 4 RdRsp(ch1, v, m1)
Disallowed: $v = 0$	

These two executions differ only by the position of the read request (highlighted), but the execution on the left is allowed (the read can observe the old value 0 because the read request preceded the write of the new value), while the execution on the right, where the read is requested *after* the write completes, is not.

The relations among the events in an execution are as follows:

- `sch` ('same channel') is an equivalence relation among all events that correspond to actions that specify a channel – that is, all events in `FPGA` except `FnReqAll` and `FnRspAll`.

- `sthd` ('same thread') is an equivalence relation that partitions all events into threads. In our model, the FPGA acts as a separate thread.

- `sloc` ('same location') is an equivalence relation among all non-fence events that connects events that access the same memory location.

- `rf` ('reads from') connects writes (either CPU writes or FPGA write responses) to reads (either CPU reads or FPGA read responses) at the same location – that is,

$$\texttt{rf} \subseteq [\texttt{W}] \,;\, \texttt{sloc} \,;\, [\texttt{R}].$$

  No read has more than one incoming `rf` edge. We use `rfe` as a shorthand for `rf` $\setminus$ `sthd`, which refers to a read from an external thread.

[1] Note that `W` and `R` contain CPU writes/reads and FPGA *responses* but not FPGA *requests*. `W` and `R` could have contained requests, or some mixture of requests and responses, but we found that these options led to more complicated axioms.

- `po` ('program order') is a strict, total order over all events within each thread. We further define $\texttt{poloc} = \texttt{po} \cap \texttt{sloc}$ and $\texttt{poch} = \texttt{po} \cap \texttt{sch}$.
- `co` ('coherence order') is a strict total order per location over all writes (either CPU writes or FPGA write responses).
- `fr` ('from-read') connects each read to all the writes that overwrite the write the read observed. Following [LWPG17], we define

$$\texttt{fr} \quad = \quad ([\texttt{R}] \,;\, \texttt{sloc} \,;\, [\texttt{W}]) \setminus (\texttt{rf}^{-1} \,;\, (\texttt{co}^{-1})^*).$$

  and we use `fre` as a shorthand for $\texttt{fr} \setminus \texttt{sthd}$.
- `readpair` connects each `RdReq` to its corresponding `RdRsp`.
- `writepair` connects each `WrReq` to its corresponding `WrRsp`.
- `fenceonepair` connects each `FnReqOne` to its corresponding `FnRspOne`.
- `fenceallpair` connects each `FnReqAll` to its corresponding `FnRspAll`.
- `pair` is a shorthand for $\texttt{readpair} \cup \texttt{writepair} \cup \texttt{fenceonepair} \cup \texttt{fenceallpair}$.
- `fencepair` is a shorthand for $\texttt{fenceonepair} \cup \texttt{fenceallpair}$.

We assume that requests and responses are paired up exactly; that is, every request has a `pair` edge to exactly one corresponding response, and vice versa. This means that we cannot reason about programs with dangling requests. It also means that we cannot reason about programs that use the same metadata tag for more than one request/response pair, but then again, such oddities are not interesting because they are easily rooted out by a preprocessing pass.

4.1.2 Consistency axioms

Before stating the axioms that capture when an execution is deemed consistent, we require a few more derived relations. The following derived relations capture the effect of fences on the

CPU and on the FPGA:

$$\begin{aligned} \texttt{fenceCPU} &= \texttt{po} \,;\, [\texttt{CPUFence}] \,;\, \texttt{po} \\ \texttt{poFnRsp} &= (\texttt{poch} \,;\, [\texttt{FnRspOne}]) \cup (\texttt{po} \,;\, [\texttt{FnRspAll}]) \\ \texttt{fenceFPGA} &= [\texttt{WrRsp}] \,;\, \texttt{poFnRsp} \,;\, \texttt{po} \,;\, [\texttt{E} \setminus \texttt{RdRsp}] \\ \texttt{fence} &= \texttt{fenceCPU} \cup \texttt{fenceFPGA} \end{aligned}$$

The `fenceCPU` relation holds between any CPU events in program order that are separated by a fence. The `fenceFPGA` relation captures the guarantee that when a fence response is received by the FPGA, all previous writes on the specified channel (or all channels if the fence does not specify one) have propagated to memory, so any subsequent read requests will see the new values. The $[\texttt{E} \setminus \texttt{RdRsp}]$ part is to allow for the fact that responses to reads that have *already been requested* may still contain old values.

The following derived relations capture the 'preserved program order' [AMT14] for the CPU and the FPGA:

$$\begin{aligned} \texttt{ppoCPU} &= \texttt{po} \setminus (\texttt{W} \times \texttt{R}) \cap \texttt{CPU}^2 \\ \texttt{ppoFPGA} &= ([\texttt{Rsp}] \,;\, \texttt{poch} \,;\, [\texttt{E} \setminus \texttt{RdRsp}]) \cup ([\texttt{RdRsp}] \,;\, \texttt{po} \,;\, [\texttt{E} \setminus \texttt{RdRsp}]) \cup \texttt{pair} \\ \texttt{ppo} &= \texttt{ppoCPU} \cup \texttt{ppoFPGA} \end{aligned}$$

The `ppoCPU` relation is inherited from the TSO memory model (with the added restriction that it only applies to CPU events). The `ppoFPGA` relation captures that (1) responses are not reordered with subsequent events on the same channel, (2) read responses are not reordered any subsequent event. and (3) request/response pairs are kept in order.

We are now ready to state the consistency axioms. An X+F execution is deemed consistent if the rules defined in figure 4.1 hold.

SC-PER-LOC is familiar from the TSO memory model; we have added the restriction to CPU events. PROPAGATION is the other standard TSO axiom, which we have enhanced with

acyclic((poloc ∪ rf ∪ fr ∪ co) ∩ CPU^2)	SC-PER-LOC
acyclic(ppo ∪ fence ∪ rfe ∪ fre ∪ co)	PROPAGATION
irreflexive(fr ; poch ; readpair)	READ-AFTER-WRITE
irreflexive(fr ; poFnRsp ; po ; readpair)	READ-AFTER-FENCE
irreflexive(rf ; po)	NO-READ-FROM-FUTURE
acyclic(fre ∪ rfe ∪ (rf \ sch) ∪ poch ∪ ppoCPU)	OBSERVE-SAME-CHANNEL
irreflexive(po ; fencepair ; po ; writepair^{-1})	FENCE-RESPONSE
irreflexive(po ;(fencepair ∪ writepair); po ; fenceallpair^{-1})	FENCE-BLOCK
irreflexive(rf ; poloc ; co)	WRITE-ORDER-CHANNEL

Figure 4.1: The axioms of the X+F memory model.

Remark. The original published version of these semantics (OOPLSA21 [IDSW21b]) contained some infidelities. These infidelities were caused by some limitations in our simulation technique that we were able to overcome by using the techniques presented in Chapter 5. The semantics presented here are the revised ones that do not contain the mistakes present in the original paper.

fenceFPGA and ppoFPGA . The two READ-AFTER-* axioms concern the situation where a read request (say r) is po-after a write response (say w) on the same location; they say that when r and w are on the same channel (READ-AFTER-WRITE), or are separated by a fence that is either on the same channel as w or on all channels (READ-AFTER-FENCE), then r must observe w (or a co-later write). NO-READ-FROM-FUTURE prevents reads observing writes that haven't been issued yet and OBSERVE-SAME-CHANNEL prevents writes from a different thread being observed out-of-order on the same channel. The FENCE-RESPONSE and FENCE-BLOCK axioms describe the reorderings that fences enforce on the executions. The WRITE-ORDER-CHANNEL deals with ordering guarantees of multiple writes to the same channel.

4.2 Restoring sequential consistency

The memory provided by the X+F system is weaker than traditional CPU systems, allowing even executions that could violate coherency. However, the system allows sequential consistency to be recovered using the appropriate synchronisation. Here, we show how this can gradually be achieved through different examples.

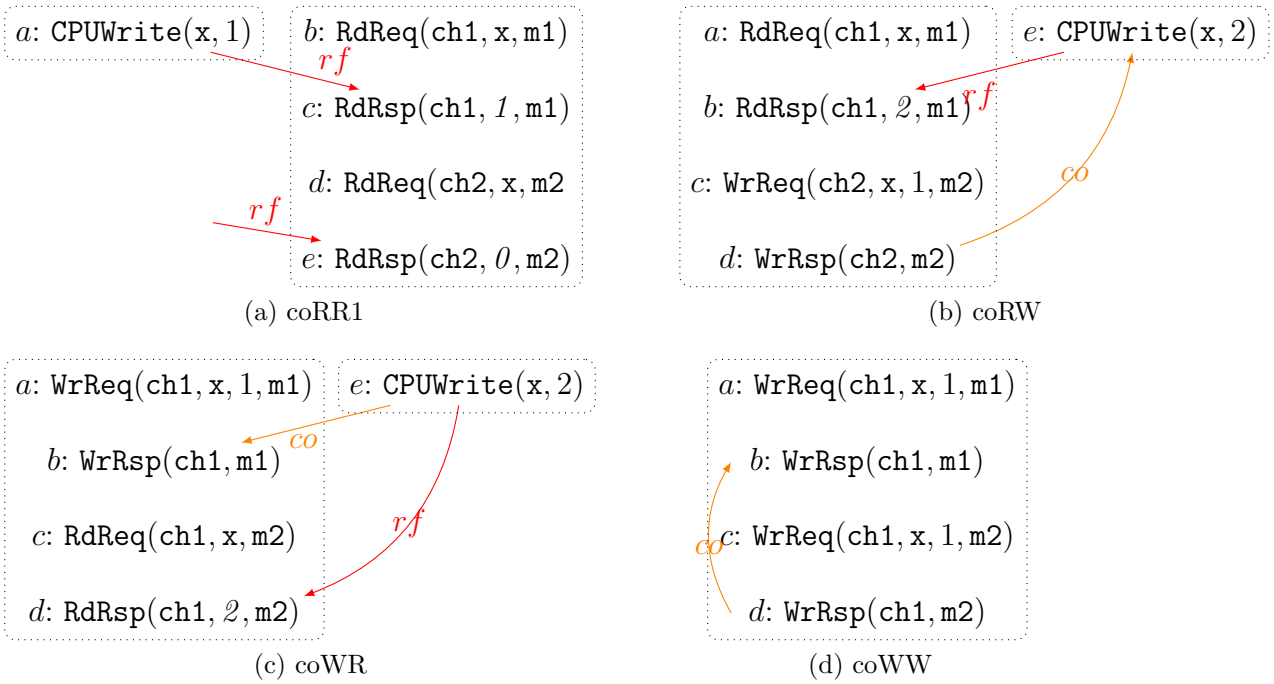


Figure 4.2: Coherancy litmus test

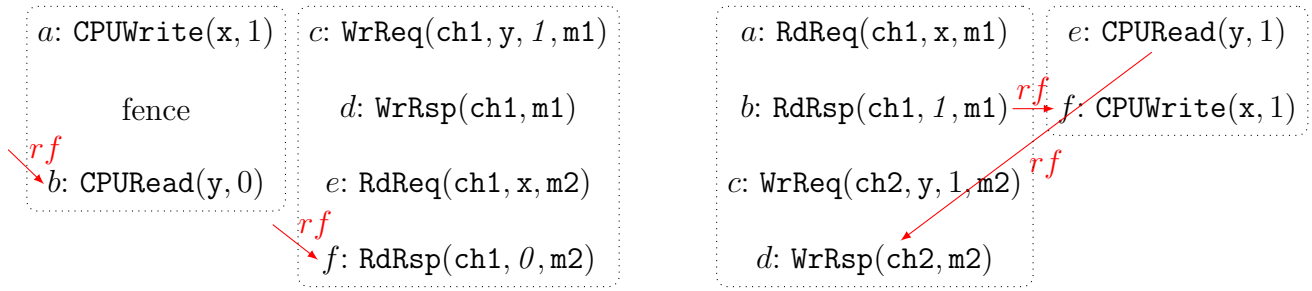
We focus on what we consider the most illustrative executions from the POWER and ARM Litmus Tests [Sew23] (these are also more commonly known as "The Periodic Table of Litmus tests). The names of the tests are taken from the original work. We show how each of these executions can be disallowed with the proper synchronisation.

4.2.1 Restoring coherancy

The X+F system does not guarantee coherency since requests can get reordered in their corresponding pools or travel via different channels at different speeds. Figure 4.2 shows how coherency can be assured, and different behaviour is forbidden.

Figure 4.2a shows the read-after-read test. To prevent the second read (**e**) from observing an old value, its corresponding read request (**d**) must only be issued after the first read response (**c**) has arrived.

Similarly, Figure 4.2b shows the write-after-read test. To prevent the FPGA write (**d**) from occurring before the CPU write(**e**), we must issue the write request (**c**) only after the previous read response (**b**) has returned with the observed CPU value.



(a) Store buffering (b) Load buffering

Figure 4.3: Load and store buffering

Figure 4.2c shows the read-after-write test. To prevent this execution, we need to ensure that the read request (**c**) is only issued after the write response has arrived (**b**) and also assure that both utilise the same channel. If different channels are used, the read can altogether bypass the write and observe the CPU write (**e**) and only afterwards, the write from the FPGA (**b**) reach the main memory.

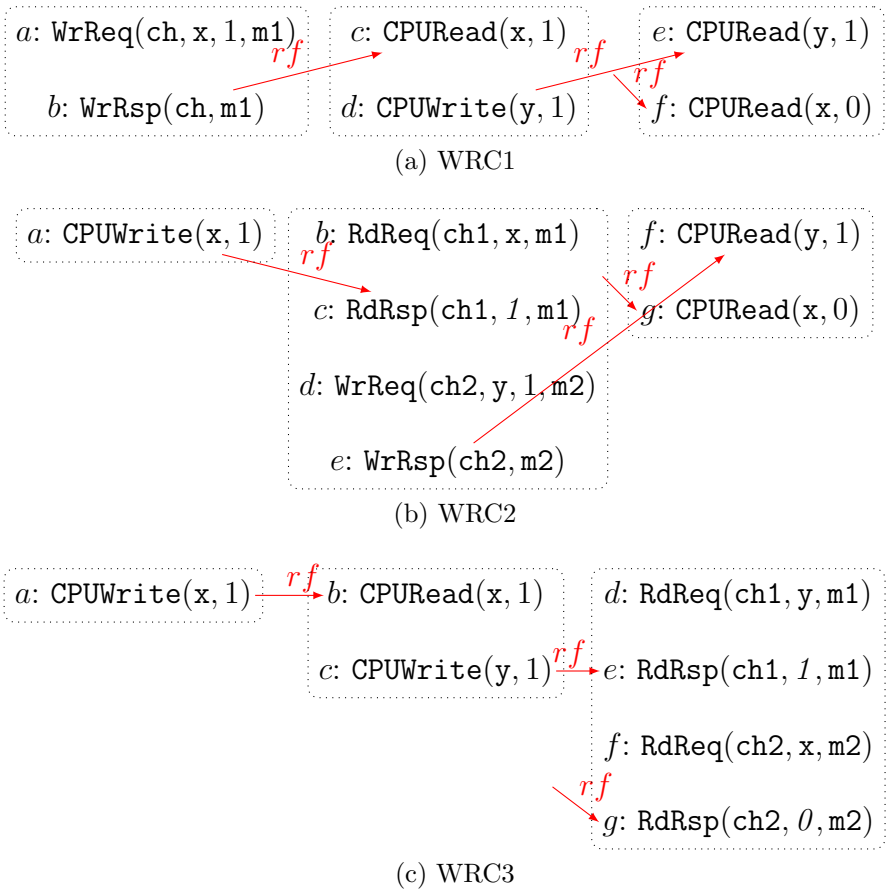
Figure 4.2d again shows an example where the same channel must be utilised for both writes to prevent the behaviour described. If different channels are used, the writes can effectively be reordered with each other.

4.2.2 Store and load buffering

Store buffering and load buffering are popular litmus tests used to characterise most memory models. We can see these tests in Figure 4.3. The store buffering effect can be observed on x86 processors while the load buffering one can not.

Store buffering is presented in Figure 4.3a. The execution can happen if the CPU write (**a**) gets reordered with the CPU read (**b**) or the response of the FPGA write (**d**) gets reordered with the response of the FPGA read (**f**). To prevent reordering on the CPU side, we need to add a fence operation between the two operations. Furthermore, to prevent the reordering on the FPGA side, we need to issue the read request (**e**) only after we have received the response from the FPGA write (**d**) and use the same channel for both operations.

Load buffering is presented in Figure 4.3b. The execution can happen if the CPU read (**e**) gets



(a) WRC1

(b) WRC2

(c) WRC3

Figure 4.4: Causal consistency litmus tests

reordered with the CPU write (**f**) or the response of the FPGA read (**b**) gets reordered with the response of the FPGA write (**d**). To prevent the reordering from occurring, we need to wait for the FPGA read response (**b**) before issuing the FPGA write (**c**). Since no reordering can occur on the CPU side, there is no reordering required on that side.

4.2.3 Causal consistency

Causal consistency is a property of memory models that ensures that the order of events is preserved across multiple threads in the system. It ensures that if an event occurs before another event in one thread, the exact ordering of events should also be observed in any other thread in the system. This can be used to coordinate the execution of multiple threads, ensuring that memory accesses are performed in a predictable order.

Figure 4.4 shows an example with multiple threads that need to coordinate with each other.

Table 4.1: The total number of disallowed and allowed litmus tests, grouped by event count.

#Events	Disallowed	Allowed
4	9	0
5	10	0
6	38	2
7	72	26
8	454	152
Total:	583	180

In Figure 4.4a, the first thread is set on the FPGA, in Figure 4.4b, the second thread is set on the FPGA, while in Figure 4.4c, the third thread is set on the FPGA. No synchronisation is required in the first case, but in the second and third cases, we need to wait for the previous responses to arrive before sending the subsequent requests.

4.3 Testing the axiomatic model

In this section, we present our method for validating the accuracy of our axiomatic memory model. We start by presenting how we can automatically generate executions from the axiomatic memory model; we then present our idea for cross-checking the operational model, the axiomatic model and the actual hardware. The final part of this Section presents how we optimised testing on the actual hardware.

4.3.1 Generating executions from the axioms

By encoding the above constraints in the Alloy modelling language, we can use the Alloy Analyzer to generate a large number of executions that violate at least one axiom, as shown by previous work by [LWPG17]. This corpus of executions can serve as a conformance suite. We generated executions that had a single FPGA thread and at least one write that we can observe.

Following [LWPG17], we only generate 'interesting' disallowed executions – those that use the least synchronisation necessary to prevent a particular outcome. We use this to generate only

the disallowed traces where every event is *critical*: i.e. removing any event from such trace will cause the trace to become allowed. This ensures that exhaustive test generation remains feasible as the event count grows. Every event in an interesting test is *critical*: i.e. removing any event from such trace will cause the trace to become allowed. Exhaustively generating all executions quickly becomes infeasible as the number of events increases. Moreover it is unnecessary to do so as such a suite would contain a large amount of redundancy. Tests may use overly-strong synchronisation, may contain operations that have no effect, or may simply duplicate a pattern already covered by another test in the suite. Conversely, a test is in fact useful if it covers some pattern not already tested within the suite.

We modify the technique of [LWPG17] to take into account the fact that in contrast to their CPU counterparts, FPGA events always occur in pairs: requests and their corresponding response. Using this approach, we are able to generate a total of 583 interesting *disallowed* executions. The second column of Table 4.1 breaks these executions down by event count.

We further generate a total of 180 *allowed* executions by removing one or more fences from these disallowed executions. Removing any CPU instruction or any pair of FPGA events from each of these 583 executions will cause it to become allowed. Removing any fence will therefore cause the execution to become allowed. Since only a subset of the disallowed executions contain fences, using this approach we generate an additional 180 allowed execution. We could remove reads or writes as well as fences, but this would require the test's postcondition to be recalculated.

4.3.2 Cross-checking the axiomatic and operational model

The operational model and the axiomatic model should be equivalent as both should accurately describe the X+F system. While developing both models we repeatedly cross-validated them against each other as can be seen in Figure 4.5. We wrote a back-end to turn an Alloy-generated execution into an input to the CBMC model, using `assume` statements to describe the sequence of events in the execution and `assert` statements to validate whether final condition can be realised. This process revealed several discrepancies between the models during development. We manually inspected each discrepancy, fixed the inaccurate model, and added the execution

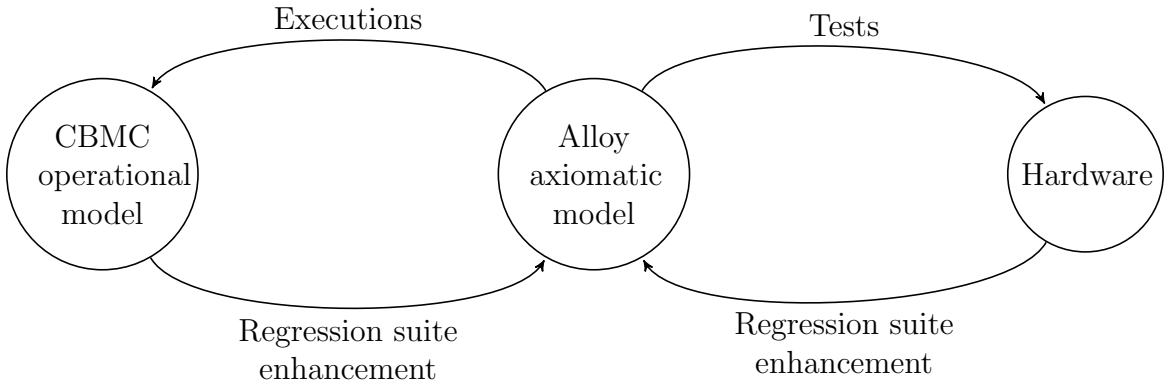


Figure 4.5: Cross-validation flow

that identified the discrepancy to our regression test suites for both models. For example, in our initial modelling attempt we oversimplified the axiomatic model by merging requests and responses into single events, as remarked in Section 4.1. As a result, the automatic cross-validation flow generated traces that were disallowed in the axiomatic model but allowed in the operational model. This modelling decision proved incorrect and was later fixed.

We used this cross-validation approach to gain confidence that the axiomatic and operational models agree for all of the litmus tests discussed in Section 4.3.1, except that we skipped 34 tests that involve more than two CPU threads: our CBMC implementation only supports two CPU threads as we found that model checking for larger thread counts did not scale. In our view, the ability to cross-validate is a key reason for developing mechanised axiomatic and operational semantics for the same memory model.

We also aimed to validate these models against the actual hardware using the Alloy-generated tests. However, synthesising a test-case for the FPGA is quite slow and didn't initially allow us to execute many tests. We were able to overcome this challenge with the aim of the soft-core processor described in the next section.

4.3.3 A Soft-core processor

To validate the memory model, we need to run a large number of litmus tests on the X+F hardware. Compiling the CPU part of the litmus test is fast, but synthesising the corresponding FPGA part takes between one and two hours. This long time might be due to the fact that the

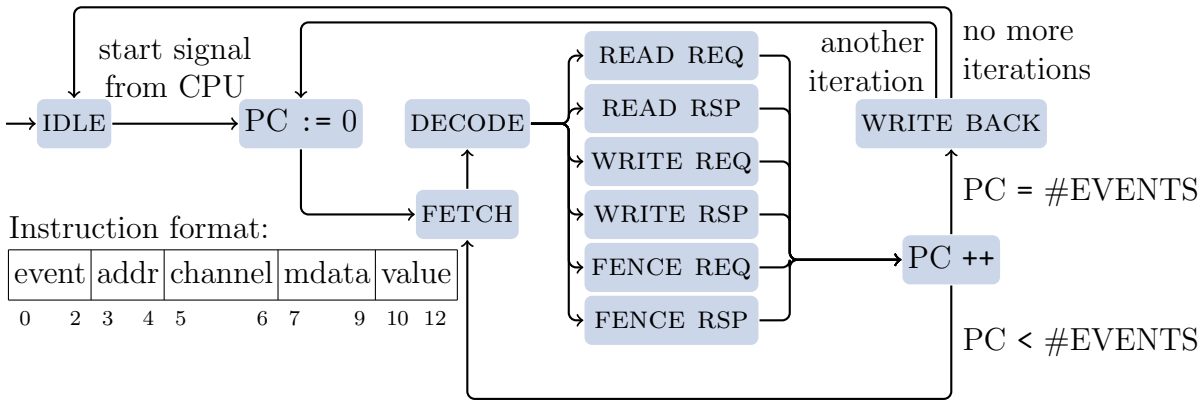


Figure 4.6: A simplified representation of the soft-core state machine and the instructions it processes.

current flow re-synthesises some elements such as the virtual-to-physical-address translator.[2] Therefore, performing synthesis separately per litmus test is not feasible.

To overcome this, we have designed a simple soft-core processor that runs on the FPGA and only needs to be synthesised once. A litmus test is encoded as a sequence of instructions and sent by the CPU to the FPGA for execution. Each instruction captures the event type, and (if relevant) an associated address, channel, metadata and value.

For each litmus test, the CPU allocates and initialises the necessary shared memory locations. It then communicates these to the FPGA, along with the instruction sequence that the FPGA thread should execute and the number of times the test should be repeated. The CPU detects when the FPGA thread has finished executing a litmus test by busy-waiting on a designated flag location in shared memory.

Once setup is finished, the CPU can start the litmus test by starting the CPU threads, signalling the FPGA to start its thread and then wait for the CPU and FPGA thread to finish. The CPU will get the state of the FPGA thread by waiting in a loop while repeatedly checking a shared memory value. It will then assert if the behaviour observed is valid.

Figure 4.6 depicts the soft-core processor as a state machine. At the beginning of execution, the processor is in the IDLE state. When the FPGA receives the signal from the CPU to

[2]Changing the Intel-provided flow might reduce synthesis time, but we anticipate that it will still be several minutes per test. This would be feasible for the final testing campaign but would still make iterative development of the model intolerable.

start, it jumps to FETCH the first instruction from its local memory, initially from a location associated with program counter value 0, then proceeds to DECODE it to decide what to execute. The soft-core processor can issue a `WrReq`, `RdReq`, `FnReqOne`, or `FnReqAll`, or it can wait for the corresponding `WrRsp`, `RdRsp`, `FnRspOne` or `FnRspAll`; each instruction type is handled via a dedicated state. The program counter is then incremented and, based on the number of remaining instructions, the processor either fetches the next instruction or, if the litmus test has finished executing, proceeds to WRITE BACK the test results. In the WRITE BACK state, the soft-core processor needs to inform the CPU about the state of the litmus test just executed. Since a litmus test can dictate the order in which read, write, and fence requests are issued but cannot control the order in which associated responses arrive, the desired sequence of events associated with a particular litmus test might be impossible to reproduce. Furthermore the FPGA cannot easily display the data it has read from a `RdReq`. This needs to be communicated back to the CPU so that it can validate (a) whether the sequence of events that occurred during test execution respects the sequence required by the litmus test, and (b) if so, whether the weak behaviour has been observed. After sending back this information, the soft-core either repeats the litmus test, if it has not yet performed the required number of test iterations, or informs the CPU (via a flag in shared memory) that it is ready to move on to the next test, and returns to IDLE.

Stress Generation Previous work [SD16, AMSS11] has shown how stress testing can expose weak memory behaviour. It motivated us to incorporate a stress generator in our processor. This is comprised of a simple circuit that monitors the request interface and issues random requests at specific intervals. The frequency of these requests is given by a parameter received from the CPU. To ensure that stress-related requests do not corrupt requests that form part of a litmus test, they are only issued on clock cycles during which the processor state machine does not need to issue a litmus test-related request of the same type.

Table 4.2: The total number of disallowed and allowed litmus tests generated, and the number of observed behaviours without stress and with stress. We write m/n for 'm observed out of n tests'. The experiments with stress were only done for a small sample of the litmus tests.

	All interesting tests, run without stress		Sample of interesting tests, run with stress	
#Events	Disallowed	Allowed	Disallowed	Allowed
4	0/9	0/0	0/1	0/0
5	0/10	0/0	0/2	0/0
6	0/38	0/2	0/2	0/0
7	0/72	0/26	0/2	1/4
8	0/454	0/152	0/3	3/6
Total:	0/583	0/180	0/10	4/10

4.4 Experimental evaluation

The cross-validation efforts described in Section 4.3 gave us a high degree of confidence in our formalization of the X+F memory model, but we also wanted to validate our model against real hardware. For this purpose, we used the same experimental setup described in Section 3.4.2. This X+F system comprises a Broadwell Xeon CPU and an Arria 10 FPGA. We used our axiomatic model to generate disallowed and allowed executions as described in Section 4.3.1 (Table 4.1). This process was feasible for up to 8 events, after which the Alloy Analyser increased in execution time dramatically.

We developed a translator that converts Alloy executions into litmus tests for the X+F hardware. This translator generates the C++ code describing the CPU threads, the instructions to be sent to the soft-core, and the assertions that check the final state. Our translator encodes each event as a separate instruction with the corresponding address, channel, metadata and value. These fields are determined based on the edges that connect these events. As an example, events that are connected by an `sch` edge will be assigned to the same channel. Executions that require more resources than are available on the actual hardware (e.g. more than three channels) are discarded.

Our approach using the soft-core processor allows us to quickly run our generated litmus tests 1 million times each. The results of these experiments can be seen in the left half of Table 4.2.

Reassuringly, we did not observe any of the disallowed behaviours, even when enabling the stress generator. Recall that the allowed litmus tests are 'only just' allowed, being derived from disallowed tests via the removal of one critical event. We were unsure whether we would observe these behaviours in practice: they might be allowed by the X+F documentation in principle but impossible to observe on our test platform in practice, or they might be observable only extremely rarely. Indeed, we did not observe any of these behaviours when we ran tests in isolation. However, we did observe a subset of the behaviours after experimenting with a variety of configurations for stress traffic, as described below.

Tuning Stress The low number of observed allowed litmus tests indicated that we need to find a better way to expose weak behaviour. The deterministic nature of FPGAs makes some executions highly unlikely. [KSTM20] have shown that tuning stress has an important role in exposing weak behaviour. We attempted to automatically script the repeated running of tests under many different stressing configurations, but found that the device was prone to becoming unresponsive, making automated tuning impossible. (We have contacted Intel to make them aware of the problem.) In the meantime, we manually tuned stress for 10 allowed tests and 10 disallowed tests, re-flashing the board each time it became unresponsive. Repeating this manual process for more tests would have taken an infeasible amount of time. As shown on the right half of Table 4.2, we were able to observe weak behaviour in 4 out of the 10 allowed executions and (again, reassuringly) no weak behaviour in the 10 disallowed executions.

Observed Behaviours Some allowed behaviours were easier to observe than others. The 10 allowed litmus tests that we executed with fine-tuned stress can be roughly categorised into three categories:

1. The easiest weak behaviours to reproduce were those caused by reorderings between channels. In such a case, if the first request was sent on a slow channel, the second one on a fast channel and the correct amount of extra traffic was added just to the slow channel, it was highly likely that weak behaviours would be observed. There were three of these tests, and all eventually exhibited weak behaviour.

2. It was significantly harder to provoke weak behaviours in litmus tests that require reordering in the request pools. There were three of these tests, and we only managed to observe weak behaviour in one of them.

3. A separate category of litmus tests were the ones that required the responses to arrive in a very specific, and often rather improbable, order. In this category there were four litmus tests. The responses are controlled by the interface logic, so the FPGA logic cannot explicitly control them. We were not able to expose any weak behaviour in this category, and it could be that such weak behaviours cannot be observed due to the way the hardware is implemented (but the Intel documentation provides no guarantees about this).

To illustrate the importance of tuning enemy traffic, consider a litmus test where two writes utilise different channels. In such a litmus test we only want to stress the channel corresponding to the first write. This causes the first write to be delayed and possibly be inverted with the second write, causing the weak behaviour. We similarly explored the effect of stress testing to expose weak behaviour in Section 3.4.3.

4.5 Related work

Axiomatic memory models represent an alternative to operational memory models that are easier to simulate [AMT14] than operational ones. A significant amount of CPUs [MSS12, AMT14, PPPK+19, PFD+17b] and GPUs [WBBD15, LSG19] have been modelled axiomatically. However, none of these works have axiomatically modelled a system where part of the threads run on the CPU and the other run on an FPGA

Axiomatic models enable the generation of large amounts of litmus tests, and previous work has taken advantage of this to automatically compare memory models [LWPG17, LSG19, WBSC17]. These works have focused on the memory model provided by various programming languages and architectures. However, these works have not tackled the problem of automatically generating litmus tests for CPU/FPGA systems.

Litmus testing does not always uncover all the weak memory effects, and previous work [AMSS10, SD16, KSTM20, ABD+15] has shown that some of these effects can only be uncovered when properly configured stress is added. Our work similarly utilises stress as a method to uncover weak memory effects but is adapted to the idiosyncrasies of the FPGA.

4.6 Summary

While axiomatic models can be considered less intuitive than operational ones, the simulation time required for such models is significantly lower. Furthermore, this axiomatic model allows us to automatically generate executions that can be used to cross-check against the operational model and the actual hardware.

While testing against the operational model did not require significant engineering effort, testing against the operational one required more effort. Therefore, we developed a soft-core processor to aid in running tests against the hardware. This process allowed us to gain confidence in the accuracy of our model.

Chapter 5

Simulating operational memory models using off-the-shelf tools

The effectiveness of the cross-checking in Section 4.3.2 is limited by the limited scalability of the operational model. Scaling up the number of events simulted by the operational memory model represents the bottleneck in our approach and limits the confidence of our evaluation. This motives us to investigate alternatives simulation techniques.

This chapter is divided into two parts: (1) a study of the strengths and weaknesses off three off-the-shelf C analysis tools and a comparison with RMEM [ABC+22] in the context of the well-known x86 memory model, and (2) an in-depth study of our CPU+FPGA memory model, enabled by our use of off-the-shelf tools.

Part 1: x86 case study. In the first part of this chapter, we are interested in the following top-level research questions:

RQ1 Can reducing the problem of memory model simulation to the analysis of a C program yield competitive performance compared with bespoke simulators?

RQ2 Of the variety of C analysis tools that are available, which are most effective for memory model simulation?

Part 2: CPU/FPGA case study. We then apply our ideas to a memory model for which there is no bespoke simulator: our X+F memory model presented in Chapter 3. It poses particular challenges due to the complexity of shared memory interactions between CPU cores and FPGA logic. In Chapter 3 we have already presented a C-based realisation of this memory model and used CBMC to validate it against a supposedly-equivalent axiomatic memory model, but our previous evaluation was restricted to small litmus tests, featuring at most 8 instructions per test, due to scalability limitations of CBMC. Given our finding from Part 1 that libFuzzer performed very well for x86 litmus tests, we investigate the following research questions in the context of the X+F model:

RQ3 How does the manner in which the memory model and litmus test are encoded as a C program impact the performance of the different tools?

RQ4 Can our approach allow more in-depth analysis of the X+F memory model, allowing it to be better validated against its axiomatic counterpart?

Thanks to the better scalability of coverage-guided fuzzing compared with SAT-based model checking, we were able to perform a substantially deeper analysis than our previous attempts. This allowed us to find four infidelities in the X+F axiomatic memory model. We have fixed the infidelities in the model so that it now accounts for additional ordering guarantees that we previously overlooked.

We start with an overview of our approach in Section 5.1. Section 5.2 shows how we have tackled **RQ1** and **RQ2** while Section 5.3 shows how we have tacked **RQ3** and **RQ4**. We discuss related work in Section 5.4 and conclude in Section 5.5.

5.1 Overview of our approach

An operational memory model takes as input the *litmus test* that defines the sequence of instructions for each concurrent component of the system, and a description of a final state

of interest. It then facilitates searching for transitions of the system that might lead from an initial state to the final state of interest. A trace that leads to the final state indicates that this behaviour is *allowed*; if no such trace exists it is *disallowed*. Non-determinism arises due to the order in which the instructions are issued by the concurrent components, and due to the internals of the memory system (such as flushing policies for buffers and caches). Once a trace that reaches a state of interest has been found, the programmer or memory model engineer can use the simulator to step through the trace in detail to better understand its behaviour.

A state-of-the-art simulator for operational memory models is RMEM [ABC+22], which has been used to simulate the memory models of ARM [PFD+17b, FGP+16], Power [SSA+11, BMO+12, SMO+12], RISC-V [PPPK+19] and x86 [OSS09]. Building a *bespoke* simulator such as RMEM requires a lot of engineering effort: not only must the memory model of interest be encoded, but algorithms for efficient reachability analysis must be implemented.

Reachability has been studied extensively in the context of program analysis, and a range of off-the-shelf tools that attempt to decide whether a program can reach a particular state are available for several languages. This leads to the following idea: instead of implementing a bespoke memory model simulator, why not implement the simulator logic as a computer program that takes a particular test scenario as input? Determining whether the test scenario is allowed would then boil down to determining whether a particular state of the program that encodes the memory model is reachable when executed on an input describing the scenario of interest, and off-the-shelf reachability analysis tools for the language of interest could be leveraged to answer this question. Subsequent detailed examination of traces would then be possible by stepping through the simulator code using a standard debugger.

We use the x86 memory model since it is simple and widely-used. Since a significant amount of recent work utilises RMEM to simulate operational models, we consider this the state-of-the-art and compare our approach to it. We investigate our idea of reducing to C and then leveraging existing tools with respect to three diverse analysis tools for C: a SAT-based model checker, CBMC [CKL04]; a dynamic symbolic execution engine, KLEE [CDE08], and a coverage-guided fuzzer, libFuzzer [Ser22]. Of these, CBMC is a fully symbolic analyser, libFuzzer is a fully

dynamic analyser, and KLEE mixes symbolic and dynamic analysis.

Despite the advantages of coverage-guided fuzzing, for particularly complex litmus tests where an allowed behaviour is exhibited by only a tiny fraction of paths, SAT-based model checking is able to demonstrate that the behaviour is allowed while exploration using coverage-guided fuzzing gets lost. Furthermore, because symbolic execution and SAT-based model checking are capable of *exhaustive* exploration—unlike fuzzing—they can be used to demonstrate that certain memory model behaviours are *disallowed*. We report on our experience putting this idea into practice, using C as the implementation language. We focus on C not out of any particular fondness for the language, but due to the availability of a diverse range of C analysis tools. We notice that there are a lot of off-the-shelf tools for deciding reachability of program paths and aim to investigate if these tools can be better used to tackle the path explosion problem. Our vision is to take advantage of these tools by first converting the model into a C program and then plug in any such tool that can analyse C programs. We can summarise our approach as reducing the decision problem of "whether a given operational model allows a given program behaviour" to the decision problem of "whether a given C program is safe", which can be handled by a variety of off-the-shelf tools.

In this section, we show how the x86 memory model can be encoded as a C program (Section 5.1.1) and illustrate how program analysis tools can be used to simulate memory models (Section 5.1.3).

The simulation will result in a program that takes as input the program described by the litmus test, explores paths allowed by the semantics of the memory model, and decides if the behaviour expected by the litmus test is possible.

5.1.1 Reducing x86 Analysis to C Reachability

Our approach is to encode memory models using the C programming language, and to leverage off-the-shelf C program analysis tools for simulation purpose. We make this idea concrete by implementing a C model of the x86 memory model in Listing 5.3. The sequence of instructions

```
struct Operation {
  Opcode type;
  int var;
  int val;
};
```

Listing 5.1: The structure of an operation

```
enum Action {
  CPU_THREAD,
  FLUSH_BUFFER
};
```

Listing 5.2: The possible actions

```
int sim_steps = choose(SIMULATION_STEPS);
for (int i = 0; i < sim_steps; i++) {
  // Can be one of the n threads in the system
  int thread = choose(NUM_THREADS);
  // Can be either CPU_THREAD or FLUSH_BUFFER
  Action action = choose(NUM_ACTIONS);
  switch (action) {
    case CPU_THREAD:
      if (!(thread_ops[thread].empty())) {
        Operation op = thread_ops[thread].pop();
        if (op.type == WRITE)
          write_to_buffer(thread, op.var, op.val);
        if (op.type == READ)
          read_buffer_or_memory(thread, op.var);
      }
    case FLUSH_BUFFER:
      if (!(buffer[thread].empty()))
        flush_buffer(thread);
  }
}
assert(final_state);
```

Listing 5.3: The pseudocode of the mechanised x86 memory model

describing the litmus test is initialised in the `thread_ops` queue of C structures and Listing 5.1 shows the structure of a thread operation. Whenever a non-deterministic value is required, we use the `choose` function. This is only a placeholder function and will be replaced by the corresponding API of a program analysis tool. We first use this function at the beginning of the simulation, for the number of simulation steps since we don't know many steps are required for each litmus test. Afterwards, the main loop of the program should ensure that the program runs for that specific number. At each loop of the simulation, the `choose` function will be invoked again to select the `action` that will be performed by a `thread`. The possible actions that the system can make are enumerated in Listing 5.2.

Each case-statement corresponds to an operational semantic rule and is guarded by an if-statement that verifies if the preconditions of the rule holds. If the `CPU_THREAD` action is chosen, the simulator will first check if there are any thread operations left for that specific thread, remove the next one from the `thread_ops` queue and attempt to process it. If this is a read operation, the x86 simulator will search for that value in the thread buffer and if it does not find it there, will search for it in main memory. Correspondingly, if the operation is a write, the simulator will add this operation to the store buffer. If instead a `FLUSH_BUFFER` action is chosen and the buffer is not empty, the simulator will transfer the data from the buffer to main memory.

It is possible to commit to an action (via the switch) before realising that its guard does not actually hold. In such a case, nothing would happen on this iteration of the for-loop. At the end of the execution, we check whether the program has reached the state that the litmus test describes. By exploring all the possible combinations of `thread` and `action` allowed values, the simulation will explore all the possible outcomes, given the `thread_ops` provided as input.

Generalisation We can generalise the structure of an operational model with the following general structure:

1. *Define opcodes, operation structure and actions.* Different systems will have different operation types and will perform different types of actions.

2. *Initialise all memory model components.* In our example, this means having store buffers be initially empty. In a more complex memory model there might be more complex components such as pools or caches.

3. *Initialising a per-thread queue of instructions.* At this point, the simulator should interface with the per-test harness so that it can initialise the queue of instructions.

4. *Write the state machine as a loop.* At each simulation step, the deployed tool will decide if the machine will consume user input or if one of the components of the machine will perform a transition. If the component has multiple choices (such as being able to flush different elements), the tool will decide which choice to explore.

5. *Check if the assertion holds.* Here the simulator should again interface with the per-test harness so that it can check if the re-orderings described by it occurred.

We are unlikely to have access to all the microarchitectural features of the system. Therefore, we cannot know the exact size of all the buffers in the model. However, we can choose sizes for these buffers that are large enough to allow all reorderings that the system is capable of performing, but no larger, so as not to add any unnecessary burden on simulation. Even though this kind of simulation is limited by its bounded nature, we can empirically verify that the bounds are sufficiently large for our litmus tests. We do this by adding `assert` statements within our model that verify if the buffers ever reach their total capacity and limit the paths available through the program.

Having described this general recipe for extending the model, we show in Section 5.3 how we can put it into practice for modelling a more intricate model for a combined CPU/FPGA system.

5.1.2 Designing the per-test harness

Reasoning about a litmus test involves combining the model with a test-specific harness. Each test-specific-harness will describe the sequence of operations for each concurrent component of the system, check if the simulation has run for a sufficient number of steps and if the reordering has been reached and finally, assert whether a particular outcome is observable.

There are multiple ways of encoding the litmus test, and the choice of encoding may affect the scalability of the program analysis tool that is subsequently applied to the resulting program. An important dimension for consideration here is branching: a piece of code that exhibits a lot of branching can be well-suited for coverage-guided fuzzing, as the branching will lead to many different coverage targets, which can help to identify diverse inputs to the program. In contrast, for symbolic tools, excessive branching can either lead to path explosion (in the case of a tool such as KLEE, that generates a seperate SMT query per path), or to a formula that involves a lot of disjunction (in the case of a tool such as CBMC that encodes an unwound version of the whole program as a single formula). Listing 5.4 shows an encoding designed to exhibit

```
1  check_litmus_test(){
2    // Check if final state reached
3    if (writesExecuted < totalWrites) return 0;
4    if (readsExecuted < totalReads) return 0;
5
6    // Check if the expected order was reached
7    if (timeEvent[0] < timeEvent[1]) return 0;
8    ...
9    if (timeEvent[n-1] < timeEvent[n]) return 0;
10
11   // Assert if register values are as expected
12   assert(reg[0]==a && ...  && reg[m] ==z);
13 }
```

Listing 5.4: Encoding the litmus test to favour high coverage.

```
1  check_litmus_test(){
2    // Check if final state reached
3    int reached = 1;
4    reached &= (writesExecuted == totalWrites);
5    reached &= (readsExecuted == totalReads);
6
7    // Check if the expected order was reached
8    int observed = 1;
9    observed &= (timeEvent[0] < timeEvent[1]);
10   ...
11   observed &= (timeEvent[n-1] < timeEvent[n]);
12
13   // Assert if register values are as expected
14   assert(reached & observed &
15           reg[0]==a & ...  & reg[m]==z);
16 }
```

Listing 5.5: Encoding the litmus test for fewer paths

more branching, while Listing 5.5 shows an encoding designed to exhibit less branching. In both cases, a set of preconditions must ensure that the program has executed enough simulation steps and correctly models the test. Lines 2-4 in Listing 5.4 and lines 3-5 in Listing 5.5 check if the executed writes and reads (described by the `writesExecuted` and `readsExecuted` variables) are equal to the writes and reads in the litmus test (described by the `totalWrites` and `totalReads` variables). More complex models might require additional checks, such as verifying that buffers are empty. Furthermore, lines 6-9 in Listing 5.4 and lines 7-11 in Listing 5.5 ensure that the events in the litmus test, (recorded in the `timeEvent` array) have occurred in the expected

order. The first encoding will immediately return when one of these preconditions is not met, while the second encoding will set the *reached* and *observed* variables to record the status of all preconditions.

Whenever a read operation is performed, its results are stored in the `reg` array. We can now assert if the operations observed by our model in the register do not contradict the litmus test. While Listing 5.4 will simply assert on the values observed in the registers, Listing 5.5 will only assert if the *reached* and *observed* variables have been set accordingly. Since the short-circuiting `&&` operation will only evaluate its second operand if the first one is true, while the `&` operator will evaluate all terms regardless. We use the `&&` in Listing 5.4 to create additional coverage points and `&` in Listing 5.5 to create fewer branching points.

We expect different tools to favour different encoding types and explore these options in Section 5.3.3 by automatically generating litmus tests using the two alternative options.

5.1.3 Using C Analysis Tools to Simulate Memory Models

The *choose* function from Listing 5.3 shows all the points in the C program where the non-determinism needs exploring, and we can plug in an analyser. To do so, we need to furnish the analyser with a means of exploring this non-determinism. Here, we describe the peculiarities of each approach and how the model needs to adjust for each specific tool.

CBMC-based Validation CBMC [CKL04] is a bounded model checker for C and C++ programs. CBMC is purely symbolic, yielding a single SAT or SMT formula encoding all executions of a program up to a certain depth. It can potentially scale poorly due to the formula getting large, leading to long SAT/SMT solving times. On the other hand, solving this big formula is all that needs to be done. Since the formula that CBMC constructs encodes *all* paths through the program up to the given depth, CBMC can be used to verify conclusively that a given scenario is *not* possible for execution traces below a certain length. Furthermore, as detailed below, we can establish conditions under which we guarantee unrolling deeply enough that all paths are considered. We chose CBMC as our model checker because it is widely used,

robust and practical and is well suited for system-level modelling.

CBMC will symbolically unwind the main simulation loop up to a certain *loop-unwind depth*, which is given as a parameter to the program. The final state may never be reached if the unwind depth is not sufficiently high enough. We can place *assert* statements to empirically verify if certain intermediate states have been reached. Furthermore, we invoke CBMC with the `--unwinding-assertions` option, whereby it checks that unwinding the program further does not lead to any more states being explored. In this mode, CBMC can *prove* that the program under test is free from assertion failures: if an insufficiently large unwinding depth for loops is used then an "unwinding assertion" fails, indicating that a higher bound is required for the proof to succeed. When using CBMC for memory model analysis, we use per-litmus test information to estimate a suitable unwinding depth, and use a script to iteratively increase this depth while it proves to be too low.

The operational semantic rule triggered at each point in the simulation is chosen non-deterministically. The premises of the operational rules are implemented using `assume` statements. If the premises of the rule are met, the operational rule can be executed and the state of the system updated. However, if the premises are not met, CBMC will update the query it sends to the SAT/SMT solver to mark that the path is not feasible.

CBMC for x86 Recall the code snippet from Listing 5.3 and note the updates in Listing 5.6 required for CBMC. The CBMC version of this model utilises a `nondet_int()` statement to allow the simulation to execute a nondeterministic number of steps. Furthermore, the `thread` and `action` is similarly chosen non-deterministically by corresponding statements. `assume` statements are utilised to guarantee that the premises of the semantic are met and infeasible paths terminated. The final `assert` statement verifies if the reordering has been detected.

KLEE-based Validation KLEE [CDE08] is a dynamic symbolic execution engine built on top of the LLVM compiler infrastructure. It encodes every path of the program separately in a symbolic manner, invoking an SMT solver to determine whether a path is feasible each time a conditional statement or assertion condition is reached. KLEE has the potential to scale better

```
// Non-deterministic number of simulation steps
int sim_steps = nondet_int();
for (int i = 0; i < sim_steps; i++) {
  // Non-deterministic thread choice
  int thread = nondet_thread(NUM_THREADS);
  // Non-deterministic action choice
  Action action = nondet_action(NUM_ACTIONS);
  switch (action) {
    case CPU_THREAD:
      assume(!thread_ops[thread].empty());
      Operation op = thread_ops[thread].pop();
      if (op.type == WRITE)
        write_to_buffer(thread, op.var, op.val);
      if (op.type == READ)
        read_buffer_or_memory(thread, op.var);
    case FLUSH_BUFFER:
      assume(!buffer[thread].empty());
      flush_buffer(thread);
  }
}
assert(final state);
```

Listing 5.6: The CBMC adaptation of the x86 memory model

than CBMC since solving formulas on a per-path basis might be easier than solving a single large formula encoding all program paths. Like CBMC, KLEE can verify that a scenario is not feasible if the associated program has a finite number of paths (though the *path explosion* problem means this is not always feasible in practice).

KLEE can be configured to exit when it encounters a specific type of error and record the test case required to reach it. Given our use case, we configure it to exit when an assertion failure is encountered. We utilise the `klee_make_symbolic()` command to mark the variables where we require non-determinism. KLEE will try different values of these variables to increase code coverage. We kill paths with unsatisfied premises by marking them with `klee_silent_exit()`.

In contrast with CBMC, KLEE utilises coverage information and, therefore, will prioritise generating test cases for paths that cover new code. This prioritisation means that KLEE has the potential to uncover paths that lead to assertion failures faster and terminate sooner. This means that the program transformation defined in Section 5.1.1 have the potential to aid the execution of the KLEE based simulation.

```
// Make number of simulation steps symbolic
klee_make_symbolic(sim_steps, sizeof(sim_steps));
for (int i = 0; i < sim_steps; i++) {
  // Make current thread symbolic
  klee_make_symbolic(thread, sizeof(thread));
  // Make current action symbolic
  klee_make_symbolic(action, sizeof(action));
  switch (action) {
    case CPU_THREAD:
      if (thread_ops[thread].empty()) klee_silent_exit();
      Operation op = thread_ops[thread].pop();
      if (op.type == WRITE)
        write_to_buffer(thread, op.var, op.val);
      if (op.type == READ)
        read_buffer_or_memory(thread, op.var);
    case FLUSH_BUFFER:
      if (buffer[thread].empty()) klee_silent_exit();
        flush_buffer(thread);
  }
}
assert(final_state);
```

Listing 5.7: The KLEE adaptation of the x86 memory model

KLEE for x86 Recall the code snippet from Listing 5.3 and note the updates in Listing 5.7 required for KLEE. The `thread` and `action` variables are declared symbolic and chosen at each simulation step. If the premises of the semantics are not meet, the simulation will exit using `klee_silent_exit()` statements. After the end of each simulation, the conditions that check if the assertion holds are verified.

libFuzzer-based Validation libFuzzer [Ser22] is an in-process library for coverage-guided fuzz testing. Coverage-guided fuzzing consists of generating test cases, monitoring their effect on the target binary's execution and updating the list of tests to increase coverage. It can be used to prove that a given behaviour is allowed, but because it is completely dynamic and does not keep track of the paths that it has already explored, it cannot be used to prove that a given behaviour is *not* allowed. However, this approach does not require expensive SAT queries and can quickly execute the program binaries with many different inputs and monitor for potential errors. It has the potential to quickly find counter-examples, but can also miss paths that lead to interesting behaviours. The fuzzer instruments the code to keep track of the areas of the

```
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
  // Additional checks to determine if size is not too small
  action_list = data[MAX_ACTIONS];
  thread_list = *data + MAX_ACTIONS;
  for (int i = 0; i < MAX_ACTIONS; i++){
    int thread = thread_list[thread_index++] % THREAD_COUNT;
    Action action = action_list[action_index++] % ACTION_COUNT;
    switch (action){
      case CPU_THREAD:
        if (thread_ops[thread].empty()) continue();
        Operation op = thread_ops[thread].pop();
        if (op.type == WRITE)
          write_to_buffer(thread, op.var, op.val);
        if (op.type == READ)
          read_buffer_or_memory(thread, op.var);
      case FLUSH_BUFFER:
        if (buffer[thread].empty()) continue();
          flush_buffer(thread);
    }
  }
  assert(final_state);}
```

Listing 5.8: The libFuzzer adaptation of the x86 memory model

code that it has already explored. It utilises this information to generate mutations on the corpus of inputs to maximise the coverage.

libFuzzer provides an array of bytes that can be used as input. We manipulate the array such that it matches the input of the system and use it whenever a source of non-determinism is required. This involves splitting the array into separate sections and ensuring that these sections do not overlap. We perform a modulo operation on each one of these bytes so that they match the encoding that we require. If the input requires the code to perform an action that does not have its premises met, the simulation will simply ignore that input and move to the next one.

libFuzzer for x86 Recall the code snippet from Listing 5.3 and note the updates in Listing 5.8 required for libFuzzer. The fuzzer provides as input to the system an array called `data` and its `size`. We partition the `data` array into two arrays: one that keeps the list of actions and one that keeps the list of threads. However, before this can actually be done, we have to ensure

Table 5.1: A comparison of the tools available, their underlying technique, memory model implementation, potential to validate disallowed behaviours and their utilisation of coverage information.

Tool	Technique	Model	Exhaustive	Guided by coverage
RMEM [ABC+22]	enumeration	Lem	✓	✗
RMEM [ABC+22]	random	Lem	✗	✗
Naïve	fuzzing	C	✗	✗
CBMC [CKL04]	SAT	C	✓	✗
KLEE [CDE08]	SAT	C	✓	✓
libFuzzer [Ser22]	fuzzing	C	✗	✓

that the `size` of the `data` is large enough to hold our statically assigned maximum number of transitions. We use simple if-statements to verify if the premises of an action are valid and if it is not, we skip it.

5.2 First case study: x86

We now present and discuss our first case-study, relating our results to research questions **RQ1** and **RQ2**. By implementing the x86 memory model according to the outline presented in Section 5.1.1, and using the CBMC, KLEE and libFuzzer tools described in Section 5.1.3, we can determine their viability in uncovering weak behaviours and their relative effectiveness to a state-of-the-art simulator, RMEM.

We run our experiments on an Intel Xeon CPU E5-2640 with 32GB RAM, under Ubuntu 20.04. We use RMEM version 0.1, CBMC version 5.11, KLEE version 5.11, and the libFuzzer deployed with clang 12. The SAT engine used by KLEE is STP version 2.3.3 and the one used by CBMC is MiniSat version 2.2.1. In pilot experiments with different solvers we observed slight differences in execution times but not significant enough to impact the comparisons between the tools. For this reason, for each tool, we use the default solver.

Table 5.1 summarises the tools that we experiment with, alongside some of their characteristics. We start our experiments by exploring the strengths of RMEM which can be considered the state-of-the-art tool at the moment and implements the memory model in a custom semantics language called Lem [MOG+14]. RMEM can run in either exhaustive mode or random mode

init: $x_0 = x_1 = \ldots = x_{n-1} = 0$

1 $x_0 \leftarrow 1$ 2 $r_0 \leftarrow x_1$	1 $x_1 \leftarrow 1$ 2 $r_1 \leftarrow x_2$	$\ldots$	1 $x_{n-2} \leftarrow 1$ 2 $r_{n-2} \leftarrow x_{n-1}$	1 $x_{n-1} \leftarrow 1$ 2 $r_{n-1} \leftarrow x_0$

Allowed: $r_0 = 0$ and $r_1 = 0$ and $\ldots$ and $r_{n-1} = 0$

(a) Store buffering using n threads

init: $x_0 = x_1 = \ldots = x_{n-1} = 0$

1 $r_0 \leftarrow x_1$ 2 $x_0 \leftarrow 1$	1 $r_1 \leftarrow x_2$ 1 $x_1 \leftarrow 1$	$\ldots$	1 $r_{n-2} \leftarrow x_{n-1}$ 2 $x_{n-2} \leftarrow 1$	1 $r_{n-1} \leftarrow x_0$ 2 $x_{n-1} \leftarrow 1$

Allowed: $r_0 = 1$ and $r_1 = 1$ and $\ldots$ and $r_{n-1} = 1$

(b) Load buffering using n threads

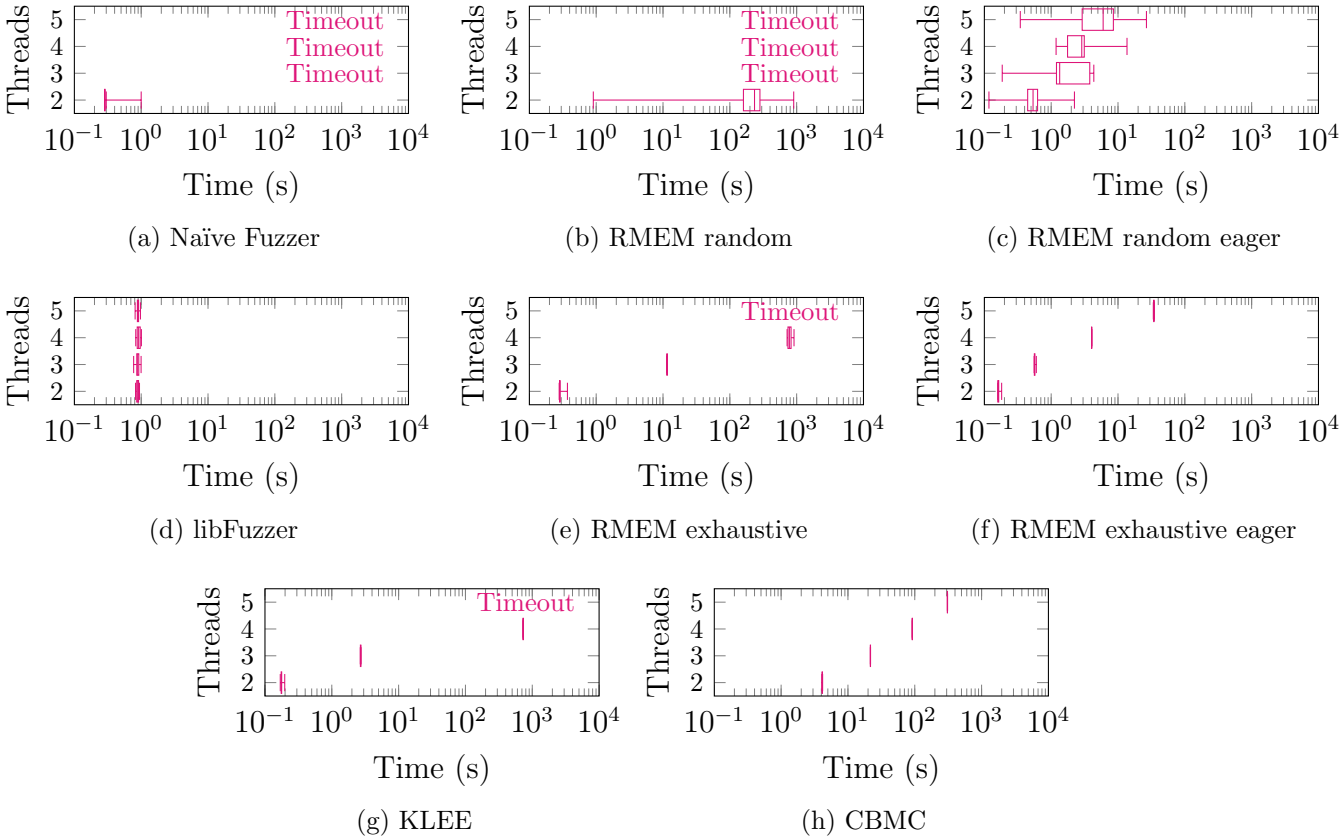
Figure 5.1: Store buffering and load buffering for n threads.

but can only prove that a behaviour is disallowed when set for exhaustive mode. Unfortunately, RMEM does not stop when it uncovers the behaviour of interest, as do all the other tools at our disposal. This means that RMEM does not report that a behaviour is allowed until it has finished exhaustive exploration, and we found that it was not trivial to temporarily modify the tool to change this.

As a sanity check, for our C models, we first utilise a naïve fuzzer, which randomly explores paths through the program's execution, similar to how the random version of RMEM does. Afterwards, we focus on the off-the-shelf CBMC, KLEE and libFuzzer tools. All tools have the potential to uncover allowed behaviours, but only RMEM, CBMC and KLEE can prove that a behaviour is not possible (because they consider all executions). Out of all the tools, only libFuzzer and KLEE use coverage information to guide their search.

To benchmark our approach, we can utilise litmus tests that generalise to any number of threads. In Figure 5.1, we present the generalisation of two such tests. In Figure 5.1a we have the *allowed* store-buffering litmus test while in Figure 5.1b we have the *disallowed* load-buffering litmus test. These tests enable us to vary the number of threads and observe how the different simulators handle them.

Given the C implementation of the x86 model, the set of program analysis tools and the litmus test, we can recall and answer our first research question:



(a) Naïve Fuzzer (b) RMEM random (c) RMEM random eager

(d) libFuzzer (e) RMEM exhaustive (f) RMEM exhaustive eager

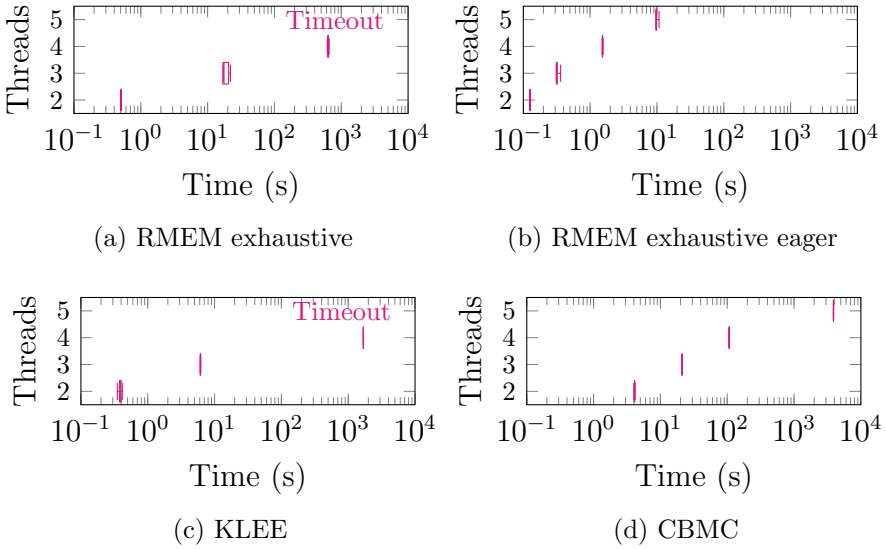
(g) KLEE (h) CBMC

Figure 5.2: Analysis times for an *allowed* litmus test (store buffering) using all tools.

RQ1 Can reducing the problem of memory model simulation to the analysis of a C program yield competitive performance compared with bespoke simulators?

We take each litmus test for different number of threads and try to determine if the reordering they describe is possible using all the tools at our disposal. We run each test/simulator combination ten times and create a box plot of the uncovered executions. For each one of the executions, we set a maximum execution time of 2 hours, and if, after this amount of time, the program still does not terminate, we stop it and report this as a timeout. Figure 5.2 shows the time in seconds required to uncover the reorderings of the store buffering litmus test and in Figure 5.3 we have the time required to prove that the load-buffering effect is not possible.

We have observed that RMEM has many possible options that we can optionally enable during simulation. Out of all these options, we have observed that one option, described as "eagerly take transitions that do not affect observable states", has the highest impact on performance. Unfortunately, there is insufficient documentation to clarify why this option is not enabled by



(a) RMEM exhaustive (b) RMEM exhaustive eager

(c) KLEE (d) CBMC

Figure 5.3: Analysis times for a *disallowed* litmus test (load buffering) using those tools that are capable of exhaustive search.

default and the possible cost of utilising it. We are in contact with the developers to clarify this and we have decided to include results with and without this option enabled for completion. While specific optimisations can enable RMEM to scale better than KLEE and CBMC, it is still outperformed by libFuzzer for large number of threads. Furthermore, the out-of-the-box tools do not require any extra engineering effort.

The naïve fuzzer was only able to uncover the trivial executions and faired poorly in all other cases. RMEM was able to handle all the test cases we have provided and fairer significantly better in the exhaustive mode but only when the option to eagerly take transitions was enabled. KLEE unfortunately timed out for larger executions, and while CBMC was able to handle all of them, it was at a significant performance overhead. However, out of all the tools we explored, libFuzzer was the fastest one.

Finding from RQ1. Our approach of reducing the problem of memory model simulation to the analysis of a C program yields competitive results, in some cases even scaling better than RMEM. The wide range of executions time we have uncovered, indicate that the search strategy is the main factor that dictates the effectiveness of the approach.

Having seen that our approach of leveraging off-the-shelf tools does yield competitive results,

we move to **RQ2**, which can further be subdivided as follows:

RQ2 Of the variety of C analysis tools that are available, which are most effective for memory model simulation?

(a) How is performance influenced by the test case size under simulation?

(b) How is performance influenced by whether the behaviour associated with the test-case is allowed according to the memory model?

Regarding **RQ2a**, the small litmus tests that feature only two threads are solved by all techniques. However, the more heavyweight SAT-based analysis performed by CBMC puts it as a disadvantage, due to the overhead of solving a SAT query reflecting a fully-unwound program. However, we can observe that the simpler methods are not always capable of uncovering the complex executions involving more threads. The naïve fuzzer is not able to handle tests featuring more than two threads, and KLEE does not scale to the five-threaded case. In this set of experiments, the only tool that is not significantly affected by the size of the litmus test is libFuzzer. However, we show in Section 5.3.4 that the execution time associated with libFuzzer does increase when we apply it to a more complex memory model with larger associated litmus tests.

Finding from RQ2a. While it makes little difference which analysis tool we deploy for small litmus tests, that is no longer the case for larger ones. In such cases, it is advisable to use libFuzzer and CBMC. While libFuzzer has been shown to be extremely effective at discovering allowed executions, unlike CBMC it cannot show that executions are disallowed. In a context where the allowed/disallowed status of a litmus test is not known, running both techniques in parallel would be advisable.

Since not all tools can prove that a behaviour is not allowed, we can only answer **RQ2b** for the ones that do. The tools have to consider all execution paths, and as a result, they require more time to do so. Having to explore all paths means that coverage information does not help. If we consider only off-the-shelf tools, CBMC is the best choice.

Finding from RQ2b. Although a heavyweight solution for simple test cases, the SAT-based approach of CBMC pays off for larger, more complex test cases when behaviours are *disallowed*. The coverage-guided searech of KLEE is not useful in this context, since the disallowed nature of the test case means that there is no possibility of an early exit: all executions will eventually need to be considered.

Summary libFuzzer is surprisingly effective at uncovering allowed executions, proving to be a better choice that the other tools that we have analysed. On the other hand, CBMC is better at proving large disallowed executions. Armed with this new insight, we now explore a more recent and challenging memory model for which a bespoke simulator does not exist, to get more insight into its behaviour.

5.3 Second case study: CPU/FPGA

We now describe our experience applying the previously-described techniques to a more complex model for a hybrid CPU/FPGA system. After introducing the system (Section 5.3.1) we present our experimental setup (Section 5.3.2), our analysis of the impact of the test harness (Section 5.3.3), scalability assessment (Section 5.3.4), and conclude by discussing the infidelities in the X+F model that we found and fixed thanks to our approach (Section 5.3.5).

5.3.1 The X+F memory model

A recent trend in heterogeneous systems is to combine a multicore CPU with a field-programmable gate array (FPGA). These hybrid CPU/FPGA systems are of particular interest because the FPGA can be customised for a specific computationally-intensive sub-task, while the overall application is coordinated by the general-purpose CPU. We focus on the Xeon+FPGA (X+F) memory model [IDSW21b] that was originally analysed using only CBMC for a small number of memory operations.

The memory model is illustrated graphically in Figure 2.3. Conceptually, the FPGA is a separate thread that runs alongside the CPU threads. In contrast to CPU threads, which issue atomic reads, writes and fences, the FPGA breaks this operations down into *requests* and *responses*. For instance, to write to memory the FPGA issues a *write request*, containing the location and value to be written. Later, if the FPGA receives a *write response* this guarantees that the write request has entered the FPGA's memory subsystem but does *not* guarantee that it has been committed to main memory. A *fence request* can be issued to indicate that writes should be propagated to main memory, and this propagation is only guaranteed to have occurred when a corresponding *fence response* is received.

The FPGA's memory subsystem is composed of *request pools*, *upstream buffers* and *downstream buffers* as shown in Figure 2.3. *Requests* and *responses* have to travel through all this components when travelling between the FPGA thread and he main memory.

We can implement the C model of the X+F system by extending the code of the x86 system from Section 5.1.1. Extending it involves adding opcodes for the FPGA thread, channel identifiers to the structure in Listing 5.1, and actions to the enumeration in Listing 5.2. The actions should correspond to the FPGA thread, *request pools*, *upstream buffers* and *downstream buffers*. Each of these elements has a separate data structure that takes into account its ordering guarantees (e.g. FIFO order for the upstream buffer)

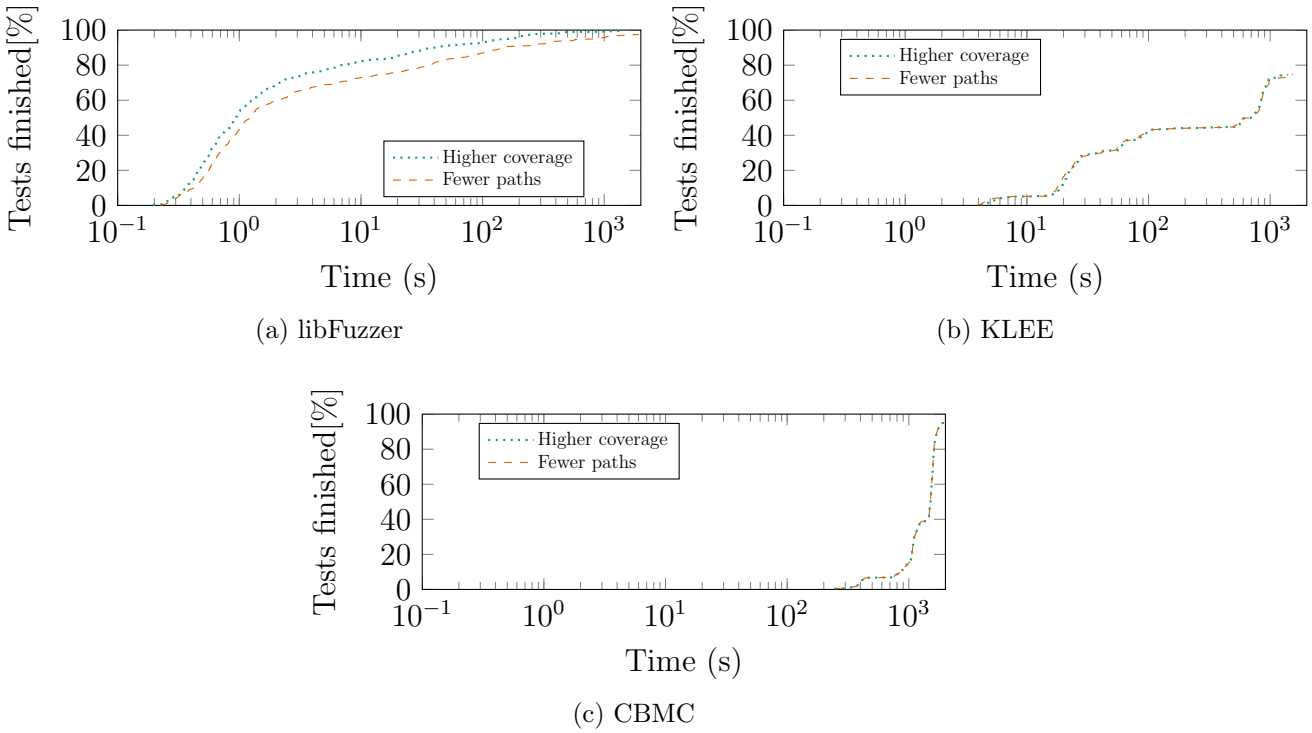
5.3.2 Experimental setup

We reuse the same configuration described in Section 5.2. Similar to the approach described in the original work of the X+F model [IDSW21b], we generate allowed executions of different sizes from the axiomatic model and run them on the operational model. The size of the litmus test is described by the number of events it contains and these events represent different operations of the CPU or FPGA.

Since there is a large number of executions that can be generated, we can limit our exploration only to more challenging and non-trivial executions. Therefore, we generate disallowed exe-

cutions where each event is critical as described in previous work [LWPG17]. By removing a fence operation from this executions, we can create allowed non-trivial executions.

5.3.3 Impact of test harness



(a) libFuzzer

(b) KLEE

(c) CBMC

Figure 5.4: The percentage of executions uncovered under a certain amount of time using the two alternative encodings for the litmus tests.

We start by adapting the test generator to enable the alternative options described in Section 5.1.2. These options involve encoding the litmus test for more coverage points (Listing 5.4) or fewer program paths (Listing 5.5). The alternative versions of the litmus test can be sent to the different tools to evaluate their impact on performance. We generate traces with events ranging from 6 up to 9.

We can now recall our research question:

RQ3 How does the encoding of the litmus test impact the performance of the different tools?

Figure 5.4 shows how the encoding affects the different tools. The graphs show the percentage

of executions uncovered under a certain amount of time. We have set a limit of 2000 seconds for each execution and ran them using the different encodings and tools.

We can see that libFuzzer takes advantage of the coverage points provided by the first encoding of the litmus tests. Furthermore, we can corroborate these results with the ones from Section 5.2 where we have seen the poor performance of the naïve fuzzer. This result highlights the importance of coverage in fuzzing tools in uncovering the transitions that expose the behaviour needed by the litmus test.

KLEE is not significantly affected by the encoding. This indicates that while the high-coverage version of the litmus test creates more SAT queries, these are easier for the solver to solve. CBMC similarly is not significantly impacted by the encoding utilised. Moreover, we can notice a stairway pattern in its execution time. This pattern results from the different number of events in the executions.

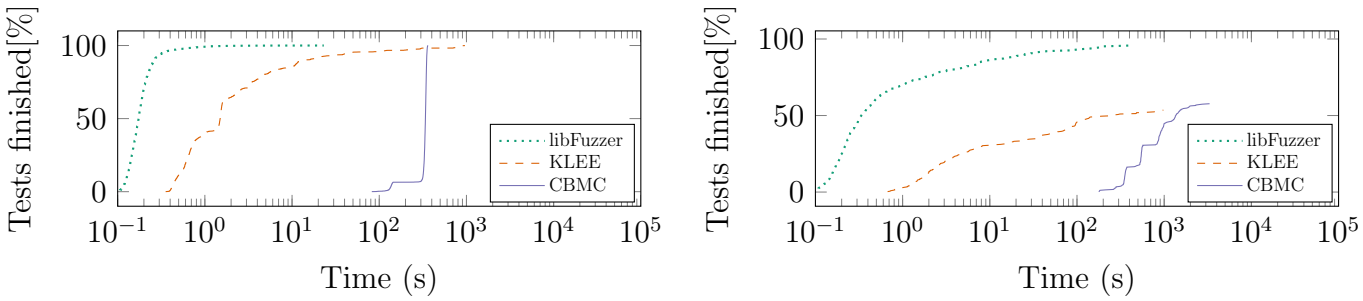
Finding from RQ3. libFuzzer is sensitive to the encoding of the litmus test and will benefit from the extra coverage points. However, KLEE and CBMC will not be substantially affected by the encoding of the tests.

5.3.4 Scaling up simulation

Section 5.3.3 shows that encoding the litmus tests for higher coverage benefits all of the tools. We can use this to further explore the scalability of the tools. We can now recall our research question:

RQ4 Can our approach allow more in-depth analysis of the X+F memory model, allowing it to be better validated against its axiomatic counterpart?

We start with small executions, with few thread operations that should not challenge any of the simulators. Therefore, we limit each execution to a maximum of five events, which allows us to generate a total number of 2481 allowed executions. To ensure that the tools do not hang, we



(a) small executions

(b) large executions

Figure 5.5: The percentage of executions uncovered under a certain amount of time using different tools. All tools were able to uncover the allowed executions in (a). However, when scaling the number of events in (b) this was no longer the case: out of 350 executions, libFuzzer uncovered 335 executions, CBMC uncovered 202, and KLEE uncovered only 189.

limit the total execution time to 500 seconds. After verifying that the performance of the tools translates to this model, we generate executions with more operations to evaluate scalability. We generate 50 executions for each thread size from 6 up to 16 operations, summing up to 350 executions. We also increase the timeout from 500 seconds to 2000 seconds.

Figure 5.5 shows the percentage of reordering that was proven allowed after a certain amount of time. For a small number of events, all tools were able to find the path through the program that led to the reordering required by all 2481 executions. However, this is no longer the case when scaling the number of events. Out of 350 executions, libFuzzer uncovered 335 executions, CBMC uncovered 202 and KLEE uncovered only 189.

libFuzzer outperforms both CBMC and KLEE, managing to uncover the path that leads to the reorderings faster than the other tools. The time required by CBMC depends on the test size and explains the stair-like pattern that we see in both figures. However, despite its slow execution, CBMC never got stuck like libFuzzer or KLEE and was the only tool that managed to uncover some of the more challenging executions. It can thus be considered the slowest yet most reliable tool.

Our strategy for fast model evaluation. We can assume that every given set of test-cases will be composed of challenging and easy executions. Our experiments show that libFuzzer can uncover easy executions quite quickly but fails with some of the more challenging ones.

On the other hand, CBMC can uncover any executions, regardless of difficulty, but requires a significant amount of time to do so. We can therefore come up with a strategy where for any given set of executions, we can first run libFuzzer to uncover the easy ones and only utilise CBMC for the more challenging ones. We will see in Section 5.3.5 how this technique has allowed us to fix some infidelities in the model.

Finding from RQ4. Deploying both libFuzzer and CBMC enables feasible exploration of the model for a considerably higher number of events and has allowed us to uncover some infidelities in the original model.

Tool limitation. For some executions, KLEE is very fast, while for others, it never terminates even after more than 24 hours. We sent sample test cases to the developers, and they informed us that these cases revealed a limitation of the tool. KLEE compares each new query with previous solved ones to reduce execution time. However, some executions create large expressions that KLEE is unable to evaluate, leading to it getting stuck. The developers are currently working on a fix.

5.3.5 Fixing the CPU/FPGA model

Throughout our experiments, we used the artefact from the work that describes the X+F memory model [IDSW21a] to generate executions that we can send to the different versions of the operational model. In the original work, experiments were run with executions with up to 8 events, but due to the increased scalability that libFuzzer offers, we were able to scale up to 16 events successfully for allowed events. These traces were significantly more complex than the original ones and allowed us to explore the model more deeply. While most executions were proven allowed by the fuzzer as expected, some were not. After also running them with CBMC and then finally manually inspecting them, we realised that these reorderings were not possible. Thus, we discovered some infidelities in the original model that we were able to fix.

As a result of this work, four axioms in the original work have been modified. For completeness,

acyclic$((\texttt{poloc} \cup \texttt{rf} \cup \texttt{fr} \cup \texttt{co}) \cap \texttt{CPU}^2)$	SC-PER-LOC
acyclic$(\texttt{ppo} \cup \texttt{fence} \cup \texttt{rfe} \cup \texttt{fre} \cup \texttt{co})$	PROPAGATION
irreflexive$(\texttt{fr} ; \texttt{poch} ; \texttt{readpair})$	READ-AFTER-WRITE
irreflexive$(\texttt{fr} ; \texttt{poFnRsp} ; \texttt{po} ; \texttt{readpair})$	READ-AFTER-FENCE
irreflexive$(\texttt{rf} ; \texttt{po})$	NO-READ-FROM-FUTURE
acyclic$(\texttt{fre} \cup \texttt{rfe} \cup (\texttt{rf} \setminus \texttt{sch}) \cup \texttt{poch} \cup \texttt{ppoCPU})$	OBSERVE-SAME-CHANNEL
irreflexive$(\texttt{po} ; \texttt{fencepair} ; \texttt{po} ; \texttt{writepair}^{-1})$	FENCE-RESPONSE
irreflexive$(\texttt{po} ; (\texttt{fencepair} \cup \texttt{writepair}) ; \texttt{po} ; \texttt{fenceallpair}^{-1})$	FENCE-BLOCK
irreflexive$(\texttt{rf} ; \texttt{poloc} ; \texttt{co})$	WRITE-ORDER-CHANNEL

Figure 5.6: The revised axioms of the X+F memory model, with modifications highlighted.

we list the complete set of corrected axioms in Figure 5.6. All of the relations mentioned in the axioms have the same definitions as in [IDSW21b], with the additional definition of `fencepair` as $\texttt{fenceonepair} \cup \texttt{fenceallpair}$.

The OBSERVE-SAME-CHANNEL was originally an irreflexivity axiom that prevented writes from different threads from being observed out-of-order on the same channel and now has been extended to disallow this behaviour for more types of events. The FENCE-RESPONSE and FENCE-BLOCK axioms were present in the original paper and describe the reorderings that fences enforce on the executions. These axioms have been extended to take into account additional restrictions that fences impose on operations. The WRITE-ORDER-CHANNEL is a new axiom, added to deal with additional ordering guarantees of multiple writes to the same channel.

5.4 Related work

Operational memory models have been proposed for widely used memory models including x86 [OSS09], POWER [SSA+11], ARM [PFD+17a, SFP+20] and RISC-V [PPPK+19], and **model checkers** have been widely used to simulate them. For instance, [AKNT12] showed how to transform the problem of verification under a weak memory model into the problem of verification under SC, by transforming each memory access in the program so that it explicitly manipulates a buffer rather than main memory directly. The verification-under-SC problem can

then be handled by an off-the-shelf model checker; [AKNT12] use CBMC, while [SB18] in later work use DIVINE. In both cases, the approach differs from ours because it relies on the design of program transformations that correspond to specific memory models (x86, PSO, RMO, and POWER), while we show how to encode the transition system of an arbitrary memory model directly in C. This makes our approach more suitable for a memory model that is still in active development, whose transition system may still be in flux, and which is not yet sufficiently well understood to be confidently translated into equivalent program transformations. Besides, it is not clear how some of the more complicated memory models, like the X+F model we considered, can be recast as program transformations; indeed, [LV16] have shown that some memory models, notably ARM's, *cannot* be explained solely using program transformations.

RMEM [ABC+22] is a **bespoke simulator** for a variety of operational memory models [SFP+20, ABC+19, PFD+17a]. It is widely used, but involves a substantial amount of engineering effort to implement the desired algorithms for exploring the models (e.g. randomly, exhaustively, or in a user-guided fashion). In our approach, we outsource the exploration task to a range of off-the-shelf tools instead. *Murphi* is another simulator for operational semantics that has been used to explore GPU memory models [SGG13] and also to verify heterogeneous cache coherency protocols [OGN+22], but again, it lacks the generality of our approach because it only supports the exploration algorithms that are hardcoded into it.

Axiomatic memory models are an alternative way of representing memory models, with numerous memory models implemented in the *herd* simulator [AMT14]. This simulator enables users to run litmus tests on top of axiomatic models. Since then, other alternatives based on axiomatic memory models have been shown effective for various other tasks. *Alloy* is an open source language and analyser for software modelling that has been used to simulate memory models [WBSC17]. The flexibility of this approach has allowed it to be configured such that only non-trivial executions are generated [LWPG17]. *CDSChecker* [ND13] is a model-checker for C++ programs that uses several techniques to minimise the number of executions behaviours that need exploring. *GenMC* [KV21] is a stateless model checker built on top of the LLVM infrastructure that can be efficiently utilised to verify C++ programs. While axiomatic memory model simulators such as these can be considered faster, they lack intuitive features of

operational ones, such as providing execution paths that lead to the reordering.

The idea of **problem reduction to a C program** has been explored in other domains; for instance, Verilator [Sny22] is a popular open-source Verilog simulator that works by translating a Verilog design into a C program that can then be executed or otherwise analysed [TSC+21].

The trinity of CBMC, KLEE and libFuzzer have been previously employed by [PZS+21], and shown to complement each other and uncover bugs in different applications. Moreover, coverage-guided fuzzing has been used as a model-exploration technique in domains where formal verification or symbolic reasoning techniques do not scale well, such as demonstrating the satisfiability of SMT formulas for floating-point arithmetic [LCDS19].

5.5 Summary

In this chapter, we have investigated how operational memory models can be simulated by reducing the decision problem of "whether a given operational model allows a given program behaviour" to the decision problem of "whether a given C program is safe", which can be handled by a variety of off-the-shelf tools. This technique has allowed us to evaluate three such tools: a model checker (CBMC), a symbolic analysis tool (KLEE) and a fuzzer (libFuzzer), comparing them to a bespoke simulator (RMEM).

Our main finding is that coverage-guided fuzzing, implemented by libFuzzer, is extremely effective at confirming allowed behaviours for the litmus test configurations we consider, generally vastly outperforming both model checking and symbolic execution. Furthermore, we find that (a) the *coverage-guided* feature that libFuzzer offers is essential—we also present results using a naïve fuzzer that is not guided by coverage, which does not perform nearly as well as libFuzzer, and (b) the exhaustive and random approaches that RMEM utilises do not scale well enough to allow meaningful analysis of memory model litmus tests with a large number of events.

The key takeaway from our experience is that we highly recommend that researchers and engineers interested in operational memory model simulation consider our "reduction to C"

approach because it lifts the burden of implementing various analysis algorithms in a bespoke tool. Our experience is that the coverage-guided libFuzzer tool particularly shines when it comes to the fast analysis of allowed behaviours, while the symbolic CBMC tool is effective at the exhaustive exploration of reasonably large litmus tests. Our second case study, on a CPU/FPGA memory model, showcases the value of our approach by allowing a number of discrepancies in an axiomatic description of this memory model to be found and fixed.

Chapter 6

Conclusion

Here we present the summary of our findings in Section 6.1 and an overview of potential future work in Section 6.2.

6.1 Summary

This work presents the first formalisation of the memory model of a heterogeneous CPU/FPGA system. In such systems, the CPU and the FPGA can access the same shared memory in a fine-grained manner. While this provides the potential for applications to take advantage of the unique features of each component, it also creates the risk of incorrect synchronisation due to each component's particular weak memory effects. This thesis shows how one can reason about these complex systems in a formal manner.

Modelling Memory in CPU/FPGA Systems We provide a detailed formal case study of the memory semantics of Intel's CPU/FPGA systems. These combine a multicore Xeon CPU with an Intel FPGA, and allow them to share main memory through Intel's CCI-P [Int19]. To gain an understanding of the variety of complex behaviours that the system can exhibit, we have studied the available X+F documentation in detail and empirically investigated the

memory semantics of the FPGA device and of CPU/FPGA interactions using a real system that integrates a Broadwell Xeon CPU with an Arria 10 FPGA.

Based on our investigations, we present a formal semantics for the X+F memory system in two forms: an *operational* semantics that describes the X+F memory system using an abstract machine, and an *axiomatic* semantics that declaratively characterises the executions permitted by the memory system independently of any specific implementation. We have mechanised the operational semantics in C, in a form suitable for analysis with the CBMC model checker [CKL04]. This allows an engineer to explore the possible behaviours of a given memory model litmus test, and supports the generation of counterexamples that can be understood with respect to the abstract machine. The axiomatic semantics, meanwhile, has been mechanised in the Alloy modelling language [Jac12]. The Alloy Analyzer can then be used to automatically generate allowed or disallowed executions, subject to user-provided constraints on the desired number of events and actors that should feature in a generated execution.

Validating Our Models We have used the combination of our mechanised operational and axiomatic semantics, plus access to concrete X+F hardware, to thoroughly validate our models. Specifically, we have used the Alloy description of our axiomatic semantics to generate *disallowed* executions that feature only *critical* events (i.e. removing any event from an execution would make the execution allowed). Using a back-end that converts an execution into a corresponding C program, we have used these executions and the CBMC model checker to validate our operational model both 'from above' and 'from below'; that is, every disallowed execution generated from the axiomatic model is also disallowed by the operational model, and removing any event from such an execution causes it to become allowed by the operational model(due to the critical nature of events). This combination of a mechanised operational and axiomatic semantics allowed us to set up a virtuous cycle where we would cross-check the models using a batch of generated tests, find a discrepancy, confirm the correct behaviour by referring to the manual or discussing with an Intel engineer, refine our axioms or our operational model, and repeat. This back-and-forth process is a compelling demonstration of the value of developing operational and axiomatic models in concert, which we hope will inspire other researchers

to follow suit. The operational model that we reached by the end of this process has been described by an Intel Senior Principal Engineer as 'definitive'.

Having gained confidence in the accuracy of our models via this cross-checking process, we proceeded to run tests against hardware both to check that execution results disallowed by the model are indeed not observed (increasing confidence that our model is sound), and to see how often unusual-but-allowed executions are observed in practice. Since synthesising an FPGA design from Verilog takes several hours, performing synthesis on an execution-by-execution basis was out of the question. Instead, we present the design of a soft-core processor customised to execute litmus tests described using a simple instruction set. The processor is synthesised once, after which the CPU can send a series of tests to the FPGA for execution, allowing us to process hundreds of tests in a matter of hours, rather than weeks. We find that when we execute our tests on the hardware 1 million times each, the 583 disallowed outcomes are never observed, but some of the 180 allowed outcomes are. We run all tests in an environment that simulates heavy memory traffic, in the hope of coaxing out weak behaviours that may otherwise be unobservable.

Putting Our Models to Use To demonstrate the utility of our formal model, we use it to reason about a producer/consumer queue linking the CPU and the FPGA. We investigate various design choices for the queue, using our model to argue why they provide correct synchronisation, and we compare their performance. Then, guided by our model, we develop *lossy* versions of the queue that omit some synchronisation, risking loss or reordering of elements as a result, but in a well-defined manner that is described by our formal model. We present experimental results exploring the performance/quality trade-off associated with these queue variants, which is relevant in the context of application domains where some loss is tolerable, such as image processing and machine learning. We also show that lossy behaviour is exacerbated when the FPGA is configured to contain additional processing elements that stress the shared-memory system; these "enemy" components can serve as a debugging aid to help shake out bugs arising from missing synchronisation.

Simulating operational models Finally, our experiments have shown that simulating operational memory models is a bottleneck in developing such systems. Validating the system's behaviours using CBMC is slow and not scalable, limiting our ability to scale in terms of the size of tests and the number of tests. Therefore we have investigated alternative methods that can be used to gain insight into the behaviour of these systems quickly. We observed that writing these models in a language such as C and using off-the-shelve tools for program analysis provides valuable insight into the behaviour of these systems. For example, libFuzzerallowed us to quickly gain insight into the system's behaviour and fix some infidelities in the previously developed models.

6.2 Future Work

Heterogeneous systems such as the one investigated here are still relatively new and under constant development [CCF+16, CCF+19]. This fast evolution is an opportunity for researchers to help build these systems. We can identify the following research directions for such systems:

Generalising the model This thesis focuses on the Intel X+F system and provides rigorous semantics that help understand the system's behaviour. However, it does not answer how well these models generalise to other heterogeneous devices. Many industry developers have adopted the Compute-Express Link (CXL) standard for interconnecting devices, and it is expected that soon we will have heterogeneous devices powered by it. Furthermore, many academic projects focus on heterogeneous CPU/FPGA systems [CRS+22]. Given that the FPGA accelerator is usually connected to the CPU main memory via some network that resembles the Peripheral Component Interconnect Express (PCIe) one, we expect many similarities. However, these assumptions should be tested once the devices are available.

Improving the soft-core processor The soft-core processor enables us to quickly run many litmus tests without considering the prohibitive cost of synthesing each. However, the downside of using such an approach is that it adds a certain amount of overhead to the testing process that

could hide some of the weak behaviours of the systems. The different stages of the processor increase delay between the different instructions of the litmus test, a delay that can hide some of this behaviour. It could be helpful to think of a system where the litmus tests are stored in the memory of the FPGA, and a simple memory controller would be able to run with much lower overhead.

Synchronisation mechanisms Our work investigates the challenges of implementing a producer/consumer queue on such systems. However, multicore systems offer various synchronisation primitives that can be used for concurrent programming. It would be interesting to understand how many of these primitives can be used for such heterogeneous systems and which one of them is more effective. This research question can be more efficiently answered if we also consider different applications that run on such systems and how such primitives impact them.

Heterogeneous programming languages Future work can also evaluate the impact of these memory models on heterogeneous programming languages. For example, SYCL enables code for heterogeneous processors to be written using standard ISO C++ [Khr22]. Like most programming languages, SYCL offers different methods that can be used to communicate between the host and the accelerator. Therefore, it would be necessary to have a translation from the memory model it offers to the one that a heterogeneous CPU/FPGA system offers.

Bibliography

[ABC+19] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. ISA semantics for ARMv8-a, RISC-v, and CHERI-MIPS. *Proc. ACM Program. Lang.*, 3(POPL), jan 2019.

[ABC+22] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. *RMEM: A tool for exploring relaxed-memory concurrency.* Cambridge, 2022.

[ABD+15] Jade Alglave, Mark Batty, Alastair F. Donaldson, Ganesh Gopalakrishnan, Jeroen Ketema, Daniel Poetzl, Tyler Sorensen, and John Wickerson. GPU concurrency: Weak behaviours and programming assumptions. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 577–591, New York, NY, USA, 2015. Association for Computing Machinery.

[AG96a] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE COMPUTER*, 29:66–76, 1996.

[AG96b] S.V. Adve and K. Gharachorloo. Shared memory consistency models: a tutorial. *Computer*, 29(12):66–76, 1996.

[AKK+16] Maleen Abeydeera, Manupa Karunaratne, Geethan Karunaratne, Kalana De Silva, and Ajith Pasqual. 4K real-time HEVC decoder on an FPGA. *IEEE Transactions on Circuits and Systems for Video Technology*, 26(1):236–249, Jan 2016.

[AKNT12] Jade Alglave, Daniel Kroening, Vincent Nimal, and Michael Tautschnig. Software verification for weak memory via program transformation. *CoRR*, abs/1207.7264, 2012.

[AMSS10] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Fences in weak memory models. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *Computer Aided Verification*, pages 258–272, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[AMSS11] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Litmus: Running tests against hardware. In Parosh Aziz Abdulla and K. Rustan M. Leino, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 41–44, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[AMT14] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Trans. Program. Lang. Syst.*, 36(2), July 2014.

[BMO+12] Mark Batty, Kayvan Memarian, Scott Owens, Susmit Sarkar, and Peter Sewell. Clarifying and Compiling C/C++ Concurrency: From C++11 to POWER. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, page 509–520, New York, NY, USA, 2012. Association for Computing Machinery.

[BY19] M. Bechtel and H. Yun. Denial-of-Service Attacks on Shared Cache in Multicore: Analysis and Prevention. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 357–367, April 2019.

[CCF+16] Young-kyu Choi, Jason Cong, Zhenman Fang, Yuchen Hao, Glenn Reinman, and Peng Wei. A quantitative analysis on microarchitectures of modern CPU-

FPGA platforms. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2016.

[CCF+19] Young-Kyu Choi, Jason Cong, Zhenman Fang, Yuchen Hao, Glenn Reinman, and Peng Wei. In-Depth Analysis on Microarchitectures of Modern Heterogeneous CPU-FPGA Platforms. *ACM Trans. Reconfigurable Technol. Syst.*, 12(1), February 2019.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 209–224, USA, 2008. USENIX Association.

[CKL04] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176, Germany, 2004. Springer.

[CRS+22] David Cock, Abishek Ramdas, Daniel Schwyn, Michael Giardino, Adam Turowski, Zhenhao He, Nora Hossle, Dario Korolija, Melissa Licciardello, Kristina Martsenko, Reto Achermann, Gustavo Alonso, and Timothy Roscoe. Enzian: An Open, General, CPU/FPGA Platform for Systems Software Research. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2022, page 434–451, New York, NY, USA, 2022. Association for Computing Machinery.

[DS13] Roland Dobai and Lukas Sekanina. Image filter evolution on the Xilinx Zynq platform. In *2013 NASA/ESA Conference on Adaptive Hardware and Systems*, June 2013.

[FBL+16] Naila Farooqui, Rajkishore Barik, Brian T. Lewis, Tatiana Shpeisman, and Karsten Schwan. Affinity-Aware Work-Stealing for Integrated CPU-GPU Processors. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and*

Practice of Parallel Programming, PPoPP '16, New York, NY, USA, 2016. Association for Computing Machinery.

[FGP+16] Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. Modelling the ARMv8 Architecture, Operationally: Concurrency and ISA. *SIGPLAN Not.*, 51(1):608–621, jan 2016.

[Gha95] Kourosh Gharachorloo. Memory consistency models for shared-memory multiprocessors. Technical report, Stanford University, Stanford, CA, USA, 1995.

[GLL+19] Ce Guo, Wayne Luk, Stanley Qing Shui Loh, Alexander Warren, and Joshua Levine. Customisable control policy learning for robotics. In *2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, volume 2160-052X, pages 91–98, 2019.

[GSQ+18] Kaiyuan Guo, Lingzhi Sui, Jiantao Qiu, Jincheng Yu, Junbin Wang, Song Yao, Song Han, Yu Wang, and Huazhong Yang. Angel-Eye: A complete design flow for mapping CNN onto embedded FPGA. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(1):35–47, Jan 2018.

[HHB+14] Derek R. Hower, Blake A. Hechtman, Bradford M. Beckmann, Benedict R. Gaster, Mark D. Hill, Steven K. Reinhardt, and David A. Wood. Heterogeneous-race-free memory models. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 427–440, New York, NY, USA, 2014. Association for Computing Machinery.

[HKV98] Lisa Higham, Jalal Kawash, and Nathaly Verwaal. Weak Memory Consistency Models. Part I: Definitions and Comparisons. Technical report, Department of Computer Science, The University of Calgary, 1998.

[HP19] John L. Hennessy and David A. Patterson. A new golden age for computer architecture. *Commun. ACM*, 62(2):48–60, January 2019.

[HZS+18] Bo-Yuan Huang, Hongce Zhang, Pramod Subramanyan, Yakir Vizel, Aarti Gupta, and Sharad Malik. Instruction-level abstraction (ILA): A uniform specification for system-on-chip (soc) verification. *CoRR*, abs/1801.01114, 2018.

[IDSW21a] Dan Iorga, Alastair Donaldson, Tyler Sorensen, and John Wickerson. *The Semantics of Shared Memory in Intel CPU/FPGA*. Zenodo, September 2021.

[IDSW21b] Dan Iorga, Alastair F. Donaldson, Tyler Sorensen, and John Wickerson. The Semantics of Shared Memory in Intel CPU/FPGA Systems. *Proc. ACM Program. Lang.*, 5(OOPSLA), oct 2021.

[Int19] Intel. *Intel Acceleration Stack for Intel Xeon CPU with FPGAs Core Cache Interface (CCI-P) Reference Manual*, November 2019. Version 2019.11.04.

[Int21] Intel. Intel academic compute environment, April 2021.

[ISWD20] Dan Iorga, Tyler Sorensen, John Wickerson, and Alastair F. Donaldson. Slow and steady: Measuring and tuning multicore interference. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 200–212, 2020.

[IWD] Dan Iorga, John Wickerson, and Alastair F. Donaldson. Simulating operational memory models using off-the-shelf program analysis tools. Submitted to ASE 2022.

[Jac12] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2012.

[Khr22] Khronos. *SYCL overview*, July 2022.

[KLP+18] Ahmed Khawaja, Joshua Landgraf, Rohith Prakash, Michael Wei, Eric Schkufza, and Christopher J. Rossbach. Sharing, protection, and compatibility for reconfigurable fabric with AMORPHOS. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI 2018, pages 107–127, USA, 2018. USENIX Association.

[KSTM20] Jake Kirkham, Tyler Sorensen, Esin Tureci, and Margaret Martonosi. Foundations of Empirical Memory Consistency Testing. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020.

[KV21] Michalis Kokologiannakis and Viktor Vafeiadis. Genmc: A model checker for weak memory models. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification*, pages 427–440, Cham, 2021. Springer International Publishing.

[Lam79] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, September 1979.

[LCDS19] Daniel Liew, Cristian Cadar, Alastair F. Donaldson, and J. Ryan Stinnett. Just fuzz it: Solving floating-point constraints using coverage-guided fuzzing. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 521–532, New York, NY, USA, 2019. Association for Computing Machinery.

[LSG19] Daniel Lustig, Sameer Sahasrabuddhe, and Olivier Giroux. A Formal Analysis of the NVIDIA PTX Memory Consistency Model. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 257–270, New York, NY, USA, 2019. Association for Computing Machinery.

[LTPM15] Daniel Lustig, Caroline Trippel, Michael Pellauer, and Margaret Martonosi. ArMOR: Defending against Memory Consistency Model Mismatches in Heterogeneous Architectures. *SIGARCH Comput. Archit. News*, 43(3S):388–400, June 2015.

[LV16] Ori Lahav and Viktor Vafeiadis. Explaining relaxed memory models with program transformations. In John S. Fitzgerald, Constance L. Heitmeyer, Stefania Gnesi, and Anna Philippou, editors, *FM 2016: Formal Methods - 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings*, volume 9995 of *Lecture Notes in Computer Science*, pages 479–495, 2016.

[LWPG17] Daniel Lustig, Andrew Wright, Alexandros Papakonstantinou, and Olivier Giroux. Automated Synthesis of Comprehensive Memory Model Litmus Test Suites. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, page 661–675, New York, NY, USA, 2017. Association for Computing Machinery.

[MKN+18] Duncan J.M Moss, Srivatsan Krishnan, Eriko Nurvitadhi, Piotr Ratuszniak, Chris Johnson, Jaewoong Sim, Asit Mishra, Debbie Marr, Suchit Subhaschandra, and Philip H.W. Leong. A customizable matrix multiplication framework for the intel harpv2 xeon+fpga platform: A deep learning case study. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '18, page 107–116, New York, NY, USA, 2018. Association for Computing Machinery.

[MKP20] Yuan Meng, Sanmukh R. Kuppannagari, and Viktor K. Prasanna. Accelerating proximal policy optimization on cpu-fpga heterogeneous platforms. *28th IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, May 2020.

[MOG+14] Dominic P. Mulligan, Scott Owens, Kathryn E. Gray, Tom Ridge, and Peter Sewell. Lem: Reusable engineering of real-world semantics. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, page 175–188, New York, NY, USA, 2014. Association for Computing Machinery.

[Moo17] Andrew Moore. *Fpgas for Dummies*. John Wiley & Sons, Incorporated, 2017.

[MSS12] Luc Maranget, Susmit Sarkar, and Peter Sewell. A tutorial introduction to the arm and power relaxed memory models. *Draft available from http://www. cl. cam. ac. uk/˜ pes20/ppc-supplemental/test7. pdf*, October 2012.

[ND13] Brian Norris and Brian Demsky. Cdschecker: Checking concurrent data structures written with c/c++ atomics. *SIGPLAN Not.*, 48(10):131–150, oct 2013.

[OGN+22] Nicolai Oswald, Vasilis Gavrielatos, Vijay Nagarajan, Theo Olausson, Daniel J Sorin, and Reece Carr. Heterogen: Automatic synthesis of heterogeneous cache coherence protocols. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, USA, 2022. IEEE.

[OSC+11] Neal Oliver, Rahul R. Sharma, Stephen Chang, Bhushan Chitlur, Elkin Garcia, Joseph Grecco, Aaron Grier, Nelson Ijih, Yaping Liu, Pratik Marolia, Henry Mitchel, Suchit Subhaschandra, Arthur Sheiman, Tim Whisonant, and Prabhat Gupta. A reconfigurable computing system based on a cache-coherent fabric. In *2011 International Conference on Reconfigurable Computing and FPGAs*, pages 80–85, 2011.

[OSS09] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-tso. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, pages 391–407, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[PFD+17a] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. Simplifying arm concurrency: Multicopy-atomic axiomatic and operational models for armv8. *Proc. ACM Program. Lang.*, 2(POPL), dec 2017.

[PFD+17b] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. Simplifying arm concurrency: Multicopy-atomic axiomatic and operational models for armv8. *Proc. ACM Program. Lang.*, 2(POPL), December 2017.

[PPPK+19] Christopher Pulte, Jean Pichon-Pharabod, Jeehoon Kang, Sung-Hwan Lee, and Chung-Kil Hur. Promising-arm/risc-v: A simpler and faster operational concurrency model. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 1–15, New York, NY, USA, 2019. Association for Computing Machinery.

[PZS+21] Siddharth Priya, Xiang Zhou, Yusen Su, Yakir Vizel, Yuyan Bao, and Arie Gurfinkel. Verifying verified code. In Zhe Hou and Vijay Ganesh, editors, *Automated Technology for Verification and Analysis*, pages 187–202, Cham, 2021. Springer International Publishing.

[RGG+12] Petar Radojković, Sylvain Girbal, Arnaud Grasset, Eduardo Quiñones, Sami Yehia, and Francisco J. Cazorla. On the evaluation of the impact of shared resources in multithreaded cots processors in time-critical environments. *ACM Trans. Archit. Code Optim.*, 8(4):34:1–34:25, January 2012.

[Rin12] Martin C. Rinard. Unsynchronized techniques for approximate parallel computing. In *RACES@SPLASH*. ACM, 2012.

[Rup15] Karl Rupp. 40 Years of Microprocessor Trend Data. `https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data`, 2015.

[RWWC16] Nadesh Ramanathan, John Wickerson, Felix Winterstein, and George A. Constantinides. A case for work-stealing on fpgas with opencl atomics. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '16, page 48–53, New York, NY, USA, 2016. Association for Computing Machinery.

[SB18] Vladimír Still and Jiri Barnat. Model checking of C++ programs under the x86-tso memory model. In Jing Sun and Meng Sun, editors, *Formal Methods and Software Engineering - 20th International Conference on Formal Engineering Methods, ICFEM 2018, Gold Coast, QLD, Australia, November 12-16, 2018, Proceedings*, volume 11232 of *Lecture Notes in Computer Science*, pages 124–140. Springer, 2018.

[SBJS15] J. Stuecheli, B. Blaner, C.R. Johns, and M.S. Siegel. CAPI: A coherent accelerator processor interface. *IBM Journal of Research and Development*, 59(1):7:1–7:7, 2015.

[SD16] Tyler Sorensen and Alastair F. Donaldson. Exposing errors related to weak memory in gpu applications. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 100–113, New York, NY, USA, 2016. Association for Computing Machinery.

[SDB+16] Tyler Sorensen, Alastair F. Donaldson, Mark Batty, Ganesh Gopalakrishnan, and Zvonimir Rakamarić. Portable inter-workgroup barrier synchronisation for gpus. *SIGPLAN Not.*, 51(10):39–58, October 2016.

[Ser22] K Serebryany. *libFuzzer: A library for coverage-guided fuzz testing.* Google, 2022.

[Sew23] Peter Sewell. *POWER and ARM Litmus Tests*, February 2023.

[SFP+20] Ben Simner, Shaked Flur, Christopher Pulte, Alasdair Armstrong, Jean Pichon-Pharabod, Luc Maranget, and Peter Sewell. ARMv8-A system semantics: instruction fetch in relaxed architectures. In *ESOP 2020 - 29th European Symposium on Programming*, pages 1–30, Dublin, Ireland, March 2020. ACM.

[SGG13] Tyler Sorensen, Ganesh Gopalakrishnan, and Vinod Grover. Towards shared memory consistency models for GPUs. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, page 489–490, New York, NY, USA, 2013. Association for Computing Machinery.

[SMO+12] Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell, Luc Maranget, Jade Alglave, and Derek Williams. Synchronising c/c++ and power. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, page 311–322, New York, NY, USA, 2012. Association for Computing Machinery.

[Sny22] Wilson Snyder. *Verilator open-source SystemVerilog simulator and lint system.* Veripool, 2022.

[SSA+11] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding power multiprocessors. In *Proceedings of the 32nd ACM SIGPLAN*

Conference on Programming Language Design and Implementation, PLDI '11, page 175–186, New York, NY, USA, 2011. Association for Computing Machinery.

[TPO10] Stanley Tzeng, Anjul Patney, and John D. Owens. Task Management for Irregular-Parallel Workloads on the GPU. In Michael Doggett, Samuli Laine, and Warren Hunt, editors, *High Performance Graphics*. The Eurographics Association, 2010.

[TSC+21] Timothy Trippel, Kang G. Shin, Alex Chernyakhovsky, Garret Kelly, Dominic Rizzo, and Matthew Hicks. Fuzzing hardware like software. *CoRR*, abs/2102.02308, 2021.

[VPK18] A. Vaishnav, K. D. Pham, and D. Koch. A survey on fpga virtualization. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 131–1317, 2018.

[WBBD15] John Wickerson, Mark Batty, Bradford M. Beckmann, and Alastair F. Donaldson. Remote-scope promotion: Clarified, rectified, and verified. *SIGPLAN Not.*, 50(10):731–747, oct 2015.

[WBSC17] John Wickerson, Mark Batty, Tyler Sorensen, and George A. Constantinides. Automatically comparing memory consistency models. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, page 190–204, New York, NY, USA, 2017. Association for Computing Machinery.

[WC17] Felix Winterstein and George Constantinides. Pass a pointer: Exploring shared virtual memory abstractions in opencl tools for fpgas. In *2017 International Conference on Field Programmable Technology (ICFPT)*, pages 104–111, 2017.

[WHN19] Y. Wang, J. C. Hoe, and E. Nurvitadhi. Processor assisted worklist scheduling for fpga accelerated graph processing on a shared-memory platform. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 136–144, April 2019.

[Xil18] Xilinx. Accelerating dnns with xilinx alveo accelerator cards, October 2018.

[Xil23] Xilinx. *Power Efficiency*, January 2023.

[YFAE14] Hsin Jung Yang, Kermin Fleming, Michael Adler, and Joel Emer. Leap shared memories: Automating the construction of fpga coherent memories. In *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 117–124, 2014.

[ZCP16] Chi Zhang, Ren Chen, and Viktor Prasanna. High throughput large scale sorting on a CPU-FPGA heterogeneous platform. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2016.

[ZP17] S. Zhou and V. K. Prasanna. Accelerating graph analytics on cpu-fpga heterogeneous platform. In *2017 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 137–144, Oct 2017.

[ZTM+18] Hongce Zhang, Caroline Trippel, Yatin A. Manerkar, Aarti Gupta, Margaret Martonosi, and Sharad Malik. Ila-mcm: Integrating memory consistency models with instruction-level abstractions for heterogeneous system-on-chip verification. In *2018 Formal Methods in Computer Aided Design (FMCAD)*, pages 1–10, 2018.