The Semantics of Shared Memory in Intel CPU/FPGA Systems

DAN IORGA, Imperial College London, UK
ALASTAIR F. DONALDSON, Imperial College London, UK
TYLER SORENSEN, University of California, Santa Cruz, USA
JOHN WICKERSON, Imperial College London, UK

Heterogeneous CPU/FPGA devices, in which a CPU and an FPGA can execute together while sharing memory, are becoming popular in several computing sectors. In this paper, we study the shared-memory semantics of these devices, with a view to providing a firm foundation for reasoning about the programs that run on them. Our focus is on Intel platforms that combine an Intel FPGA with a multicore Xeon CPU. We describe the weak-memory behaviours that are allowed (and observable) on these devices when CPU threads and an FPGA thread access common memory locations in a fine-grained manner through multiple channels. Some of these behaviours are familiar from well-studied CPU and GPU concurrency; others are weaker still. We encode these behaviours in two formal memory models: one operational, one axiomatic. We develop executable implementations of both models, using the CBMC bounded model-checking tool for our operational model and the Alloy modelling language for our axiomatic model. Using these, we cross-check our models against each other via a translator that converts Alloy-generated executions into queries for the CBMC model. We also validate our models against actual hardware by translating 583 Alloy-generated executions into litmus tests that we run on CPU/FPGA devices; when doing this, we avoid the prohibitive cost of synthesising a hardware design per litmus test by creating our own 'litmus-test processor' in hardware. We expect that our models will be useful for low-level programmers, compiler writers, and designers of analysis tools. Indeed, as a demonstration of the utility of our work, we use our operational model to reason about a producer/consumer buffer implemented across the CPU and the FPGA. When the buffer uses insufficient synchronisation – a situation that our model is able to detect – we observe that its performance improves at the cost of occasional data corruption.

CCS Concepts: • **Theory of computation** → *Operational semantics*; *Axiomatic semantics*; • **Computer systems organization** → **Heterogeneous (hybrid) systems**.

Additional Key Words and Phrases: CPU/FPGA, Core Cache Interface (CCI-P), memory model

ACM Reference Format:
Dan Iorga, Alastair F. Donaldson, Tyler Sorensen, and John Wickerson. 2021. The Semantics of Shared Memory in Intel CPU/FPGA Systems. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 120 (October 2021), 27 pages. https://doi.org/10.1145/3485497

Authors' addresses: Dan Iorga, Department of Computing, Imperial College London, London, SW7 2AZ, UK, d.iorga17@imperial.ac.uk; Alastair F. Donaldson, Department of Computing, Imperial College London, London, SW7 2AZ, UK, alastair.donaldson@imperial.ac.uk; Tyler Sorensen, Department of Computer Science and Engineering, University of California, Santa Cruz, USA, tyler.sorensen@ucsc.edu; John Wickerson, Department of Electrical and Electronic Engineering, Imperial College London, London, SW7 2AZ, UK, j.wickerson@imperial.ac.uk.

2475-1421/2021/10-ART120
https://doi.org/10.1145/3485497

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 [Rupp 2015]. Now, with the future of Moore's law looking uncertain [Hennessy and Patterson 2019], 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.

A recent trend in heterogeneous systems is to combine a homogeneous multicore CPU with a field-programmable gate array (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 (avoiding the cost of manufacturing an application-specific integrated circuit), while the overall 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 [Abeydeera et al. 2016], neural networks [Guo et al. 2018], and image filtering [Dobai and Sekanina 2013].

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 [Intel 2019; Oliver et al. 2011], the IBM CAPI [Stuecheli et al. 2015] and the Xilinx Alveo [Xilinx 2018] – offer a *fine-grained* shared-memory interface between the CPU and FPGA. This 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., Farooqui et al. 2016; Ramanathan et al. 2016; Tzeng et al. 2010].

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. Fine-grained CPU/FPGA systems are new, and applications that exploit them are only just emerging, so we see this as an opportune time to examine their semantics rigorously and lay solid foundations for compiler writers and low-level application developers.

Modelling memory in CPU/FPGA systems. Our contribution is a detailed formal case study of the memory semantics of Intel's latest CPU/FPGA systems. These combine a multicore Xeon CPU with an Intel FPGA, and allow them to share main memory through Intel's Core Cache Interface (CCI-P) [Intel 2019]. We refer to this class of systems as X+F (Xeon+FPGA) throughout.[1] 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 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 [Clarke et al. 2004]. 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

[1] Other works have called these systems HARP [e.g. Moss et al. 2018], but we could find no official documentation from Intel using this terminology. Our naming scheme is consistent with recent work [Choi et al. 2019].

to the abstract machine. The axiomatic semantics, meanwhile, has been mechanised in the Alloy modelling language [Jackson 2012]. 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 a set of 583 *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 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.

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.

Contributions. In summary, the contributions of this paper are:

- an operational semantics for the memory system of Intel X+F CPU/FPGA devices, implemented in a form that is suitable for analysis using the CBMC model checker (Section 3);
- an axiomatic version of the semantics, formalised as an Alloy model in a manner that facilitates the automated generation of interesting executions (Section 4);

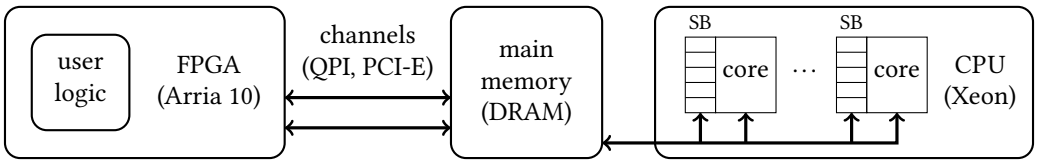


Fig. 1. Overview of the X+F memory system. The FPGA (left) communicates with main memory through several distinct channels, each mapped to a hardware bus, while the Xeon CPU (right) communicates through coherent caches. Each CPU core contains a store buffer (SB), which allows write/read reorderings.

- the design of a soft-core processor that allows memory model litmus tests to be executed on FPGA hardware in an efficient manner (Section 5);
- a large experimental campaign on an X+F implementation using a set of 583 disallowed and 180 allowed litmus tests generated from our axiomatic model, confirming that our model is empirically consistent with the hardware (Section 6); and
- a case study using our model to reason about different versions of an X+F implementation of a producer/consumer queue (Section 7).

Auxiliary material. Our CBMC and Alloy implementations and the Verilog code for our soft-core processor are available online [Iorga et al. 2021].

2 MEMORY CONSISTENCY IN THE X+F SYSTEM

We begin with an overview of the X+F memory system, which is depicted in Figure 1. 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.

As with any shared-memory system, the behaviour of loads and stores is governed by a *memory model*. The simplest (and strongest) memory model is *sequential consistency* [Lamport 1979], which states that every behaviour of a concurrent program must correspond to some interleaving of its instructions. Most modern CPU architectures use a *relaxed* memory model, which means that some memory operations are allowed to be reordered, with the aim of improving performance. In the case of the X+F system, the behaviour of loads and stores by the CPU and the FPGA is defined by the CCI-P standard from Intel [2019]. CCI-P 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 1. The most recent available generation of this system has three channels: a cache-coherent Quick Path Interconnect (QPI) channel and two Peripheral Component Interconnect Express (PCI-E) channels.

There are three main sources of relaxed memory behaviours. First, the Xeon CPU implements the x86-TSO memory model [Owens et al. 2009]. As such, each core has a store buffer (SB), which may allow writes to be re-ordered 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 that have different latencies.

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 2.1), and discussing

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

Fig. 2. 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).

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).

2.1 FPGA Coherency

The write/read litmus test of Figure 2a 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 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 CCI-P interface to memory, the behaviour *is* allowed. This is documented [Intel 2019, page 41], and observable in practice: we ran 1M iterations of Figure 2a under heavy memory traffic and observed non-coherent behaviour in around 0.1% of them.

To disallow this extremely weak behaviour, one of two CCI-P interface features must be used. First, FPGA-issued memory instructions can specify an explicit *channel* (cf. Figure 1). For instance, in Figure 2b, instructions 1 and 3 target channel 1, as indicated by "ch1:". Yet 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 2b 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.

The other mechanism for ensuring coherence is illustrated in Figure 2c, in which the write to and read from x do *not* target the same channel. A write response only 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 2c uses a fence for synchronisation. Once the fence *response* is observed, all writes must have reached main memory, so subsequent reads from different channels are guaranteed to observe up-to-date values.

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 3: an FPGA instruction stream is shown on the left and a CPU stream on the

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		

Fig. 3. 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.

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

Fig. 4. 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.

right. The FPGA stream has its distinctive synchronisation constructs: channel annotations and response-waiting. 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 1 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 2.1, 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.

Message Passing. Now we move on to some heterogeneous variants of the classic message-passing (MP) litmus test, shown in Figure 4. 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.

Figure 4a 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 channels that the client uses; we show an example of such a library (a circular buffer) in Section 7. To prevent the writes being reordered, a fence that synchronises across all channels is required [Intel 2019, page 41]. The CPU side of synchronisation is much simpler: TSO

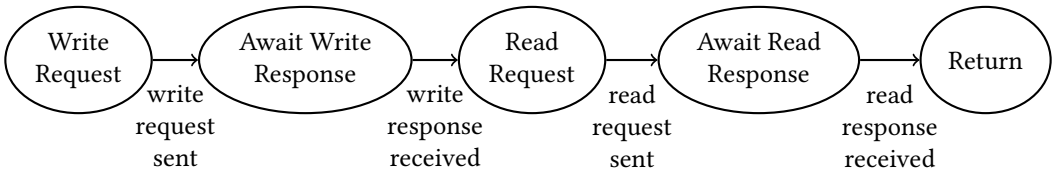


Fig. 5. A state machine corresponding to the litmus test in Figure 2b. The FPGA will only exit the Write/Read Request states once it has managed to send the corresponding request. It will only exit the Await Write/Read Response once the corresponding response has been received.

preserves read/read order so no extra instructions are required. (A weaker CPU architecture, such as PowerPC or Arm, would require different reasoning.)

Figure 4b shows another variant of the test, where this time the FPGA is the consumer. In this case, the FPGA simply needs to wait for the read response, which means that the read has been satisfied; no further synchronisation is required [Intel 2019, page 41]. The CPU side does not require additional synchronisation either, because write/write order is preserved in TSO.

2.3 Implementing Litmus Tests on the FPGA

Where 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 that is implemented on the FPGA. The circuit takes the form of a state machine. As an example, Figure 5 illustrates the state machine corresponding to the litmus test in Figure 2b. 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 in a similar fashion, after which the test is finished.

As another example, the litmus test from Figure 2a can be obtained by removing the "Await Write Response" state from Figure 5. Likewise, the litmus test of Figure 2c can be obtained by replacing the "Await Write Response" state with a "Fence Request" state followed by an "Await Fence Response" state.

3 AN OPERATIONAL FORMALISATION OF THE X+F MEMORY MODEL

As the examples in Section 2 show, the low-level shared-memory interface on the X+F system is complex and nuanced. We were only able to determine the outcomes of these tests through a combination of careful documentation-reading, discussion with the X+F engineers, and empirical testing. Channels and fences can interact in subtle ways and without sufficient synchronisation even single-stream shared-memory programs can yield counter-intuitive behaviours. Adding an instruction stream from a distinct computing element (e.g. a Xeon CPU) requires careful reasoning about the interaction of two memory models.

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). We then describe the set of *states* in which the memory system can reside (Section 3.2), and the set of *transitions* between states that the memory system can make in response to the FPGA's and CPU's actions (Section 3.3).

Having presented the model, we then justify our key modelling decisions with respect to available documentation about the X+F system (Section 3.4) and present an executable implementation of our model that enables reasoning about the behaviour of litmus tests using the CBMC tool (Section 3.5).

This executable model is also the basis for reasoning about the memory model idioms that underpin efficient implementations of synchronisation constructs in Section 7.

3.1 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).
- `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.

3.2 States

Figure 6 describes the states in which the system can reside.

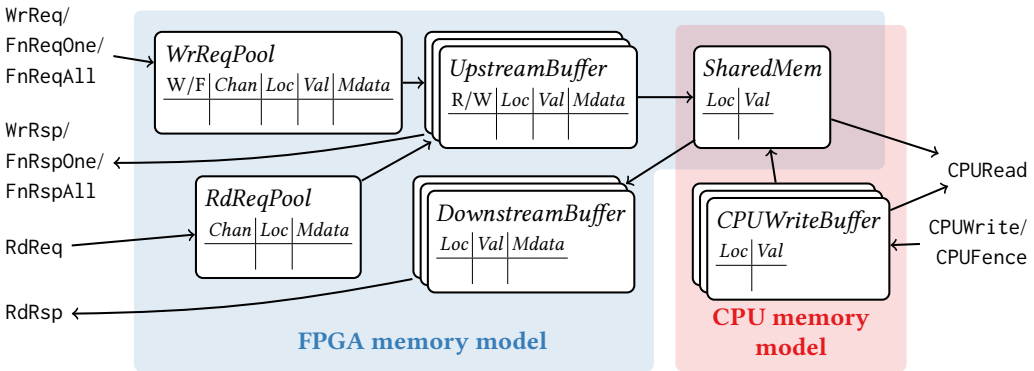
Working through the definitions in Figure 6a, we see the locations, values, metadata tags, and channels that we have encountered already. We use the notation $X_\perp$ for the set X extended with an additional $\perp$ element, which represents a blank.

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.

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 [Choi et al. 2019]. 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

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

(a) Formal definitions



(b) The system state, pictorially

Fig. 6. The memory system state that combines the FPGA view and the CPU view

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 6b.

3.3 Transitions

The state of the system can evolve by the CPU or FPGA performing one of the actions listed in Section 3.1 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 7) and those that only affect the CPU's view (Figure 8). These two sets of transitions can be combined (Figure 9) to describe the evolution of the entire system state.

The FPGA-related transitions defined in Figure 7 can be understood as follows. **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 8 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 9 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.

3.4 Justifications for modelling decisions

The model presented above was informed by studying the CCI-P manual [Intel 2019]. 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.

Write Request

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

Read Request

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

Fence Request One Channel

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

Fence Request All Channels

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

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

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

Fig. 7. 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

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

Fig. 8. Operational semantics, CPU side (following Owens et al. [2009])

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

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

Fig. 9. Operational semantics, FPGA and CPU combined

> "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." [Intel 2019, page 40]

> "Reads to the same [channel] may complete out of order; the last read response always returns the most recent data." [Intel 2019, 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.

init: x = 0

CPU:	FPGA:
1 CPUWrite(x, 1)	1 RdReq(ch1, x, m1)
2 CPUWrite(x, 2)	2 RdReq(ch2, x, m2)
	3 RdRsp(ch2, *v1*, m2)
	4 RdRsp(ch1, *v2*, m1)

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

Fig. 10. A litmus test that motivates downstream buffers

init: x = y = 0

CPU:	FPGA:
1 CPURead(y, *v1*)	1 WrReq(ch1, x, 1, m1)
2 CPURead(x, *v2*)	2 WrRsp(ch1, m1)
	3 FnReqOne(ch2, m2)
	4 FnRspOne(ch2, m2)
	5 WrReq(ch1, y, 1, m3)
	6 WrRsp(ch1, m3)

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

Fig. 11. A litmus test to check our understanding of **Flush Write Request to Upstream Buffer**

> "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." [Intel 2019, page 39]

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

> "All future writes on the same physical channel replace the data." [Intel 2019, page 38]

Downstream buffers. Figure 10 shows an allowed execution, adapted from the CCI-P manual [Intel 2019, 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

> "guarantees that all [...] writes preceding the fence are committed to memory before any writes following [the fence] are processed." [Intel 2019, 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 11, 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.

Table 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

3.5 CBMC implementation and litmus tests

We have implemented our model in CBMC [Clarke et al. 2004] 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 [Intel 2019].

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 7 and 8. The 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.

Simulation bounds. Naturally, the guarantees provided by this kind of simulation are limited by its bounded nature. Table 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 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).

4 AN AXIOMATIC FORMALISATION OF THE X+F MEMORY MODEL

We now present an axiomatic formalisation of the X+F memory model. Axiomatic formalisations are attractive because they can be easily compared against each other [Alglave et al. 2014]. The more immediate advantage for us is the possibility of generating a suite of conformance tests automatically from the axioms (Section 4.3).

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 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. 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` ∪ `WrRsp`,
- `R` for `CPURead` ∪ `RdRsp`,[2]
- `Req` for `RdReq` ∪ `WrReq` ∪ `FnReqAll` ∪ `FnReqOne`,
- `Rsp` for `RdRsp` ∪ `WrRsp` ∪ `FnRspAll` ∪ `FnRspOne`,
- `CPU` for the events from the CPU, i.e. `CPUWrite` ∪ `CPURead` ∪ `CPUFence`, and
- `FPGA` for the events from the FPGA, i.e. `Req` ∪ `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:

[2]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.

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

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

  No read has more than one incoming rf edge. We use rfe as a shorthand for $\mathtt{rf} \setminus \mathtt{sthd}$, which refers to a read from an external thread.
- po ('program order') is a strict, total order over all events within each thread. We further define poloc = po ∩ sloc and poch = po ∩ 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 Lustig et al. [2017], we define

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

  and we use fre as a shorthand for $\mathtt{fr} \setminus \mathtt{sthd}$.
- 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 readpair ∪ writepair ∪ fenceonepair ∪ 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.

4.2 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 [E \ 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' [Alglave et al. 2014] 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 following ten constraints all hold:

- **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
- **irreflexive**$(\texttt{fre} ; \texttt{rfe} ; \texttt{poch})$ — OBSERVE-SAME-CHANNEL
- **irreflexive**$(\texttt{po} ; \texttt{fenceallpair} ; \texttt{po} ; \texttt{writepair}^{-1})$ — FENCE-ALL-RESPONSE
- **irreflexive**$(\texttt{poch} ; \texttt{fenceonepair} ; \texttt{po} ; \texttt{writepair}^{-1})$ — FENCE-ONE-RESPONSE
- **irreflexive**$(\texttt{po} ; \texttt{writepair} ; \texttt{po} ; \texttt{fenceallpair}^{-1})$ — FENCE-ALL-BLOCK
- **irreflexive**$(\texttt{poch} ; \texttt{writepair} ; \texttt{po} ; \texttt{fenceonepair}^{-1})$ — FENCE-ONE-BLOCK

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 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. If a write request precedes a fence request, and the fence is either on all channels (FENCE-ALL-RESPONSE) or is

Table 2. 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

on the same channel as the write (FENCE-ONE-RESPONSE), then the write response must be received before the fence response. If a write request follows a fence request, and the fence is either on all channels (FENCE-ALL-BLOCK) or is on the same channel as the write (FENCE-ONE-BLOCK), then the write response must be received after the fence response.

4.3 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 Lustig et al. [2017]. 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 Lustig et al. [2017], we only generate 'interesting' disallowed executions – those that use the least synchronisation necessary to prevent a particular outcome. 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. We modify the technique of Lustig et al. [2017] 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 2 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. We know that executions obtained in this way are allowed because every event in an interesting disallowed execution is critical. (We could remove reads or writes as well as fences, but this would require the test's postcondition to be recalculated.)

4.4 Cross-checking the axiomatic and operational models

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. 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 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.

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 and used for testing against hardware in Section 6, except that we skipped 34 tests that involve more than two CPU threads: our CBMC

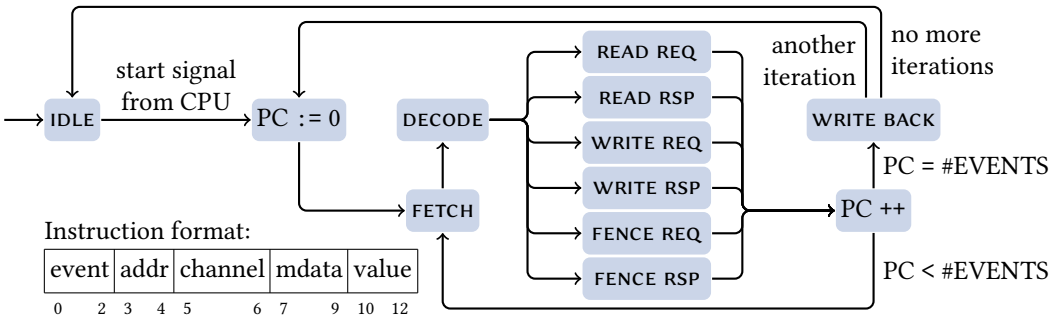


Fig. 12. A simplified representation of the soft-core state machine and the instructions it processes.

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.

5 A SOFT-CORE PROCESSOR FOR RUNNING TESTS ON THE X+F SYSTEM

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 current flow re-synthesises some elements such as the virtual-to-physical-address translator.[3] 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.

Figure 12 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 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

[3] 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.

Table 3. 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

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 [Alglave et al. 2011; Sorensen and Donaldson 2016] 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.

6 EXPERIMENTAL EVALUATION

The cross-validation efforts described in Section 4.4 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 gained access to an X+F system though the Intel Academic Compute Environment [Intel 2021]. 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 (Table 2). 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 3.

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. Kirkham et al. [2020] 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 3, 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).

7 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. 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 send them back to the CPU via a second queue. For example, Wang et al. [2019] employed such queues to implement graph algorithms on the X+F system.

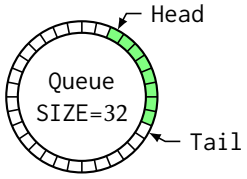


Fig. 13. A producer/consumer queue

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

Listing 1. Enqueue

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

Listing 2. Dequeue

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* sychronisation instructions are used: our memory model predicts that this should lead to a queue that loses messages, and indeed we observe this behaviour on the hardware in practice.

7.1 Implementation

A producer/consumer queue can be implemented using a circular array, as shown in Figure 13. The `SIZE` of the array limits the total number of elements that can be added.

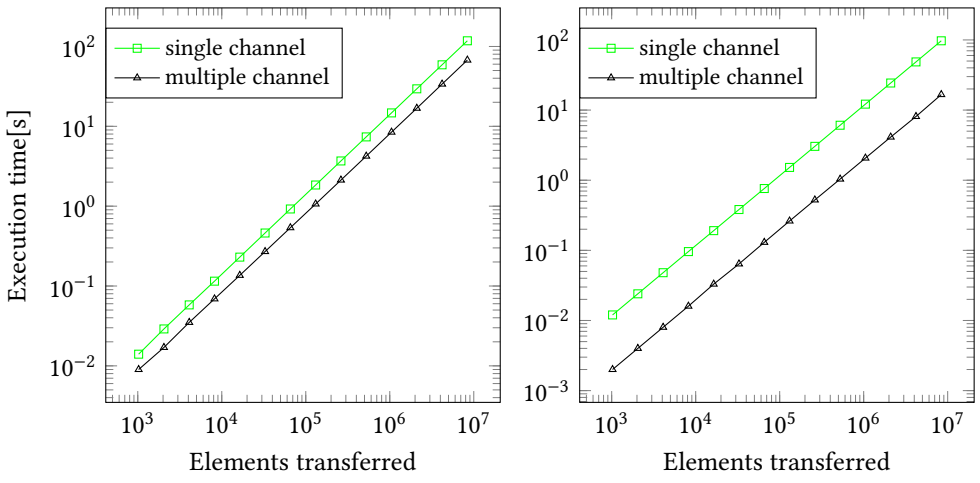
Listings 1 and 2 present the pseudocode of the enqueue and dequeue operations. The producer (resp. consumer) must ensure that the queue is not full (resp. empty) before adding (resp. dequeuing) an element. To achieve this, a simple lock-free implementation will continuously read `Head` (or `Tail`), spinning 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.

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 3.5) to confirm that these litmus tests are all guaranteed to behave correctly according to our model.

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.

7.2 Performance comparison

We use the same experimental platform as in Section 6. We repeated each experiment 10 times, but after observing that there was virtually no variance, we decided to report results just from the first



(a) FPGA enqueue (b) FPGA dequeue

Fig. 14. Execution time of the two variants of the queue running alongside heavy traffic

run. We simulate traffic on the FPGA by using the same technique described in Section 5, whereby random additional memory requests are sent every 10 clock cycles.

Figure 14 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. 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.

Contention on shared resources of multicore systems caused by memory stressing can significantly impact execution time of completely independent processes [Bechtel and Yun 2019; Iorga et al. 2020; Radojković et al. 2012]. 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 14a and 14b 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. 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.

7.3 Exploring incorrect behaviour

The implementations described in Section 7.1, and the associated performance results in Section 7.2, 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 Rinard [2012].

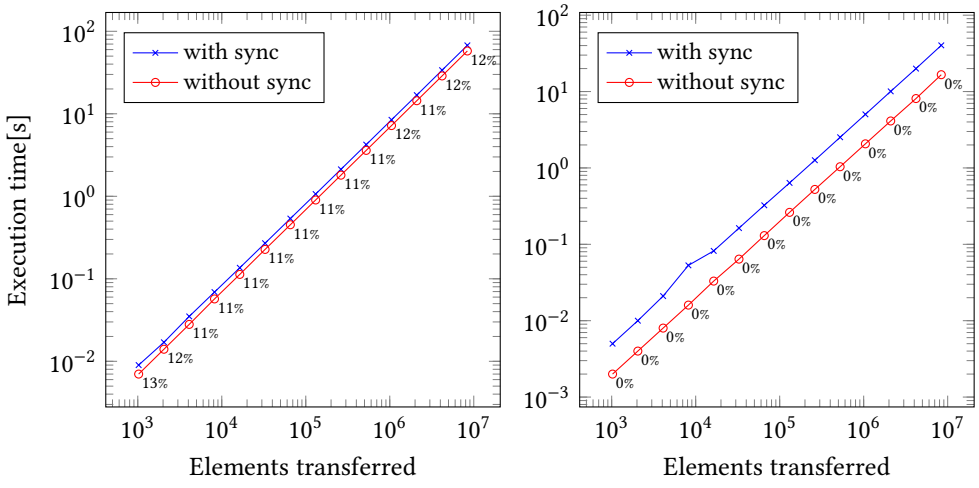


Fig. 15. Comparing the fully synchronised versions of the **multiple channel** queue with the improperly synchronised versions. The numbers next to the data points give the percentage of elements received incorrectly when synchronisation is omitted.

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 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 than the correctly synchronised versions, while the lossy dequeues are about 30–40% faster.

Figure 15a and Figure 15b show the same experiment with added stress. 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.

A lack of synchronisation causes elements to be lost, but this behaviour can only be observed when significant stress is applied. This echoes our experiments in Section 6 where the allowed behaviours were observed only when the stress was properly tuned. There is a clear performance benefit of the lossy versions, which might be tempting if the application can tolerate data loss.

8 FURTHER RELATED WORK

CPU/FPGA applications. Zhang et al. [2016] 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. Wang et al. [2019] and Zhou and Prasanna [2017] have shown that some graph algorithms are similarly well-suited to these platforms. Winterstein and Constantinides [2017] 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 [Guo et al. 2019; Guo et al. 2018; Meng et al. 2020].

FPGA synchronisation. Synchronisation primitives such as locks and barriers have been shown to be effective at enforcing orderings between FPGA threads [Yang et al. 2014]. Other works have shown how threads running on GPUs can be synchronised [Sorensen et al. 2016]. 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.

Memory modelling. CPU memory models such as x86 [Owens et al. 2009], POWER [Sarkar et al. 2011], Arm [Pulte et al. 2018], and RISC-V [Pulte et al. 2019] are now fairly well understood, as are some GPU memory models [Alglave et al. 2015; Lustig et al. 2019]. However, these models do not apply to systems where threads are on different devices.

Lustig et al. [2015] 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. Hower et al. [2014] 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 Hower et al. [2014] 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.

Zhang et al. [2018] show how to reason about systems-on-chip by building what is called an *instruction-level abstraction* (ILA) [Huang et al. 2018] 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.

9 CONCLUSION

Fundamental limits of hardware scaling mean that modern systems feature an increasing variety of heterogeneous platforms. CPU/FPGA systems represent an attractive option among these platforms, though they offer some unique programmability challenges. Our work lays the foundation for reasoning about memory consistency in such systems by examining one exemplar system in detail: the X+F system, which contains an Intel x86 CPU and an Altera FPGA. For this system, we provide formal semantics in two forms: operational and axiomatic. We have mechanised the operational semantics in C and the axiomatic semantics in the Alloy modelling language. We have validated the models via a cross-checking process and by running automatically generated tests on the hardware with the aid of a soft-core processor.

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.

This work shows how heterogeneous communication semantics can be rigorously defined, tested, and implemented in a realistic synchronization primitive. We hope this work will inspire future investigations as the computing landscape becomes even more heterogeneous.

Acknowledgements

We thank Michael Adler for many valuable discussions. We are grateful to Intel Labs for giving us access to an X+F test machine. Our work was financially supported by the EPSRC via the IRIS Programme Grant (EP/R006865/1) and the HiPEDS Doctoral Training Centre (EP/L016796/1).

REFERENCES

Maleen Abeydeera, Manupa Karunaratne, Geethan Karunaratne, Kalana De Silva, and Ajith Pasqual. 2016. 4K Real-Time HEVC Decoder on an FPGA. *IEEE Transactions on Circuits and Systems for Video Technology* 26, 1 (Jan 2016), 236–249. https://doi.org/10.1109/TCSVT.2015.2469113

Jade Alglave, Mark Batty, Alastair F. Donaldson, Ganesh Gopalakrishnan, Jeroen Ketema, Daniel Poetzl, Tyler Sorensen, and John Wickerson. 2015. GPU Concurrency: Weak Behaviours and Programming Assumptions. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (Istanbul, Turkey) *(ASPLOS '15)*. Association for Computing Machinery, New York, NY, USA, 577–591. https://doi.org/10.1145/2694344.2694391

Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. 2011. Litmus: Running Tests against Hardware. In *TACAS (Lecture Notes in Computer Science, Vol. 6605)*. Springer, 41–44.

Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Trans. Program. Lang. Syst.* 36, 2, Article 7 (July 2014), 74 pages. https://doi.org/10.1145/2627752

M. Bechtel and H. Yun. 2019. Denial-of-Service Attacks on Shared Cache in Multicore: Analysis and Prevention. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 357–367. https://doi.org/10.1109/RTAS.2019.00037

Young-Kyu Choi, Jason Cong, Zhenman Fang, Yuchen Hao, Glenn Reinman, and Peng Wei. 2019. In-Depth Analysis on Microarchitectures of Modern Heterogeneous CPU-FPGA Platforms. *ACM Trans. Reconfigurable Technol. Syst.* 12, 1, Article 4 (Feb. 2019), 20 pages. https://doi.org/10.1145/3294054

Edmund Clarke, Daniel Kroening, and Flavio Lerda. 2004. A Tool for Checking ANSI-C Programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004) (Lecture Notes in Computer Science, Vol. 2988)*, Kurt Jensen and Andreas Podelski (Eds.). Springer, 168–176.

Roland Dobai and Lukas Sekanina. 2013. Image filter evolution on the Xilinx Zynq Platform. In *2013 NASA/ESA Conference on Adaptive Hardware and Systems*. https://doi.org/10.1109/AHS.2013.6604241

Naila Farooqui, Rajkishore Barik, Brian T. Lewis, Tatiana Shpeisman, and Karsten Schwan. 2016. 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*. ACM, 30:1–30:2.

C. Guo, W. Luk, S. Q. S. Loh, A. Warren, and J. Levine. 2019. Customisable Control Policy Learning for Robotics. In *2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, Vol. 2160-052X. 91–98.

Kaiyuan Guo, Lingzhi Sui, Jiantao Qiu, Jincheng Yu, Junbin Wang, Song Yao, Song Han, Yu Wang, and Huazhong Yang. 2018. 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 (Jan 2018), 35–47. https://doi.org/10.1109/TCAD.2017.2705069

John L. Hennessy and David A. Patterson. 2019. A New Golden Age for Computer Architecture. *Commun. ACM* 62, 2 (2019), 48–60.

Derek R. Hower, Blake A. Hechtman, Bradford M. Beckmann, Benedict R. Gaster, Mark D. Hill, Steven K. Reinhardt, and David A. Wood. 2014. Heterogeneous-Race-Free Memory Models. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (Salt Lake City, Utah, USA) *(ASPLOS '14)*. Association for Computing Machinery, New York, NY, USA, 427–440. https://doi.org/10.1145/2541940.2541981

Bo-Yuan Huang, Hongce Zhang, Pramod Subramanyan, Yakir Vizel, Aarti Gupta, and Sharad Malik. 2018. Instruction-Level Abstraction (ILA): A Uniform Specification for System-on-Chip (SoC) Verification. *CoRR* abs/1801.01114 (2018). arXiv:1801.01114 http://arxiv.org/abs/1801.01114

Intel. 2019. *Intel Acceleration Stack for Intel Xeon CPU with FPGAs Core Cache Interface (CCI-P) Reference Manual.* https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/mnl-ias-ccip.pdf Version 2019.11.04.

Intel. 2021. Intel Academic Compute Environment. https://wiki.intel-research.net/

Dan Iorga, Alastair Donaldson, Tyler Sorensen, and John Wickerson. 2021. *The semantics of Shared Memory in Intel CPU/FPGA*. https://doi.org/10.5281/zenodo.5468873

Dan Iorga, Tyler Sorensen, John Wickerson, and Alastair F. Donaldson. 2020. Slow and Steady: Measuring and Tuning Multicore Interference. *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)* (2020), 200–212.

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

Jake Kirkham, Tyler Sorensen, Esin Tureci, and Margaret Martonosi. 2020. Foundations of Empirical Memory Consistency Testing. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 226 (Nov. 2020), 29 pages. https://doi.org/10.1145/3428294

L. Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.* 28, 9 (Sept. 1979), 690–691. https://doi.org/10.1109/TC.1979.1675439

Daniel Lustig, Sameer Sahasrabuddhe, and Olivier Giroux. 2019. A Formal Analysis of the NVIDIA PTX Memory Consistency Model. In *ASPLOS*. ACM, 257–270.

D. Lustig, C. Trippel, M. Pellauer, and M. Martonosi. 2015. ArMOR: Defending against memory consistency model mismatches in heterogeneous architectures. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. 388–400.

Daniel Lustig, Andrew Wright, Alexandros Papakonstantinou, and Olivier Giroux. 2017. Automated Synthesis of Comprehensive Memory Model Litmus Test Suites. In *ASPLOS*. ACM, 661–675.

Yuan Meng, Sanmukh R. Kuppannagari, and Viktor K. Prasanna. 2020. Accelerating Proximal Policy Optimization on CPU-FPGA Heterogeneous Platforms. *28th IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)* (May 2020). http://par.nsf.gov/biblio/10144121

Duncan J. M. Moss, Krishnan Srivatsan, Eriko Nurvitadhi, Piotr Ratuszniak, Chris Johnson, Jaewoong Sim, Asit K. Mishra, Debbie Marr, Suchit Subhaschandra, and Philip Heng Wai Leong. 2018. 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*. ACM, 107–116.

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. 2011. A Reconfigurable Computing System Based on a Cache-Coherent Fabric. In *2011 International Conference on Reconfigurable Computing and FPGAs, ReConFig*. IEEE Computer Society, 80–85.

Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. A Better x86 Memory Model: x86-TSO. In *Theorem Proving in Higher Order Logics*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 391–407.

Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. 2018. Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8. *Proc. ACM Program. Lang.* 2, POPL (2018), 19:1–19:29.

Christopher Pulte, Jean Pichon-Pharabod, Jeehoon Kang, Sung Hwan Lee, and Chung-Kil Hur. 2019. Promising-ARM/RISC-V: a simpler and faster operational concurrency model. In *PLDI*. ACM, 1–15.

Petar Radojković, Sylvain Girbal, Arnaud Grasset, Eduardo Quiñones, Sami Yehia, and Francisco J. Cazorla. 2012. On the Evaluation of the Impact of Shared Resources in Multithreaded COTS Processors in Time-critical Environments. *ACM Trans. Archit. Code Optim.* 8, 4, Article 34 (Jan. 2012), 25 pages. https://doi.org/10.1145/2086696.2086713

Nadesh Ramanathan, John Wickerson, Felix Winterstein, and George A. Constantinides. 2016. A Case for Work-stealing on FPGAs with OpenCL Atomics. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 48–53.

Martin C. Rinard. 2012. Unsynchronized Techniques for Approximate Parallel Computing. In *RACES@SPLASH*. ACM. https://people.csail.mit.edu/rinard/paper/races12.unsynchronized.pdf

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

Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. 2011. Understanding POWER multiprocessors. In *PLDI*. ACM, 175–186.

Tyler Sorensen and Alastair F. Donaldson. 2016. Exposing Errors Related to Weak Memory in GPU Applications. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 100–113. https://doi.org/10.1145/2908080.2908114

Tyler Sorensen, Alastair F. Donaldson, Mark Batty, Ganesh Gopalakrishnan, and Zvonimir Rakamarić. 2016. Portable Inter-workgroup Barrier Synchronisation for GPUs. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, New York, USA, 39–58.

J. Stuecheli, B. Blaner, C.R. Johns, and M.S. Siegel. 2015. CAPI: A Coherent Accelerator Processor Interface. *IBM Journal of Research and Development* 59, 1 (2015), 7:1–7:7. https://doi.org/10.1147/JRD.2014.2380198

Stanley Tzeng, Anjul Patney, and John D. Owens. 2010. Task management for irregular-parallel workloads on the GPU. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on High Performance Graphics*. Eurographics Association, 29–37.

Y. Wang, J. C. Hoe, and E. Nurvitadhi. 2019. 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)*. 136–144. https://doi.org/10.1109/FCCM.2019.00028

F. Winterstein and G. Constantinides. 2017. Pass a pointer: Exploring shared virtual memory abstractions in OpenCL tools for FPGAs. In *2017 International Conference on Field Programmable Technology (ICFPT)*. 104–111.

Xilinx. 2018. Accelerating DNNs with Xilinx Alveo Accelerator Cards. https://www.xilinx.com/support/documentation/white_papers/wp504-accel-dnns.pdf

H. J. Yang, K. Fleming, M. Adler, and J. Emer. 2014. LEAP Shared Memories: Automating the Construction of FPGA Coherent Memories. In *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*. 117–124.

Chi Zhang, Ren Chen, and Viktor Prasanna. 2016. High Throughput Large Scale Sorting on a CPU-FPGA Heterogeneous Platform. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. https://doi.org/10.1109/IPDPSW.2016.117

Hongce Zhang, Caroline Trippel, Yatin A. Manerkar, Aarti Gupta, Margaret Martonosi, and Sharad Malik. 2018. ILA-MCM: Integrating Memory Consistency Models with Instruction-Level Abstractions for Heterogeneous System-on-Chip Verification. In *FMCAD*. IEEE, 1–10.

S. Zhou and V. K. Prasanna. 2017. Accelerating Graph Analytics on CPU-FPGA Heterogeneous Platform. In *2017 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. 137–144. https://doi.org/10.1109/SBAC-PAD.2017.25