On accelerating concurrent short-running general-purpose tasks using FPGAs

Alexander Kroh

A thesis in fulfillment of the requirements for the degree of

Doctor of Philosophy

UNSW
AUSTRALIA

School of Computer Science and Engineering

Faculty of Engineering

The University of New South Wales

February 2020

THE UNIVERSITY OF NEW SOUTH WALES
Thesis/Dissertation Sheet

Surname or Family name: **Kroh**

First name: **Alexander** Other name/s: **Andre**

Abreviation for degree as given in the University calendar: **PhD**

School: **School of Computer Science and Engineering** Faculty: **Faculty of Engineering**

Title: On accelerating concurrent short-running general-purpose tasks using FPGAs

Abstract

FPGA technology is becoming a vital alternative to CPU-based processing as the performance of CPU technology plateaus. This is particularly prevalent in data centers and is impacting supercomputer design. In this thesis, I investigate the ability of FPGA technology to accelerate general-purpose systems, such as desktop computers and mobile devices. With high-performance, energy-efficient FPGA compute hardware, such systems could benefit from both a reduced execution time and an extended battery life.

Rather than executing a few long-running tasks, general-purpose systems typically host a large volume and variety of short-running tasks. For this reason, low-latency communication between the CPU and the FPGA is essential. My investigation begins by evaluating communication mechanisms between the tightly-coupled CPU and FPGA on the Xilinx Zynq platform as an example of a modern, commercial, heterogeneous system. This platform is an example of the growing trend of improving communication latency and throughput by co-locating the FPGA and the CPU in the same package.

My investigation assesses the potential of accelerating an operating system task scheduler. I demonstrate that scheduler priority queue acceleration can improve the performance of inter-process communication, but only if the communication method between CPU and FPGA is carefully chosen.

I then derive a formula for minimising a single task's completion time by partitioning the workload between the CPU and FPGA. That formula considers the latency and computational overhead required for signalling between the CPU and FPGA. The formula decides if, and by how much, the task should be executed in hardware.

I extend that formula to a dynamic execution environment and propose a framework that supports the acceleration of concurrent short-lived general-purpose workloads. I evaluate that framework in an emulated multitasking environment. Multiple applications that share a diverse set of accelerators are used to emulate general-purpose workloads.

I conclude that modern tightly-coupled devices can support short-lived general-purpose workloads and advance the state-of-the-art in how to effectively use such technology for this application. In a dynamic environment, response to changing demand must be quick or a diverse set of resident accelerators must be maintained to avoid reconfiguration delays overwhelming the throughput gains of FPGA acceleration.

Originality Statement

'I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, or substantial proportions of material which have been accepted for the award of any other degree or diploma at UNSW or any other educational institution, except where due acknowledgement is made in the thesis. Any contribution made to the research by others, with whom I have worked at UNSW or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project's design and conception or in style, presentation and linguistic expression is acknowledged.'

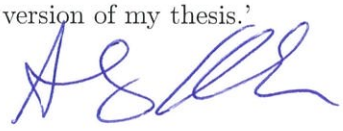
Alexander Kroh
February 6, 2020

Copyright Statement

Alexander Kroh
February 6, 2020

Authenticity Statement

'I certify that the Library deposit digital copy is a direct equivalent of the final officially approved version of my thesis.'



Alexander Kroh
February 6, 2020

UNSW SYDNEY | Australia's Global University

INCLUSION OF PUBLICATIONS STATEMENT

UNSW is supportive of candidates publishing their research results during their candidature as detailed in the UNSW Thesis Examination Procedure.

Publications can be used in their thesis in lieu of a Chapter if:

- The candidate contributed greater than 50% of the content in the publication and is the "primary author", ie. the candidate was responsible primarily for the planning, execution and preparation of the work for publication
- The candidate has approval to include the publication in their thesis in lieu of a Chapter from their supervisor and Postgraduate Coordinator.
- The publication is not subject to any obligations or contractual agreements with a third party that would constrain its inclusion in the thesis

Please indicate whether this thesis contains published material or not:

☐ This thesis contains no publications, either published or submitted for publication

☒ Some of the work described in this thesis has been published and it has been documented in the relevant Chapters with acknowledgement

☐ This thesis has publications (either published or submitted for publication) incorporated into it in lieu of a chapter and the details are presented below

CANDIDATE'S DECLARATION

I declare that:

- I have complied with the UNSW Thesis Examination Procedure
- where I have used a publication in lieu of a Chapter, the listed publication(s) below meet(s) the requirements to be included in the thesis.

Candidate's Name	Signature	Date (dd/mm/yy)
Alexander Kroh		6/02/2020

Abstract

FPGA technology is becoming a vital alternative to CPU-based processing as the performance of CPU technology plateaus. This is particularly prevalent in data centers and is impacting supercomputer design. In this thesis, I investigate the ability of FPGA technology to accelerate general-purpose systems, such as desktop computers and mobile devices. With high-performance, energy-efficient FPGA compute hardware, such systems could benefit from both a reduced execution time and an extended battery life.

Rather than executing a few long-running tasks, general-purpose systems typically host a large volume and variety of short-running tasks. For this reason, low-latency communication between the CPU and the FPGA is essential. My investigation begins by evaluating communication mechanisms between the tightly-coupled CPU and FPGA on the Xilinx Zynq platform as an example of a modern, commercial, heterogeneous system. This platform is an example of the growing trend of improving communication latency and throughput by co-locating the FPGA and the CPU in the same package.

My investigation assesses the potential of accelerating an operating system task scheduler. I demonstrate that scheduler priority queue acceleration can improve the performance of inter-process communication, but only if the communication method between CPU and FPGA is carefully chosen.

I then derive a formula for minimising a single task's completion time by partitioning the workload between the CPU and FPGA. That formula considers the latency and computational overhead required for signalling between the CPU and FPGA. The formula decides if, and by how much, the task should be executed in hardware.

I extend that formula to a dynamic execution environment and propose a framework that supports the acceleration of concurrent short-lived general-purpose workloads. I evaluate that framework in an emulated multitasking environment. Multiple applications that share a diverse set of accelerators are used to emulate general-purpose workloads.

I conclude that modern tightly-coupled devices can support short-lived general-purpose workloads and advance the state-of-the-art in how to effectively use such technology for this application. In a dynamic environment, response to changing demand must be quick or a diverse set of resident accelerators must be maintained to avoid reconfiguration delays overwhelming the throughput gains of FPGA acceleration.

Acknowledgements

I would first and foremost like to acknowledge my supervisor, Oliver Diessel. Oliver and I have shared numerous good times throughout my PhD. I am thankful that he kept me motivated and squelched dark thoughts of returning to industry before completion. Without his guidance and support, I would not have grown to become the researcher and author that I am today.

I would like to thank the numerous anonymous reviewers for their constructive feedback that has helped guide my research. Those comments and suggestions not only improved submitted work, but also shaped the direction of future work.

My numerous colleagues. In particular, Hamed Nosrati, Dimitris Agiakatsikas, Lingkan Gong and Amirreza Zarrabi, who I have shared many coffees, chats and memories with during the course of my candidature.

I would like to acknowledge the Australian government, who supported my research through an Australian Government Research Training Program Scholarship. Data61 and the University of New South for the top-up scholarships and travel funds that allowed me to make the most of my candidature.

Finally, I would like to thank my family. My patient wife, Melanie Kroh for her support and understanding as I embarked on my research adventure. My mother, Irmgard Kroh, for providing a cave and nutrition while writing my dissertation. And, of course, my daughter, Emily Kroh for her love and support.

Abbreviations

ABI application binary interface.

ACP accelerator coherency port.

API application programming interface.

ASIC application-specific integrated circuit.

ASID address space identifier.

ASIP application specific integrated processor.

AXI advanced extensible interface.

BRAM block random access memory (RAM).

CAPI Coherent Accelerator Processor Interface.

CPU central processing unit.

CSR configuration space registers.

DE device.

DMA direct memory access.

DPR dynamic partial reconfiguration.

DRAM dynamic RAM.

DSB data synchronisation barrier.

DSP digital signal processing.

FIFO first in, first out.

FIR finite impulse response.

FIU FPGA interface unit.

FPGA field programmable gate array.

FPT the International Conference on Field-Programmable Technology.

GP general-purpose.

GPU graphics processing unit.

H²RC the International Workshop for Heterogeneous High-performance Reconfigurable Computing.

HARP hardware accelerator research program.

HLS high-level synthesis.

HP high-performance.

HW hardware.

ID identifier.

ILA integrated logic analyser.

IO input/output.

IOMMU input/output (IO) memory management unit (MMU).

IPC inter-process communication.

IRQ interrupt request.

ISA Industry Standard Architecture.

JPEG Joint Photographic Experts Group.

LLC last level cache.

LUT lookup table.

MMIO memory-mapped IO.

MMU memory management unit.

MRE mean relative error.

OS operating system.

PaaS platform as a service.

PCI peripheral component interconnect.

PCIe peripheral component interconnect (PCI) express.

PMU performance monitoring unit.

PSL POWER Service Layer.

QPI QuickPath Interconnect.

RAM random access memory.

SaaS software as a service.

SCU snoop control unit.

SEV send event.

SM shared memory.

SMMU system MMU.

SO strongly-ordered.

SoC system on chip.

SRAM static RAM.

SW software.

TCB thread control block.

TLB translation lookaside buffer.

TRETS ACM Transactions on Reconfigurable Technology and Systems.

WCET worst case execution time.

WFE wait for event.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Traditional central processing unit (CPU) performance has become limited by its exponential growth in power requirements with performance. Although we have turned to graphics processing units (GPUs) to improve performance-per-watt, such architectures are tuned for applications that perform parallel floating point operations rather than general-purpose computations. On the other hand, reconfigurable custom compute hardware (HW) has shown promise in a wide range of computing applications and we have seen a rapid adoption of reconfigurable field programmable gate array (FPGA) devices in the cloud [WUZY19,SFJ+19,CSZ+14], data centers [PCC+14,WPAH16], and high-performance supercomputing [VB13, CA07, BBB+07].

The application domain of FPGA-based accelerators has traditionally been limited to long-running, compute-intensive workloads. Examples include large-scale database queries [CP16, BS13, WPC+16, YOY17], machine learning [WXH+16, WGY+17, KAA+17], data mining [ŠRS12, ZZJB13, BP06], genome sequencing [CCC+16, PWP+03], pattern matching [SIOA17, DMBS12, ISA16, CH07], and filtering [FAL+16]. These tasks involve transferring large amounts of data between main memory and FPGA-local memory or building an application-specific data streaming machine. Therefore, such tasks rely on high memory throughput rather than low memory latency.

In contrast, my research is focused on accelerating many concurrent, short-running tasks that are commonly found in servers and consumer mobile/desktop computers. For such tasks, the overhead of transferring data between the CPU and the accelerator often outweighs the savings in execution time. Additionally, the long reconfiguration delay of FPGA HW may lead to a task executing to completion on the CPU before an accelerator is available to assist with processing.

Tightly-coupled, high-performance CPU-FPGA systems have emerged in which a general-purpose processor and programmable logic are located on the same device [IBM14, Xil14, Int19a]. Not only does this close proximity reduce the communication latency between them, it also allows HW and software (SW) to access shared memory (SM) via the on-chip last level cache (LLC). While such low-latency data access has shown promise in traditional applications [Bea16, WC17, WMW+16], it may also extend the range of applications that are suitable for HW acceleration from long-running tasks to short-running tasks.

By targeting concurrent general-purpose tasks for acceleration, we increase the likelihood that accelerators can be shared between tasks. This amortises the cost of reconfiguration across the tasks that the accelerator benefits. When an accelerator is required and not yet configured, the requesting short-running task may complete before it is ready for use. However, once it is available, that accelerator can be used by the requesting task and any similar task that executes on the system. While this benefits multitasked systems that are operated by a single user, we also expect such a model to benefit servers, software as a service (SaaS) and platform as a service (PaaS) systems. In this case, many users share the compute platform. SaaS and PaaS users can be migrated between servers to further increase the potential for accelerator sharing.

With this research we expect that a larger number of end-users could benefit from faster and cheaper computational machines that exploit HW acceleration. My vision is that all general purpose computers will be provisioned with reconfigurable HW and that this reconfigurable HW will improve the performance of both server workloads as well as everyday activities such as email filtering, document processing, or even photo classification and tagging.

1.1 Thesis objectives

Major CPU vendors, such as IBM, Intel and ARM, have developed platforms that can integrate programmable logic, as embodied in FPGAs, more closely with their CPUs. IBM has released an accelerator card interconnect that provides a view of the memory hierarchy that is consistent with that of the CPU [SBJS15]. Intel provides reconfigurable devices that are compatible with a typical CPU socket [Int19b]. Finally, Intel and Xilinx have both released devices that co-locate an ARM CPU and an FPGA on the same package [Xil14] [Int19a].

I hypothesise that the emergence of these new tightly-coupled CPU-FPGA architectures allow short-running tasks to benefit from HW acceleration.

The aims of my thesis are as follows:

A1 To demonstrate that tightly-coupled CPU-FPGA systems extend the spectrum of applications that can benefit from HW acceleration.

A2 To provide models that SW engineers can use to select the most appropriate communication mechanisms for short-running tasks.

A3 To show that overall performance of general-purpose systems can be improved when many concurrent, short-running tasks share FPGA-based accelerators.

1.2 Thesis contributions

My thesis makes the following key contributions:

C1 I show that tightly-coupled CPU-FPGA systems can improve the performance of inter-process communication by offloading the management of SW task scheduling queues from the operating system (OS) to HW.

This contribution is motivated by A1.

C2 Via C1, I evaluate very short transfers (32 bits) between CPU and FPGA using the communication methods that are provided by the tightly-coupled Zynq-7020 system on chip (SoC).

This contribution is motivated by A2.

C3 I extend C2 to provide a more comprehensive evaluation of the CPU overhead and latency of communication across the available communication methods on the Zynq-7020 SoC. This contribution allows SW engineers to select the optimum communication method for their application.

This contribution is motivated by A2.

C4 I propose and evaluate a methodology for partitioning a single accumulator workload between SW and a single FPGA-based accelerator to minimise overall execution time.

This contribution is motivated by A2.

C5 I extend C4 to consider many concurrent accelerators that are shared by many concurrent tasks.

This contribution is motivated by A2.

C6 I propose and evaluate a framework that supports a dynamic range of FPGA-based accelerators and incorporates the low-latency communication methods identified in C3. A set of concurrent, short-running workloads for a set of functions are partitioned using C5 to minimise execution time.

This contribution is motivated by A3.

C7 Using C6, I identify CPU time-sharing as a hazard for cooperative processing C4 in multitasking environments. I propose and evaluate a method for compensating for that hazard.

This contribution is motivated by A3.

1.3 Thesis structure

While the primary contribution of my thesis is an exploration of the benefits of tightly-coupled accelerators for short-running tasks, my thesis spans a number of disjoint domains. An independent literature survey is therefore provided in each of the main contribution chapters (Chapters 3, 4 and 5).

Chapter 2 provides background information to the reader. It surveys commercially available tightly-coupled systems and identifies their differentiating features. The focus of that survey is the Zedboard that was used throughput this thesis as an apparatus for experimentation. Chapter 2 also presents a HW model that identifies the expected HW features of my work and how selected commercially available platforms implement it. Chapter 2 concludes by defining and modelling general-purpose tasks.

Chapter 3 covers my research in accelerating an OS SW task scheduler (C1) and, in the process, evaluates very short transfers on the tightly-coupled Zynq-7020 SoC (C2). A priority queue was implemented in HW and connected to the CPU using a range of the available communication ports. The chosen method of connection determined whether HW provided a performance improvement or degradation. As a domain outside the scope of this thesis, Chapter 3 includes a brief literature survey of OS task scheduler acceleration.

Chapter 4 covers my research in partitioning a workload between HW and SW to maximise system utilisation and minimise completion time. This work extends the exploration of communication methods in Chapter 3 to consider a range of transfer sizes (C3) and provides a methodology for partitioning a workload in a single task environment (C4).

In Chapter 5 I present a framework for accelerating many concurrent short-lived tasks (C6). Hazards associated with CPU time-sharing and cooperative processing are described along with proposed countermeasures (C7). Finally, I extend the partitioning algorithm from Chapter 4 to consider a dynamic set of resident accelerators in a multitasking environment (C5).

Chapter 2

Background

General-purpose systems are systems that are capable of performing a wide variety of tasks. In this chapter, I provide an overview of the concepts and HW of such systems. I begin by outlining reconfigurable computing and its application to general-purpose systems. I then describe general-purpose processors and their key features. I continue by comparing the techniques used by these different technologies to support a wide range of concurrent tasks. After a brief summary of the memory hierarchy of general-purpose systems, I identify methods of coupling reconfigurable HW to different locations in that hierarchy. The chapter next identifies and discusses commercially available tightly-coupled general-purpose systems. I conclude this chapter with a model of the assumed HW that is used throughout this thesis.

2.1 Reconfigurable computing

Reconfigurable HW provides a flexible, programmable HW system. Operations are programmed as custom logic circuits that can be connected in series or in parallel using a configurable routing network. The flexibility of such customised HW provides an improved execution time and energy efficiency when compared to traditional general-

purpose CPUs [FBCS12].

The most common reconfigurable device is the FPGA. The primary logic unit of an FPGA is a lookup table (LUT). Through the use of design tools, engineers configure each LUT to provide an output that is determined by the set of inputs applied to the LUT.

LUTs can be configured for a wide range of arithmetic and logic operations. Two concrete examples are a multiplexer and an adder. Given a LUT with 3 inputs[1], the design tools can implement a multiplexer by using one input to select which of the remaining 2 inputs should be observed at the output. Alternatively, the tools may configure such a LUT to provide single-bit addition. In that case, the 2 inputs provide 1-bit values that should be added while the remaining input provides a carry in. The LUT is then configured to report the sum of those signals while a second LUT can be configured to report the carry out. By cascading these addition LUTs, the bit-width of the input numbers can be extended and optimised to meet the application's needs. Additionally, the remaining LUTs on the FPGA can be configured such that many of these addition operations are performed in parallel or sequentially. This flexibility allows a target application to be optimised for energy and execution time.

When compared to fixed HW, the flexibility provided by an FPGA comes at the cost of performance. The additional circuits that allow a LUT and their interconnection to be configured adds delay to the designed circuit. Considering this, FPGAs provide a set of fixed optimised HW resources, such as block memories for local storage, multipliers, digital signal processing (DSP) blocks, general-purpose processors [Xil14], and machine learning functions [Xil20a].

Despite the performance penalty, FPGAs have become a popular alternative to fixed application-specific integrated circuit (ASIC) HW. The exponential development cost of ASICs with respect to design complexity make FPGA HW more favourable for low volume production and products still under development. When an FPGA is used, the

[1]Modern devices comprise 6-input LUTs, which aim to reduce user circuit area and improve speed.

functionality of HW can be changed quickly and remotely without changing the physical HW platform. Putnam et. al. showed that the large-scale use of FPGAs in data centers can improve the throughput of web search engine ranking [PCC+14]. Such a large-scale deployment using ASICs would require that each ASIC be physically replaced in each machine when a better ranking algorithm is discovered.

Many cloud providers now provide FPGA HW in their PaaS product range [WUZY19] [SFJ+19]. The end user provides a HW design and leases time for its deployment on a remote FPGA for data processing.

2.1.1 Automating design

A challenge for FPGA application design lies in the nature of programming parallel circuits [BRS13]. One must consider parallel processing and how a design will be mapped to the available resources on the FPGA. This presents a steep learning curve for new programmers and has led to a shortage in skilled FPGA programmers.

A technique known as high-level synthesis (HLS) is being used to decrease the gap between the performance of reconfigurable HW and its programmability [RJ16]. These advancements in tools allow FPGA designs to be constructed from more familiar higher-level languages, such as C. Using HLS, a wider set of engineers are able to exploit the performance improvement and energy efficiency of FPGA-based designs.

Others extrapolate opportunities for reconfigurable HW from compiled applications. Vahid et. al. developed a run-time tool for profiling an application to identify frequently executed code segments [VSL08]. They synthesise these segments to HW while the application is executing. Once the HW design has been generated and configured into the FPGA, the application will begin to use the accelerator rather than executing in SW. This technique allows existing applications to use FPGA-based accelerators when the original applications sources are unavailable.

2.1.2 Dynamic partial reconfiguration

The set of useful accelerators varies across tasks. One task may benefit from an accelerator that performs decryption while another benefits from decompression acceleration. Given a wide range of tasks for general-purpose systems [GRE+01, CPM97], it is not feasible to maintain all accelerator configurations in programmable logic at the same time.

A technique known as dynamic partial reconfiguration (DPR) extends the effective area of the FPGA by sharing it in both space and time [VF18] [GSB+00] [WH95]. When using DPR, the programmable logic is partitioned into regions that can each host an accelerator. The target region is first disconnected from the routing network. The logic configuration of the target area is then replaced by a new configuration. Finally, the new configuration is connected back to the routing network and is ready to be used by an application.

Ideally, reconfigurable regions are homogeneous. When regions are of uniform size and contain identical logic and routing resources, each accelerator can be configured into any available reconfigurable region [SWP04] [Bre96]. An alternative approach is to synthesise all accelerators for all reconfigurable regions [CKPP15]. The accelerator implementation that corresponds to the target region is selected when the accelerator is configured.

As the area requirements of accelerators vary, some logic in fixed-sized regions will inherently be unused. Burns et. al. propose a method to support reconfigurable regions of flexible size by rerouting resources to fit the available area at configuration time [BDH+97]. While this leads to a more flexible placement, it also leads to fragmentation of the FPGA area. Although there may be sufficient area to configure an accelerator, that area may be sparsely distributed throughout the FPGA. Diessel et. al. propose a method of compacting the configured accelerators to increase the usable area for new configurations [DE97]. While those methods improve the area utilisation of the FPGA, they incur overheads in reconfiguration latency and processing time.

When the sequence of tasks is known in advance, the scheduling of accelerators in programmable logic can be computed offline [GSB+00]. The FPGA area is then shared in

time by swapping an accelerator that is no longer needed for the accelerator that will be needed next. However, the arrival times of tasks in general-purpose systems are sporadic. It is difficult to predict which task the user will perform next. Therefore it is difficult to predict which accelerators should be configured in programmable logic at any given time [WP02].

2.2 General-purpose processors

General-purpose CPUs consists of one or more processor cores that generally execute one instruction, from a fixed set, for each system clock cycle. An instruction stream is loaded from memory and executed sequentially by each core. Collectively these instructions can perform a wide range of tasks.

CPU instructions can be placed into four categories: 1. data access instructions move data between system memory and the CPU core's register file; 2. arithmetic instructions perform arithmetic and logic operations on data that is contained in the register file; 3. control flow instructions change, sometimes conditionally, the stream of instructions that the core executes; 4. system instructions control CPU state, such as operating mode and coprocessor behaviour. Each core is capable of executing an independent stream of instructions.

Modern superscalar processors are capable of executing more than one instruction per cycle. The CPU identifies data dependencies between instructions to determine if a strict order of execution is required. When instructions can be executed independently, and if processing resources are available, the instructions can be executed in parallel or out of program order [PH90].

The CPU uses IO peripherals to collect instructions and data, and to present processing results to the user. Peripherals are controlled via a set of registers known as configuration space registers (CSR). CPU cores access the CSR using dedicated system instructions (programmed IO) or by memory-mapped IO (MMIO). MMIO is the most common method of communication and is used throughout this thesis. MMIO enables the CPU to access

the CSR of connected peripherals executing read and write instructions to an address range that is reserved for the target peripheral. For example, SW may enable a peripheral by writing the value 0x00000001 to a control register of the peripheral that is mapped to address 0x40000000. SW may then transfer data to the peripheral by writing a sequence of values to the address 0x40000004.

A general-purpose system can be constructed with a variety of CPUs and memories of various size. The address at which memory is accessed by the CPU also depends on where that memory is connected to the system. With a wide range of computer HW configurations available, an OS is used to provide an abstract model of HW to hosted applications. Most important in that model is processor time-sharing and virtual memory. These abstractions will be discussed in the sections that follow.

2.3 Sharing compute resources

General-purpose systems typically host many concurrent tasks that compete for resources. When fewer processing resources are available than tasks demand, processing resources must be shared. In this section, I identify method for sharing processing time in the context of both CPU and FPGA systems.

2.3.1 CPU time-sharing

When more SW tasks execute than there are CPU cores, each task is given a fixed time in which to execute before they are swapped for another task. The OS configures a timer to interrupt the CPU when the time slice of a task expires. Since a task can be interrupted at any time, the CPU must ensure that enough task *context* is saved so that the task can resume execution unaware of the interruption when the context is later restored. In this case, the context is CPU register file content and the position of the task in its instruction stream.

Such a time-sharing environment is referred to as being multiprogrammed when the system provides one CPU core and multitasked when tasks are scheduled across many cores. In this thesis, we refer to multitasking as an environment in which tasks are scheduled on one or more CPU cores.

2.3.2 Accelerator time-sharing

Time-sharing of the FPGA area was discussed in Section 2.1.2. In this section I discuss time-sharing of a configured accelerator. Accelerator time-sharing follows the same principles of CPU time-sharing: the context of the accelerator is periodically swapped for the context of another. While the context of a SW task is limited to a small number of CPU registers, the context of a HW task includes all memories that are local to the accelerator. Therefore, a large amount of data may need to be saved and restored, particularly when the entire dataset of the task is stored locally for low-latency access.

One approach to context extraction is to use the configuration port of the FPGA to read back the state of FPGA resources [SLM00] [KP05] [HTK15] [JHE+13]. While the configuration includes the content of local memories used by the accelerator, it also includes the content of unused memories. The extraction of such a large quantity of data requires additional time for which the accelerator must be idle. Additionally, it does not consider shared state between the accelerator and external resources. Such state may reflect the status of outstanding transactions to or from off-chip memory.

While mechanisms for saving and restoring CPU context are inherently built into the instruction set of the CPU, such features should be designed to efficiently swap accelerator context [KHT07]. In this case, a port must be implemented by the accelerator that allows application-specific context to be exchanged. The logic and routing required for extraction presents an area overhead in programmable logic, but reduces context switching time.

The context size can be further reduced by notifying an accelerator of an impending context switch [MNM+04] [XPN16]. The accelerator can then continue execution until it arrives

at a suitable preemption point. While this assumes that accelerators will respond to the context switch request, it allows the accelerator to complete outstanding transactions to external resources before its context is swapped.

The need to store context can be eliminated completely when tasks are partitioned into short-running subtasks [VPKG18] [LP09a]. The tasks can then execute to completion before their hypothetical time slice expires. This avoids both the overhead and design complexity of context swapping.

2.4 Virtual memory

The virtual memory system provides a uniform view of memory layout across systems and applications. It achieves that by translating each memory access made by a hosted application to a unique location in RAM. This allows many concurrent applications to access private physical memory using common virtual addresses.

A trivial implementation of a virtual memory system is to divide the application address space into segments that can be relocated to disjoint segments of physical memory [Den65]. Base and limit segment descriptor registers are used to configure a virtual-physical translation for each segment. When a virtual address is accessed, HW performs a lookup of these segments to perform the correct translation. The transaction is then directed to physical memory using the translated address.

When multiple tasks share time on a CPU core, the set of segment descriptors must be exchanged as part of the task's context. This ensures that the correct translations are performed for the running application and prevents one application from accessing the address space of another.

The segment descriptor approach is limited by the number of segment descriptor registers that the system provides. An alternate approach that overcomes that limitation is to store the content of such registers in system memory [BCD69]. When switching between

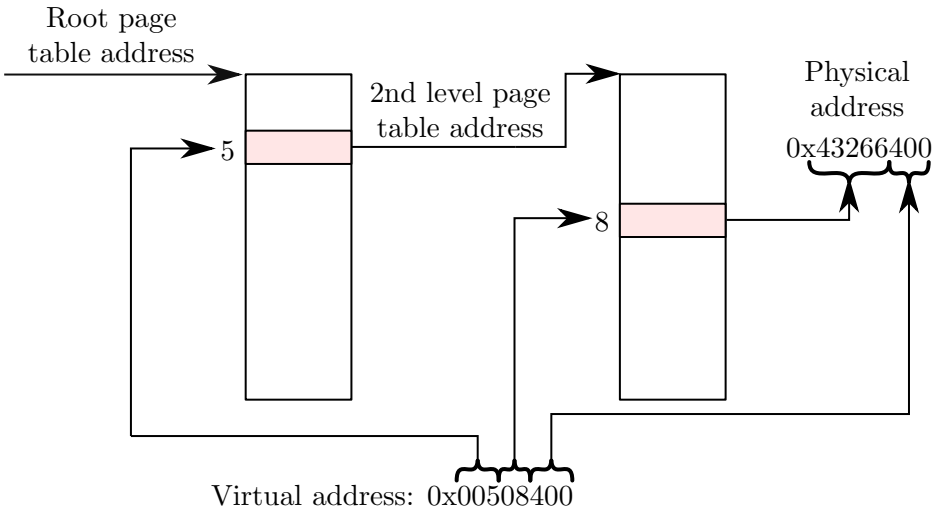


Figure 2.1: Virtual memory translation using a page table.

address spaces, only the memory address of those registers needs to be saved and restored. The number of segment descriptors is now limited only by the size of system memory. Such data structures are referred to as page tables as they represent a table of virtual to physical memory translations. Memory is partitioned into fixed sized pages that are typically 4 KB in size. Such page tables are found in modern virtual memory systems [ARM05] [Int16].

To conserve memory, page tables provide a hierarchical lookup (Figure 2.1). The root page table is indexed by the most significant bits of the address and, if a translation in that virtual address range exists, the table entry provides the memory address of the next table in the hierarchy. The next significant bits of the virtual address are then examined to find an index within that table. At the last level of the hierarchical lookup, the translation provides the physical address of the translation as well as the memory access permissions and attributes.

The overhead of repeated page table traversals are avoided by using a translation lookaside buffer (TLB). The TLB is a cache that stores recent translations for the running application. Subsequent translations can then be served from the TLB rather than requiring a page table lookup. When performing a context switch to another application, we must ensure that the TLB serves only the cached translations of the new application. This is done by tagging each TLB entry with an address space identifier (ASID). We then configure the

TLB to only serve translations for the configured identifier. That allows translations for one application to remain resident while another application executes.

On modern systems, the TLB is filled by a dedicated coprocessor that traverses the page table structures. The page table walker may also anticipate future translations and speculatively walk the page table structures. This reduces data access latency and the likelihood of a CPU stall when the translation is needed.

The TLB and page table walker are collectively referred to as the MMU. Each core is provided with a private MMU for translations.

As well as a translation service, the MMU allows memory to be paged to disk when system memory becomes exhausted. In that case the OS may free memory by temporarily moving it to disk. It then removes corresponding translations from page tables and TLBs. The next time that data is accessed, the MMU will interrupt the OS because a translation is unavailable. The OS responds by restoring data from disk, perhaps to a different location in system memory, and updates the corresponding page table entries. The application can then continue unaware that its memory was ever moved to disk.

Memory paging also allows applications to begin sooner and consumer less memory. By prioritising the loading of application instructions and data with the order in which they are accessed, the application can begin execution as soon as the first instruction has been loaded from disk. Instructions and data that are never executed or accessed may never occupy system memory.

2.4.1 Hardware virtual memory

MMUs have also been deployed for HW peripherals. On Intel architectures, such an MMU is referred to as an IOMMU. On ARM, it is referred to as a system MMU (SMMU). We adopt Intel's naming convention in this thesis.

The IOMMU provides memory translation and memory protection for peripherals that

access system memory. While that allows applications and peripherals to share a common address space for data, IOMMUs are more commonly used when hosting other OSs as an application on the system [PH90]. In that case, the IOMMU ensures that the hosted OS and selected peripherals have a consistent view of memory addressing.

Winterstein et. al. propose that each accelerator should be provisioned with its own IOMMU [WC17]. This allows the application to "pass a pointer" directly to accelerators rather than manually translating addresses on the accelerator's behalf. The system improves performance by allowing accelerators to share memory through a common addressing scheme and by avoiding the translation of addresses that accelerators never access. For example, an algorithm that operates on a tree may only access 5% to 28% of tree nodes [WC17].

With the common goal of simplifying the programming model, Mirian et. al. explored integration methods for MMUs and FPGA-based accelerators [MC15]. The authors investigate systems where accelerators are provided with dedicated IOMMUs and where accelerators share the MMU of CPU core. The CPU in this study was implemented in the FPGA. Although the authors conclude that an independent IOMMU is best, it is difficult to determine if the observed benefit is not simply due to a collective increase in TLB size.

Vogel et. al. provide a detailed study of the design trade-offs for an IOMMU implemented in programmable logic [VMB19]. The flexibility provided by the programmable logic enables their design to outperform a hard macro IOMMU.

As well as a unified address space, the IOMMU maintains the memory abstraction models provided by the OS. This allows data accessed by HW to be loaded lazily and paged to disk when memory is low. Without an IOMMU, the OS must *pin* memory that is used by peripherals to ensure that it is always available. For short running tasks, the overhead of calling the OS to pin such memory can be significant.

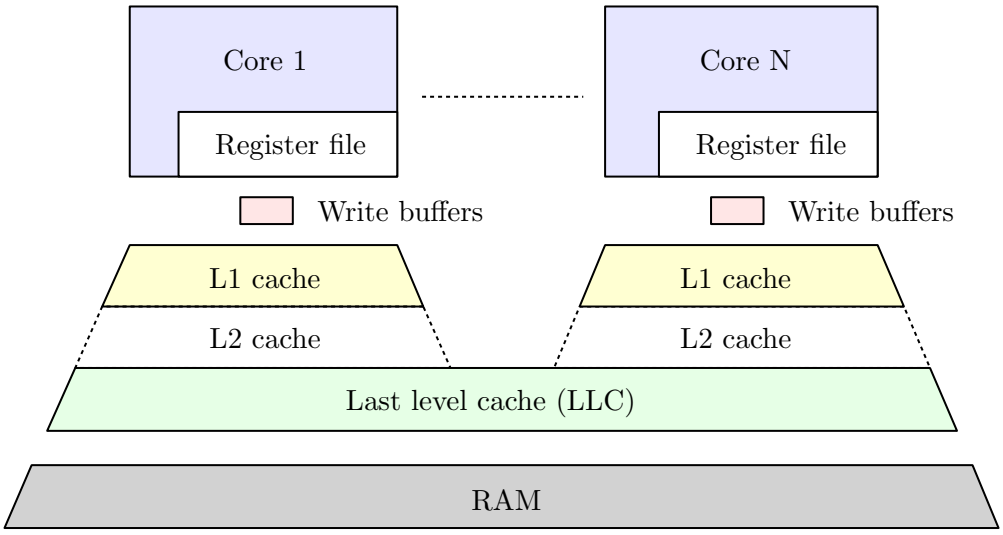


Figure 2.2: The cache hierarchy.

2.5 The memory hierarchy

Two design trade-offs in memory design are storage capacity and access speeds. For example, static RAM (SRAM) technology is 8 to 16 times faster than dynamic RAM (DRAM) technology, but DRAM provides a 4 to 8 times larger capacity [PH90].

Due to an increasing gap in processor performance and memory speed, modern CPU architectures include low-latency on-chip buffers for frequently accessed data [Goo83]. While off-chip memory exploits the high capacity of DRAM, these on-chip caches use SRAM for low-latency access. A hierarchy of caches are typically deployed with latency and capacity increasing with their distance from the CPU core (Figure 2.2). While there is generally a private cache for each core and a shared LLC, the number of caches in the hierarchy differs from system to system [ARM13].

Caches store data in *lines* of size 32 B to 128 B. When a CPU core accesses data that is not resident in the cache, a full cache line of data will be transferred to the cache from another source in the memory hierarchy. We refer to this as a *cache miss*. The data is typically found in lower levels of the memory hierarchy, but may also be served from the caches of other CPU cores.

A resident cache line must be evicted in order to make space for the arriving data. If the evicted line has been modified, the data must also be updated in lower levels of the cache hierarchy. Once the cache line has been filled, the data access request of the CPU can be served, and all future access to data within that cache line will be made with a reduced latency until that cache line is evicted.

Data coherency between caches is configured by SW. The cacheability and coherency of data access can be configured using the memory access attributes that are stored in the page tables of the virtual memory system. SW can configure whether or not data can be cached, and whether coherency should be maintained across per-core caches.

Alternatively, dedicated system instructions may be used to control coherency. SW can *invalidate* a cache line such that future accesses are served from lower levels of the cache hierarchy. This operation is particularly important when SW reads data that has been written by a peripheral that is not connected to the cache hierarchy. Similarly, SW can *clean* a cache line such that its content is written to lower levels in the hierarchy where they can be observed by such peripherals.

Cache maintenance instructions can only be executed by the OS [ARM05]. By cleaning or invalidating a cache line, an application can corrupt the data of another application. Therefore, an application must use the OS as a proxy for such operations via a system call.

2.5.1 Memory consistency

Transaction buffering is used throughout the memory hierarchy to allow the CPU to continue execution before a write transaction completes. To maintain consistency, future reads may be served from those buffers. Similarly, two writes to the same address can be merged such that only the final write transaction arrives at the target.

Write-merging can interfere with the correct operation of a peripheral. In most cases, write-merging is safe for applications. The intermediate value will not be observed by SW

after an updated value has been written [ARM05]. However, when those transactions are destined for a MMIO connected peripheral, the order and presence of those transactions may impact the behaviour of the peripheral. Such peripherals typically provide a control register for resetting the peripheral to a known state. If the sequence of writes to that register are merged, the reset signal will not be observed by the peripheral.

Write-merging can be controlled in a similar way to cache coherency. Memory access properties are provided by the page tables of the virtual memory system. On the ARM architecture, the attributes are strongly-ordered (SO) and device (DE). The SO attribute prevents transactions from using the cache and enforces a strict ordering of those transactions. In that case, the CPU stalls until the target acknowledges that the transaction is complete. The DE attribute prevents transactions from using the cache, but allows transactions to be merged. In that case, barrier instructions must be executed by the CPU to enforce the desired ordering. On the other hand, the DE attribute reduces CPU execution time because stalls are avoided for transactions that do not require a strict ordering [PS15].

2.6 Accelerator coupling

In this section I discuss historical and modern methods of integrating a CPU and FPGA into a system. I begin by describing the features of the traditional loose-coupling approach, where the FPGA is connected as a coprocessor. I then discuss the features of a system that integrates both FPGA and CPU such that accelerators have direct access to the CPUs register file. I conclude by describing the tightly-coupled architectures that is the foundation of this thesis.

2.6.1 Loose coupling

FPGA HW has historically been connected to a high-performance, general-purpose CPU via an accelerator card or module [Bit09, PS14, VKVF16, SMT+12]. The CPU offloads

work to a coprocessor that is implemented in programmable logic. A direct connection between FPGA and CPU provides a control interface to the CPU, typically via PCIe or AXI buses. The CPU in this case is the master and initiates all communication on the bus.

The control interface provides MMIO access to the CSR of the FPGA and instantiated accelerators. MMIO communication can be divided into three components: 1. the address of the register to be accessed; 2. the data to be transferred to or from the register; and 3. a signal to indicate that a transaction has been requested..

Transfers to the MMIO bus are synchronous in that the connected peripheral is made aware that the transaction is taking place. In some cases, the signal of a transaction has side effects, such as pushing register content to a first in, first out (FIFO) buffer rather than a register. Similarly, a read transaction may return data from that FIFO while also removing that data from the FIFO.

MMIO transfers for large quantities of data are CPU intensive because the CPU must manage the transfer of each word of data. Due to a limited amount of buffering, the CPU may stall waiting for previous transactions to complete before it can submit the next.

A direct memory access (DMA) engine was introduced to early systems to reduce the CPU overhead of data transfer to peripherals. In this case, dedicated HW is programmed to manage the transfer on behalf of the CPU. Early Industry Standard Architecture (ISA) implementations provided a fixed set of DMA controllers to the system. Each DMA engine had direct master access to both RAM and the FPGA. As a slave peripheral to the DMA controller, the accelerator was still unable to initiate its own transactions to RAM. For this reason, short transfers needed to be managed independently by the CPU.

Modern PCIe interfaces allow connected peripherals to initiate transfers to RAM themselves. Peripherals can now manage efficient short transfers, but for loosely-coupled architectures, such transfers are not coherent with the CPU cache (Figure 2.3a). Further to this, the latency of RAM access is large relative to RAM that is co-located with the FPGA on

the accelerator card. Therefore, a bulk transfer of all data is typically made to the local memory of the FPGA before processing begins.

An interrupt request (IRQ) provides a method for peripherals to signal completion. This allows the CPU to process background tasks while waiting for tasks on the accelerator card to complete. When a peripheral asserts an IRQ signal, CPU execution is interrupted. The CPU then determines the cause of the IRQ and schedules the registered event handler.

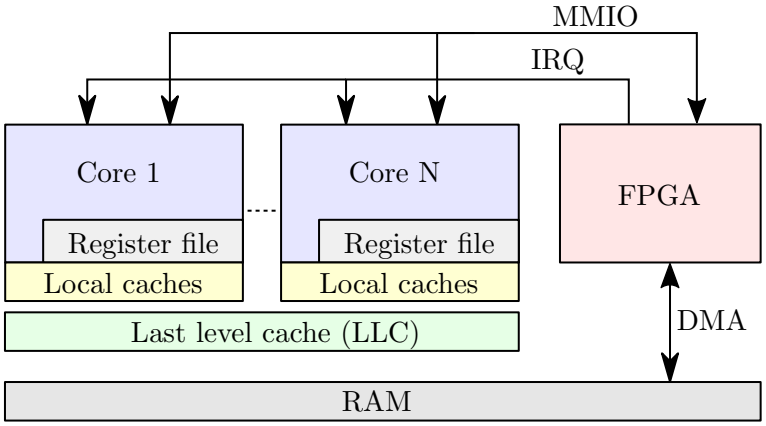
2.6.2 Integrated CPU and FPGA

An integrated architecture for CPU and FPGA has also been explored (Figure 2.3b). Such architectures provide the FPGA with direct access to the CPU instruction pipeline and register file. This architecture allows custom CPU instructions to be constructed and provides low-latency access to data stored in the CPU core [KBT10] [YRS05] [OK17] [YMHB00] [HW97] [WH95].

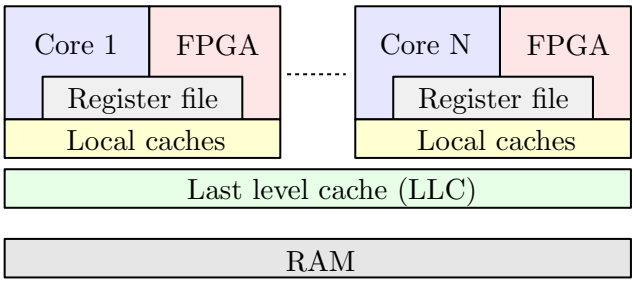
Yiannacouras et al. propose a framework for reducing the development effort of soft-core processor design [YRS05]. The engineer can select specific instructions that should be included in the processor.

Such augmentation of the processor's instruction set leads to an application specific integrated processor (ASIP) rather than a general-purpose processor. One counter-example is the dynamic instruction set computer (DISC) proposed by Wirthlin et. al. [WH95]. DISC uses DPR to change the active set of custom instructions to match the application's needs. While instructions are being reconfigured, the instruction pipeline of the CPU is stalled.

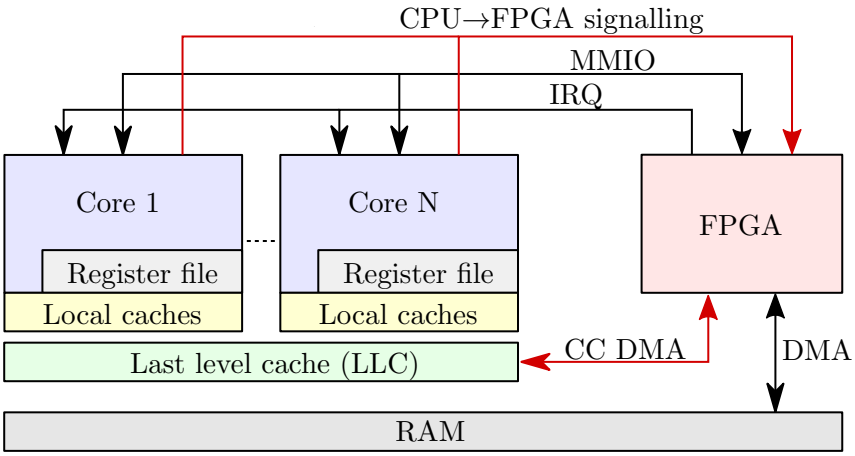
To my knowledge, such architecture are not commercially available for high-performance CPUs. This architecture is currently limited to processors that are implemented in the FPGA fabric itself, such as the MIPS, NIOSII and Microblaze soft-core processors. Such processors are limited by the performance of reconfigurable HW when compared to a fixed processor design.



(a) Traditional loose coupling.



(b) CPU integration.



(c) Tight coupling.

Figure 2.3: CPU-FPGA coupling architectures.

2.6.3 Tight coupling

As the focus architecture for my research, this section provides only an overview of tightly-coupled architectures. More detailed information is provided at relevant junctures throughout the thesis. The unique features provided by commercially available platforms will be discussed in the sections that follow.

Tightly-coupled architectures allow the FPGA to access data directly from CPU caches (Figure 2.3c). The FPGA is typically connected to the shared LLC of the system. The cache hierarchy maintains coherency between the LLC and the private CPU caches to avoid the need for SW-managed coherency. Not only does this method of integration reduce the latency of data access, it also avoids the CPU overhead of maintaining coherency. As a privileged operation, such overhead may include the cost of a system call to the OS to perform the operation on the application's behalf.

The range of low-latency signalling between CPU and FPGA is extended when using tightly-coupled systems. While IRQs are still supported, the CPU can signal to the FPGA in the same way that it signals to other CPU cores, allowing it to be a peer to other processors rather than merely a coprocessor.

On some systems, it is possible to extend the cache hierarchy into the FPGA by enabling CPU caches to retrieve data from the FPGA [Int19b]. A signal from the CPU cache to the soft FPGA cache can be available to notify the soft cache when a cache line has been written to by a CPU. In that case, the cache line can be invalidated in the FPGA to maintain cache coherency.

2.7 Commercial tightly-coupled systems

In this section, I identify commercially available systems that support a tight coupling between FPGA and CPU. I describe the unique features of each architecture and how those features may be exploited. I conclude this section with a matrix of features that each

systems provides.

2.7.1 IBM

IBM provides a Coherent Accelerator Processor Interface (CAPI) to PCIe connected accelerators [IBM14]. The FPGA card is expected to implement a POWER Service Layer (PSL) shell in the programmable logic. The shell provides a layer of abstraction for the PCIe interface and includes an IOMMU and local coherent cache.

The cache and the IOMMU provide a uniform programming model with SW. The application can pass pointers due to the unified addressing between HW and SW and need not consider cache coherency. The local cache also serves to further reduce the latency of main memory by storing recently accessed data.

The PSL includes task management functions, such as job control and preemptive context-switching. Although a method is provided to send a signal to an accelerator, that method is via PCIe and hence delivered with high latency.

2.7.2 Intel

Intel provides a tightly-coupled CPU and FPGA using their QuickAssist QPI platform [Int19b]. Rather than a PCIe-connected accelerator card, the QuickAssist platform enables an FPGA to be inserted into a second CPU socket of the motherboard. Communication between sockets is via a low-latency QuickPath Interconnect (QPI) bus. For improved data throughput, data buses are 512 bit wide.

Like IBM, Intel provides a shell, named the FPGA interface unit (FIU), to be instantiated in the FPGA. That shell provides virtual channels that accelerators use to access RAM and a coherent soft cache. The FIU also provides a channel for MMIO transactions from the CPU to the accelerator. In contrast to CAPI, the shell does not provide task management services. Instead, these protocols and logic must be implemented entirely by

the application.

Although Intel provides a fixed IOMMU, Intel's shell also provides a soft IOMMU to improve flexibility. The fixed IOMMU supports translations for only one device. The soft IOMMU provides independent translations for many concurrent accelerators.

QuickAssist provides a low-latency signalling mechanism between CPU and FPGA via cache line invalidation notifications. The accelerator must first write to cache-coherent memory to ensure that data is present in the soft cache. When SW writes to a corresponding cache line, the line is evicted from the soft cache of the FPGA to maintain coherency. The eviction additionally transfers a "uMSG" packet to the accelerator to inform it of the eviction. From this uMSG, the accelerator receives a low-latency signal to inform it that SM content has been updated.

While direct IRQ signalling is not supported, the shell allows accelerators to tunnel IRQ signals through the provided virtual channels.

As part of Intel's hardware accelerator research program (HARP), numerous studies have evaluated the performance of the Intel's QuickAssist platform. Chang et al. improved the performance of genome sequencing by 4× [CCC+16]. István et al. demonstrated a 4× throughput for pattern matching in databases when compared to similar SW-only implementation [ISA16].

Choi et al. evaluate memory bandwidth and latency of a QuickAssist platform and a loosely-coupled PCIe-based accelerator card [CCF+16]. They find that QuickAssist provides 3.4× more throughput than the PCIe connection of the accelerator card (Table 2.1). However, local memory on the accelerator card provides 0.4× more throughput than the QuickAssist platform. From this we infer that, for long running tasks, the overhead of a bulk transfer of data to the FPGA-local memory of a loosely-coupled system could improve overall throughput. However, for short-running tasks, the high latency of that transfer may outweigh the improvement in bandwidth.

Table 2.1: Summary of CPU-FPGA communication bandwidth and latency for PCIe-based and QPI-based platforms [CCF+16].

Coupling	Method	Latency (μs)	Throughput (GB/s)
Loose	PCIe	160	1.6
	FPGA-local DRAM	0.54	9.5
Tight	QPI	0.36	7.0

Weisz et al. use the QuickAssist platform to explore the opportunities for cache-coherent architectures in pointer chasing applications [WMW+16]. Pointer chasing is fundamental to image and speech recognition, as well as machine learning. The proposed solution involves the CPU performing the traversal of the tree while feeding the FPGA pointers for tree payload data. Tree meta-data can be shared via cache-coherent QPI while payload data can be sourced from RAM to reduce cache pollution.

2.7.3 Xilinx

Xilinx has released the Zynq family of devices that tightly couples high-performance, embedded processors with programmable logic. The Zynq is the embedded SoC that we used to explore all of the ideas presented in this thesis. We therefore provide more detail on the Zynq than other commercially available systems. The Zynq has many similarities with Intel's equivalent range of tightly-coupled embedded systems. Both vendors use ARM processors, which define the interface between CPU and FPGA.

More precisely, we used Avnet's Zedboard to evaluate our design [Avn19]. The Zedboard is a low-cost development platform that features the Zynq-7020 SoC. Zynq-7000 series SoCs provide dual ARM Cortex-A9 application processors and on-chip programmable logic. Communication between the ARM cores and the programmable logic is achieved via a range of ARM AXI communication buses [ARM11]:

- **General-purpose (GP) AXI3**: The GP AXI ports offers 32-bit data transfers with bulk transfer sizes of up to 64 bytes. The Zynq-7000 series provides two GP ports in both master and slave interface configurations. The masters provide MMIO

interfaces from the CPU to the FPGA while the slave interfaces provide the FPGA with access to RAM.

- **High-performance (HP) AXI3**: Four HP AXI port offers 64-bit data transfers with bulk transfer sizes of up to 128 bytes. Zynq-7000 series provides the HP port only as a slave interface from FPGA to RAM. The HP port provides improved throughput over the GP slave equivalent due to the increased bus width and buffer depth.

- **Accelerator coherency port (ACP) AXI3**: Like the HP port, the ACP provides a slave interface port that offers 64-bit data transfers with bulk transfer sizes of up to 128 bytes. Unlike the HP port, transfers can optionally be cache-coherent with the CPU. In the case of the ACP, the slave device is the snoop control unit (SCU), which connects each ARM core to the memory hierarchy and maintains cache-coherence between the connected masters and ACP-connected peripherals [ARM12]. When the accelerator performs a read transaction, the SCU can retrieve the appropriate data from the shared L2 cache or the L1 cache of any connected CPU. When the accelerator performs a write transaction, the SCU writes the appropriate data into the L2 cache and invalidates any corresponding L1 cache lines in all CPUs.

It is not possible to implement a coherent soft cache on Zynq-7000 series devices. The ACP does not provide the necessary mechanisms for serving data to the CPU or for the SCU to broadcast cache line invalidations to the FPGA. However, the more recent Zynq Ultrascale+ does provide such mechanisms [Xil19a]. The Ultrascale+ also provides a fixed IOMMU while a soft IOMMU must be implemented on the Zynq-7000 devices.

Both the Zynq-7000 series and the Zynq Ultrascale+ provide low-latency signalling mechanisms between CPU and FPGA in both directions. In addition to traditional IRQ support, the Zynq extends inter-core signalling to the FPGA fabric. This allows the integration of a soft-core within the FPGA that is considered a peer to the fixed-core processors. When a CPU core executes the wait for event (WFE) instruction, it is placed in a low-power mode until another CPU executes a send event (SEV) instruction. From programmable logic,

the accelerator can issue an event by toggling the *EVENT_EVENTI* signal. Similarly, the accelerator can observe the execution of the SEV instruction as a toggling of the *EVENT_EVENTO* signal.

The Zynq-7020 SoC provides maximum CPU and FPGA operating frequencies of 666 MHz and 464 MHz respectively [Xil]. However, the interface between processor and programmable logic is limited to 240 MHz. Significantly higher frequencies can be expected with more modern devices, such as the Zynq Ultrascale+, which provides maximum CPU and FPGA operating frequencies of 1500 MHz and 600 MHz respectively [Xil20b].

Many have studied the capabilities of the tightly-coupled Zynq [SWWB13] [PS15] [SSS15]. Sadri et al. evaluated the energy and performance of a finite impulse response (FIR) filter using each port available on the Zynq [SWWB13]. They found that the ACP offered similar performance to an HP port while transactions could be served from the cache. When the dataset grew beyond the size of the cache, ACP bandwidth fell from 1650 MB/s to 650 MB/s. The authors also observed a reduction in ACP performance when a SW task was scheduled that competed with the accelerator for cache bandwidth. Other studies have found similar results when comparing Intel's competing Cyclone device with the Zynq [MRAF18] [CFMRAF17] [MSFRA15].

Powell et. al. explored the effect of coherency and buffering attributes on transactions [PS15]. The authors found that these memory attributes have a significant impact on performance. DE transactions completed faster than SO because completion is reported once the transaction is accepted by write buffers in the interconnect. To enforce the desired strict ordering, SO transaction only report completion when they have arrived at their destination.

2.7.4 Commercial systems feature matrix

A summary of the features supported by the described commercially-available tightly-coupled systems is presented in Table 2.2.

Table 2.2: Commercial tightly-coupled systems feature matrix.

System	MMIO	Cache-coherent	FPGA cache	IOMMU	Signalling
IBM CAPI	PCIe	✓	✓	soft (shell)	IRQ
Intel QuickAssist	PCIe	✓	✓	soft (shell)	uMSG/IRQ
Xilinx Zynq-7000	AXI	✓	✗	soft	SEV/IRQ
Xilinx Zynq US+	AXI	✓	✓	fixed/soft	SEV/IRQ

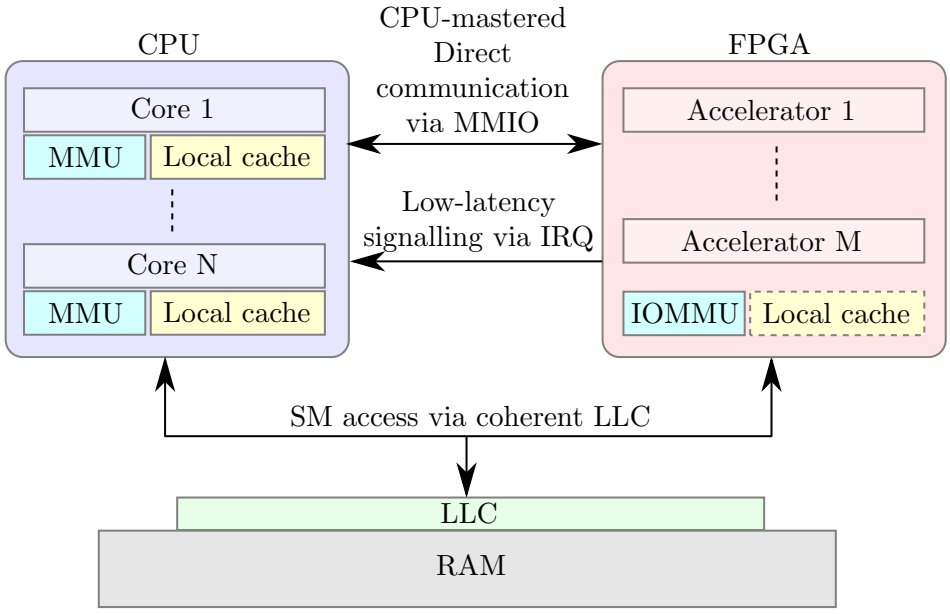


Figure 2.4: Hardware model.

2.8 Hardware model

The HW platform considered in this thesis is assumed to be composed of one or more general-purpose processors that support protected execution modes and MMUs (Figure 2.4). These features are required for an operating system that provides a concurrent execution environment for multiple general-purpose applications.

The HW system is also assumed to provide an FPGA for configurable HW accelerators. The FPGA and the CPU are assumed to communicate directly via MMIO, and via SM. SM communication is assumed to be cache-coherent in order to support low-latency transfers and to avoid the CPU overhead of cache-maintenance operations [KL08].

Table 2.3: Vendor support for assumed hardware model.

	MMIO	IRQ	Cache-coherent SM
Xilinx Zynq-7000	AXI	✓	AXI to L2
Intel QuickAssist	PCIe	✓	PCIe to L3
IBM CAPI	PCIe	✓	PCIe/CAPI to L3

We assume that the system includes an IOMMU to provide a unified address space between SW and accelerator. That IOMMU may be provided in fixed HW or in the programmable logic. The IOMMU simplifies the programming model, memory management and avoids the overhead of manual memory translations by short-running tasks.

The IOMMU requires IRQ support to interrupt the CPU when a virtual memory translation is not available. This is important when the OS loads applications lazily or moves memory pages to and from disk.

This thesis will demonstrate the usefulness of low-latency signalling from CPU to FPGA is limited due to its low bandwidth. We find that the latency of MMIO is relatively low and allows some context to be transferred with the signal. Therefore we do not include low-latency signalling from CPU to FPGA in our model.

The assumed HW model is provided by the major FPGA vendors (Table 2.3).

Chapter 3

Fine-grained transfers on tightly-coupled Zynq

In this chapter, I evaluate the ability of the tightly-coupled Xilinx Zynq-7000 series All Programmable System on Chip to support fine-grained interactions between CPU and FPGA. The target for my case study is the SW task scheduler of a microkernel OS. Fine-grained interaction in this case involves the insertion and removal of a SW thread handle from a priority queue that is implemented in HW (Figure 3.1). Communication between the kernel SW and the HW-resident task queue involves the transfer of a single 32-bit word in both cases.

The task scheduler is the primary function of all OS. Not only is the scheduler invoked periodically to ensure CPU time-sharing between threads, it is also invoked on demand to facilitate communication or synchronisation between two threads, and when scheduling high-priority tasks in response to critical external events.

A large amount of effort is invested into the optimisation of the scheduler. The scheduler's role is to facilitate the concurrent execution of many tasks, rather than contributing to the processing required by those tasks. For that reason, the execution time of the scheduler is considered a system overhead that all users seek to minimise.

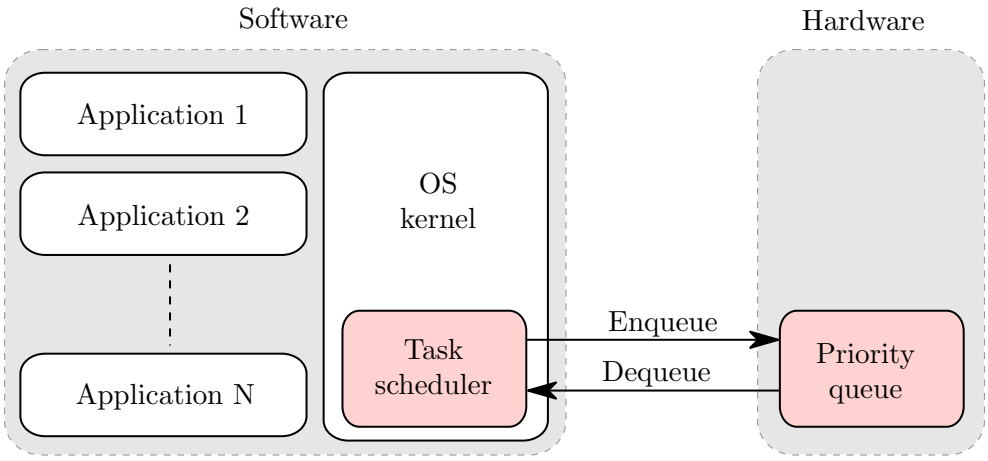


Figure 3.1: Accelerated OS kernel scheduler architecture.

Given that the OS scheduler is highly optimised and involves operations that are very short in nature, we expect it to be challenging to gain any improvement in execution time through HW acceleration. The choice of this case study motivates us to optimise the communication methods used to the best of our ability. By carefully selecting the fine-grained communication methods used between HW and SW, we were able to achieve a 5.5% reduction in execution time for synchronous communication between two threads.

Anticipating the difficultly of accelerating short-running tasks, the study presented in this chapter was primarily motivated by a desire to determine whether or not migrating SW functions to programmable logic reduces the execution time jitter of a SW task. Reduced jitter is particularly important for real-time systems that require a known bound on execution time. Large jitter leads to pessimism in the worst case execution time (WCET) – a metric that determines whether or not an external event achieves a timely response.

Jitter is caused by the non-deterministic nature of execution on modern superscalar processors. Instruction throughput enhancing features, such as the cache hierarchy and branch predictor, reduce the amount of time that the CPU spends idle. This is typically done by predicting program and data flow with stochastic heuristics. For example, the instruction and data caches are typically configured for random cache-line replacement when new data requires space in the cache (Section 2.5). That random replacement causes jitter in execution time as the choice of cache line determines future data access completion

times. When SW functions are implemented in HW, the associated code and data no longer need to occupy the cache. This improves the likelihood that critical OS or application code stays resident in the cache.

In this study we also found that HW acceleration provides a significant improvement in execution time variance. The migration of the SW task scheduler into HW removed most sources of non-deterministic execution time and reduced execution time variance by 58%. Unfortunately, further improvement was limited by the dominating influence of the branch predictor of the CPU.

3.1 Contributions

The contributions of this work are:

- To show that tightly-coupled CPU-FPGA systems can improve execution time variance for high-performance real-time systems.
- Tightly-coupled CPU-FPGA systems can improve the performance of inter-process communication by offloading the management of the SW task scheduling queues from the OS SW to HW;
- Tightly-coupled CPU-FPGA systems extend the range of applications that can benefit from reconfigurable custom computing. Without the use of cache-coherent interconnects, a performance benefit was not observed.

3.2 Publications

Parts of this work were peer-reviewed and accepted into the International Workshop for Heterogeneous High-performance Reconfigurable Computing (H^2RC) in 2015 and in ACM Transactions on Reconfigurable Technology and Systems (TRETS) in 2019. While

this research is my own work, it was only made possible with the editorial support and shepherding provided by my coauthor and supervisor, Oliver Diessel.

[KD15] A. Kroh. and O. Diessel. Towards OS kernel acceleration in heterogeneous systems. *First International Workshop on Heterogeneous High-performance Reconfigurable Computing (H^2RC)*, 2015.

[KD19] A. Kroh and O. Diessel. Efficient fine-grained processor-logic interactions on the cache-coherent Zynq platform. *ACM Trans. Reconfigurable Technol. Syst.*, 11(4):25:1–25:22, January 2019.

3.3 Prior work

The migration of an OS kernel scheduler from SW to HW has been studied prior to this work, primarily as a means of improving execution time and jitter in real-time systems. However, prior work has generally been limited to either soft-core systems [OLAH13, DG12], simulated HW [NRL07, LSV05, KSM03] or loosely-coupled systems [MB02, DT13]. To the best of our knowledge, the performance benefits of this migration has not been evaluated using low-latency, cache-coherent communication with fixed-core processors.

Ong et. al. augmented a soft-core NIOS-II processor with a HW-accelerated task scheduler [OLAH13]. The accelerator was connected to the system interconnect bus and additionally provided a periodic timer and processor interrupt. The authors observed a 72% improvement in inter-task communication execution time and a reduction in IRQ handler jitter from 25.4% to 1.59%. Although these improvements are significant, the use of a soft- rather than fixed-core processor penalises the CPU in the evaluation. It is well known that fixed-circuits have higher maximum operating frequencies when compared to programmable logic.

HW-assisted scheduling has shown promise in symmetric multi-core architectures. Using cycle-accurate simulation, Nácul et al. showed that HW-assisted scheduling reduces the

context switch time between two threads from 10,000 CPU cycles to 947 CPU cycles [NRL07]. While the scheduler chooses the next thread for execution, it has no access to the CPU register file: saving and restoring CPU state must be done in SW. The proposed system uses dedicated ports on an ARM926EJ-S processor for communication between the CPU and the scheduler, implemented in HW. The authors measured throughput improvements in graphic filtering and network packet processing applications to be 46× and 10×, respectively.

Mooney et al. proposed a modular OS framework [MB02]. In their work, the system engineer can choose between HW or SW equivalent implementations for OS subsystems. Key subsystems include dynamic memory management, locking, and deadlock detection. HW-assisted locking aims to improve both execution time and the predictability of lock access times. By moving locking to HW, deadlock detection improves system safety without significant run-time overheads. Simulated HW experiments showed that HW-assisted OS functions can provide speed-ups of 27% or more for database applications.

HW schedulers can be provided as programmable logic [KSM03], or as dedicated circuits within an ASIC [NRL07]. While ASIC implementations provide fixed scheduling policies, programmable logic allows for flexible, application-specific scheduling policies that can be swapped on-demand.

The deployment of HW schedulers has been explored as both independent coprocessors and integrated CPU features. In the latter case, the scheduler has direct access to the execution pipeline and register file of the CPU. This allows the scheduler to swap the entire thread context in a single cycle without degrading CPU pipeline performance [DG12]. Such an architecture requires modification of the CPU itself. Modern SoCs, such as the Xilinx Zynq and Intel Cyclone V, offer programmable logic and a high-performance general purpose fixed-core processor on a single die, but these systems do not provide direct access to the CPU register file.

Dahlstrom et. al. implemented a HW-accelerated SW task scheduler on the Zynq SoC [DT13]. The focus of that work was on obscuring the view of thread context from other threads by storing it in programmable logic. The study showed up to 50% speedup

(1500 CPU cycles), even though the entire thread context (68 bytes) had to be transferred between CPU and accelerator on every context switch. Although that work used a tightly-coupled CPU-FPGA system, it did not explore the use of the low-latency, cache-coherent communication that such coupling provides.

3.4 System architecture

In our work, we decided to investigate accelerating the seL4 microkernel [KAE+14]. seL4 kernel operations are very short in nature and difficult to accelerate using the traditional CPU-offload model. Additionally, the kernel is the central gateway for all application resource management and communication: the performance of the kernel impacts all hosted applications.

seL4 has a complete WCET analysis [BSC+11, SKH16] and has recently been extended for real-time applications [LMAH18]. seL4 is considered to be a microkernel because operating system services and device drivers are implemented as user applications rather than being provided directly by the kernel. A key advantage of this approach is that only a small amount of SW must be trusted to ensure the correct operation of the system. Drivers, servers and applications all execute in a low-privilege operating mode of the CPU and are isolated by the MMU HW of the CPU (Section 2.4). Because of this isolation, inter-process communication (IPC) is used frequently to communicate between tasks.

We decided to attempt to accelerate the seL4's task scheduler because it is a very frequently used function of the kernel. Although the scheduler is not a long-running operation, the performance of the scheduler is critical to IRQ handler latency and efficient IPC between client-server application SW. Once the system has been initialised, the kernel provides 3 key functions, all of which can result in an invocation of the kernel scheduler:

1. *IRQ delivery.* When the kernel receives an IRQ exception, the kernel unblocks any thread that is waiting for the IRQ. If the unblocked thread is of a higher priority

than the currently active thread, the kernel must reinsert the active thread into the scheduling queue and replace it with the new highest priority runnable thread.

2. *Preemption IRQ.* The preemption IRQ ensures the fair sharing of CPU time between threads of the same scheduling priority. When the preemption IRQ arrives, the kernel reinserts the current thread into the scheduling queue and replaces it with the next runnable thread in round-robin order.

3. *Inter-Process Communication.* (IPC) is a primitive for data transfer and synchronisation between threads. When a thread sends an IPC request to another thread, the kernel blocks the sender and inserts the receiving thread into the scheduling queue. The scheduler is then invoked to choose a new thread for execution.

seL4 provides a *fixed-priority preemptive scheduler* such that a thread never executes while a runnable thread of higher priority exists in the system. It has been carefully tuned for low-latency IPC through SW optimisations that consider both the number of instructions executed, data locality and cache footprint. We therefore expect it to be challenging to gain a benefit from HW acceleration.

3.4.1 Software scheduler

The set of runnable threads in the seL4 microkernel is implemented as a doubly-linked list with one list for each thread priority (Figure 3.2). The kernel maintains the *next* and *previous* pointers of this list along with the thread context as part of the thread control block (TCB) of each thread. The kernel appends a thread to the tail of its associated list when it has exhausted its allocated execution time interval or when it transitions from the blocked to the runnable state. If the thread has been preempted, perhaps because a higher priority thread has become unblocked and is now runnable, the kernel records the remaining execution time of the thread and adds the thread to the head of its associated list, rather than the tail. The kernel maintains a set of head and tail pointers for each priority in a global structure known as the *ksReadyQueues*. When the kernel invokes the

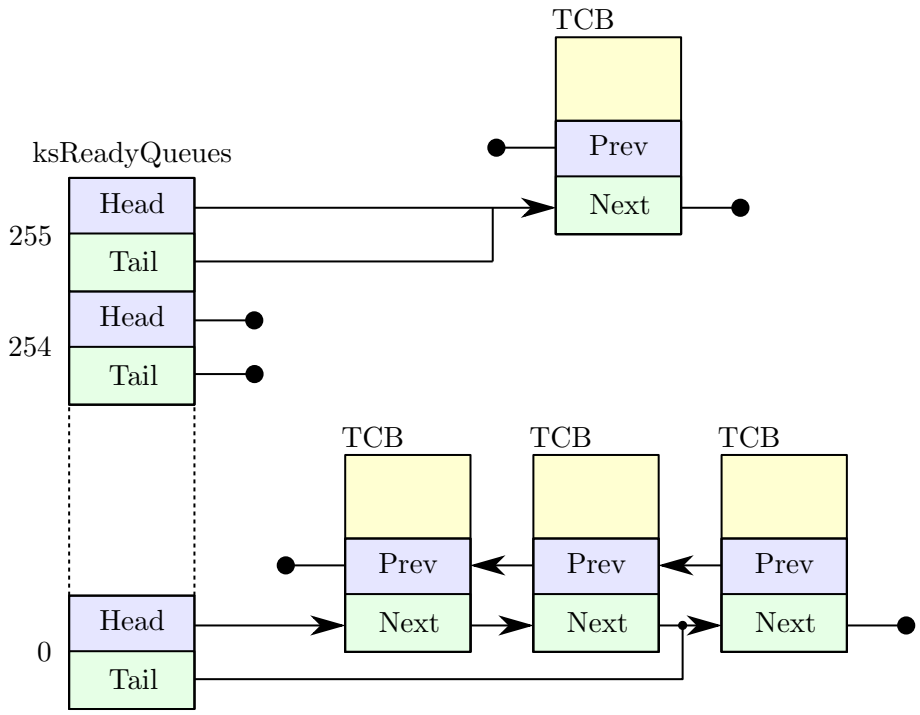


Figure 3.2: Software architecture of the legacy task scheduler.

scheduler, it walks the *ksReadyQueues* from the highest priority (255) to the lowest priority (0) until it finds a non-empty list of runnable threads. If a non-empty list of runnable threads is found, the scheduler removes the thread at the head of this list and marks it as the active thread. If no runnable thread is found in the system, the kernel schedules an implicit *idle* thread until an external IRQ causes a waiting thread to become runnable again.

The seL4 kernel implements a *fastpath*, a hand-optimised path for common operating system calls. The *fastpath* allows IPC from a low-priority sender thread to a higher- or equal-priority receiver thread to complete without costly scheduler invocations. Under these conditions, because the kernel implements a *fixed-priority preemptive scheduler*, the kernel knows that there is no runnable thread with a higher priority than the IPC sending thread. If the receiver is blocked waiting for this IPC and is of a higher- or equal-priority to the sender, the kernel knows that the receiver will be the new highest priority runnable thread in the system. For this reason, the sender can be blocked and the receiver can

immediately become the new active thread without invoking the task scheduler of the kernel.

Since the commencement of this work, the SW architecture of the kernel scheduler was further optimised by supplementing the design with a two-level bitmap lookup. Each bit in the second level bitmap corresponds to one of 32 priorities. If the bit is set, the associated priority contains at least one runnable thread. If the bit is clear, there are no runnable threads for the associated priority. In the same way, the first-level bitmap reflects the presence of a runnable thread in each group of 32 priorities. The scheduler uses the single-cycle Count Leading Zeroes *CLZ* instruction on the first level bitmap and maps the result to the appropriate second level bitmap. The scheduler repeats this process on the second level bitmap to find the highest priority at which a runnable thread can be found. Finding the highest-priority runnable thread thus becomes a constant time operation, irrespective of its location within the *ksReadyQueues*.

We consider both SW implementations in our evaluation. Although the bitmap lookup improves the average execution time of the scheduler, the scheduler performs better without bitmap augmentation when thread priorities are high.

3.4.2 Accelerator design

The HW design must provide the same features as the SW design to ensure compatibility. It must allow a thread to be inserted at both the head and tail of a selected *ksReadyQueue* and allow a thread to be removed from the head of the highest priority non-empty *ksReadyQueue*.

A simple structure that satisfies these design goals is a priority queue. While much research has been undertaken on priority queue implementations [MLC14, IK07], the behaviour of insertions with equal priority are generally ill-defined. In our case, it is important that threads of equal priority are removed from the priority queue in the order in which they were inserted. Since our research is not focused on priority queue HW design and implementation, we used a trivial implementation to explore our ideas.

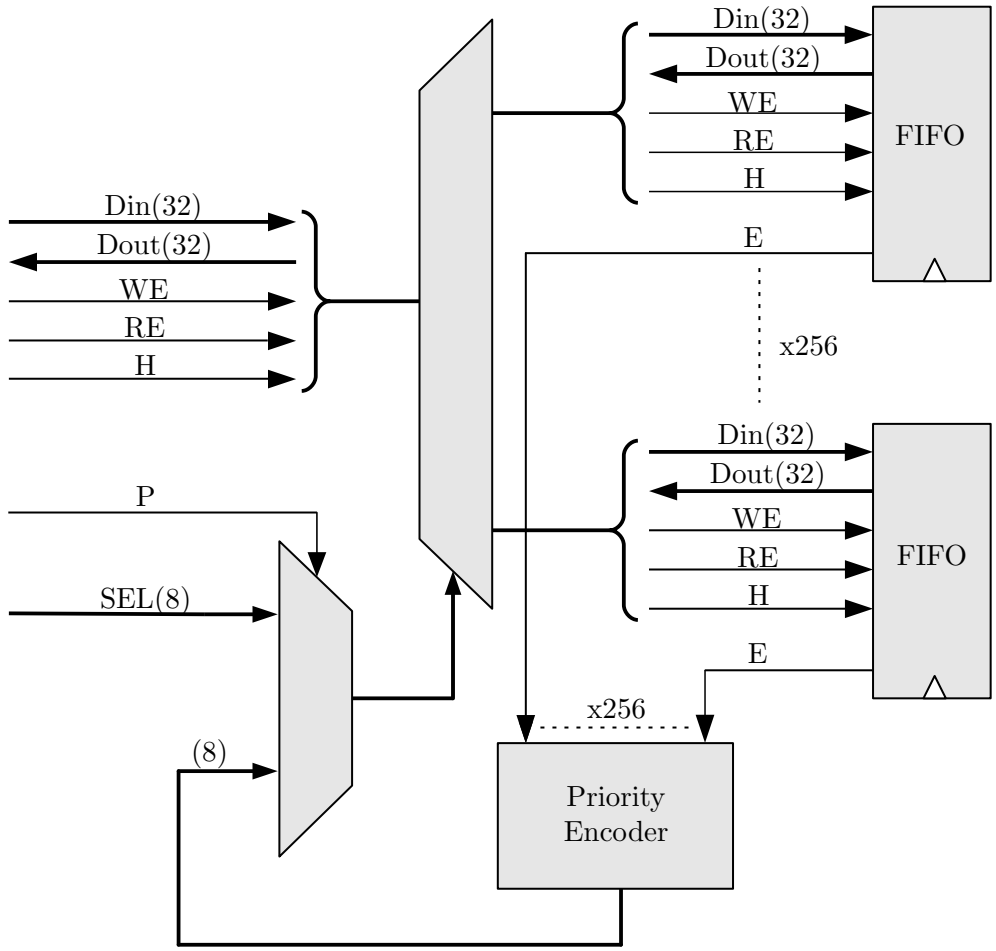


Figure 3.3: Hardware architecture of the priority queue.

We used a HW architecture that closely follows that of the SW architecture for this study (Figure 3.3). We replaced the *ksReadyQueues* by FIFOs, where FIFO data represents a reference to the TCB of a thread in main memory. By transferring only the location of the TCB in main memory, we reduced the throughput requirement for priority queue transactions. This also preserved the ability of the CPU to optimise reads and writes to cacheable global memory when accessing thread context.

The H signal of the priority queue allows the scheduler to add a thread to either the head or the tail of a FIFO. Write enable (WE) and read enable (RE) signals control the addition (push) or removal (pop) of a FIFO entry. If the scheduler asserts neither WE nor RE, a read operation returns the appropriate entry from a FIFO without removal. The scheduler

uses the *SEL* signal to select a specific priority (FIFO) for the priority queue operation. Each FIFO also provides an E signal, which indicates if the corresponding FIFO is empty. Each E signal is routed to a 256-bit asynchronous priority encoder. When the state of any E signal changes, the priority encoder output updates to reflect this change before the next rising edge of the subsystem clock. The priority encoder allows the scheduler to both identify and remove the highest priority runnable thread from the set of runnable threads in a single clock cycle. With the P signal asserted, the priority queue ignores the *SEL* signal and uses the priority encoder output in its place to select the highest priority non-empty FIFO as the target of the transaction.

The HW implementation provides acceleration by offloading the task of manipulating the priority queue from SW. The SW implementation of the *ksReadyQueues* requires that SW maintains a doubly-linked list of threads for each priority. Our HW implementation relieves the burden of list maintenance from SW by allowing SW to manipulate the head or tail of a *ksReadyQueue* with a single transaction to the accelerator. Additionally, the highest priority thread in the *ksReadyQueues* can be requested and removed from the schedule in a single transaction. This eliminates the need to search the scheduling queue or maintain a hierarchy of bitmaps.

We acknowledge that the use of fixed-size FIFO components presents scalability concerns in the design, however, more scalable priority queue implementations can also be supported. One must only ensure that the number of clock cycles required to update the highest priority runnable thread is less than the number of clock cycles between scheduler transactions. Alternatively, FIFO content can be stored in off-chip RAM with high-priority runnable threads cached in block RAM for low-latency access.

3.4.3 Target system hardware

In our study, we considered the GP AXI3 master and ACP AXI3 slave interfaces of the tightly-coupled Zynq-7020 SoC (Section 2.7.3). The GP master port is the only port that provides direct communication between the CPU and the accelerator. Access to

GP-connected peripherals were made with the DE memory attribute to allow transactions to be buffered. The ACP provides the best performing SM interface due to its coherency with the CPU cache. All transactions to SM were configured to make use of the L2 cache of the CPU.

Communication between CPU and accelerator has three components:

1. A **command** must be transferred to the accelerator to communicate the desired action. An example of such a command is to push a thread to the head of a particular scheduling queue.

2. **Data** must be transferred to or from the accelerator. In our case, this data represents a reference to a thread in main memory.

3. A **signalling** mechanism is needed to inform the accelerator that a new command is available and that a response is expected.

If we connect the accelerator as a slave peripheral on the GP AXI bus, signalling is a bi-product of the AXI handshaking protocol during a transfer. If we connect the accelerator as a master peripheral, communication is via SM and we must provide some other method for signalling. We can use a second GP master port for this purpose, or we can use the *EVENT_EVENTO* signalling method. SW can assert a single wire in the CPU for exactly one CPU clock cycle by executing the *SEV* instruction. This signal is generally used to signal an event to other embedded processor cores. However, the programmable logic can observe a toggled variant of this signal. The processor toggles the *EVENT_EVENTO* signal for each execution of the *SEV* instruction.

The following subsections describe the connection of the accelerator to the GP AXI master port and the ACP AXI slave port.

11	10	9	8	7	6	5	4	3	2	1	0
P	SEL								H	0	0

Figure 3.4: Address mapping of GP accelerator peripheral.

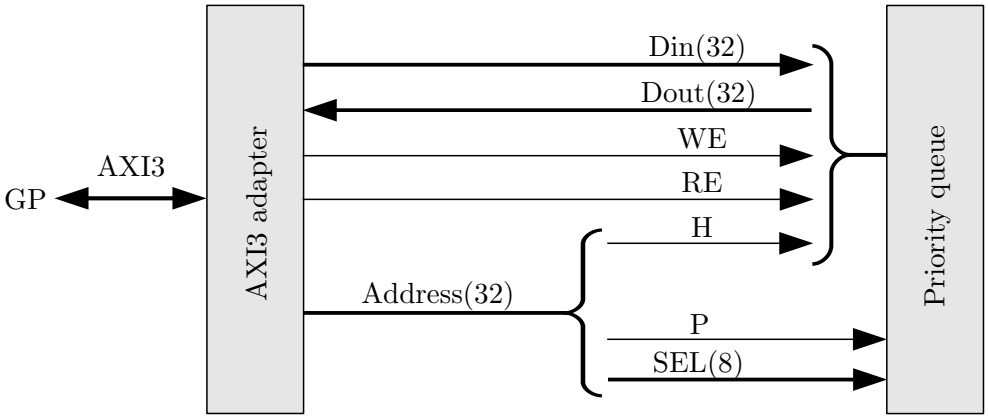


Figure 3.5: Architecture of GP HW scheduler communication.

3.4.4 GP connected accelerator

We designed the accelerator as a slave peripheral with the CPU communicating directly with the accelerator through MMIO. The accelerator decodes the address that is provided by the CPU to produce a command for the scheduler transaction as shown in Figure 3.4. Bits 0 and 1 of the address are reserved for word alignment, bit 2 is mapped to the *H* signal, which selects between the head or tail of the queue, and bits 3 through 10 are mapped to the *SEL* signal, which selects the priority for the transaction. Additionally, bit 11 is mapped to the *P* signal, which instructs the accelerator to ignore bits 3 through 10 and instead select the highest priority non-empty queue for the transaction. We developed an AXI adapter component to translate the complex AXI handshake protocol into a simple *write enable* (*WE*) and *read enable* (*RE*) signalling mechanism (Figure 3.5).

SW inserts a thread into the schedule at a specific priority by initiating a single write to memory. A handle to the thread is transferred as the data portion of the transfer, while the desired priority is encoded in the address portion of the transfer. Similarly, SW removes a thread from the schedule by initiating a single read request from a specific memory address. A single read transaction can also be used to both identify and remove the highest priority

Table 3.1: Command and data encoding for ACP-based scheduler accelerator operations.

Operation	31	30	29	28-10	9 - 8	7 - 0
Enqueue	0	0	H	&Thread	0	SEL
Dequeue	1	0	H	0	0	SEL
Dequeue (highest priority)	1	0	0	0	0	0

runnable thread by executing a read instruction with bit 11 of the source address set.

We clocked the HW implementation of the *ksReadyQueues* directly from the GP AXI bus clock at 100MHz.

3.4.5 ACP connected accelerator

Communication must be achieved indirectly through SM when using the ACP port (Section 2.7.3). The ACP bus is 64 bits wide: two words can be transferred per clock cycle. We assigned one word to represent the handle to the thread. The constraints of a seL4 thread object enforce an alignment of 10 bits and the object must be accessible within the range (0xE0000000 to 0xFFFFFFFF). These constraints provided 12 bits for encoding the command (Table 3.1).

The SM approach prevented us from using the AXI handshaking protocol to signal the accelerator when a new command was provided. Instead, we used the *SEV* processor instruction, which the accelerator detects as a state toggle of the *EVENT_EVENTO* wire in programmable logic (Section 2.7.3). The transfer of the signal and data in this case was decoupled; we had to ensure that the data was observable by the accelerator before the signal of its presence arrived. We used the *data synchronisation barrier (DSB)* processor instruction to ensure that any *store* operation had completed before the *SEV* instruction was executed.

In HW, we extended the priority queue implementation to include a trivial finite state machine (Figure 3.6). After the processor signals the accelerator, the accelerator reads the provided command and data from SM by issuing a 64-bit read transaction on the ACP

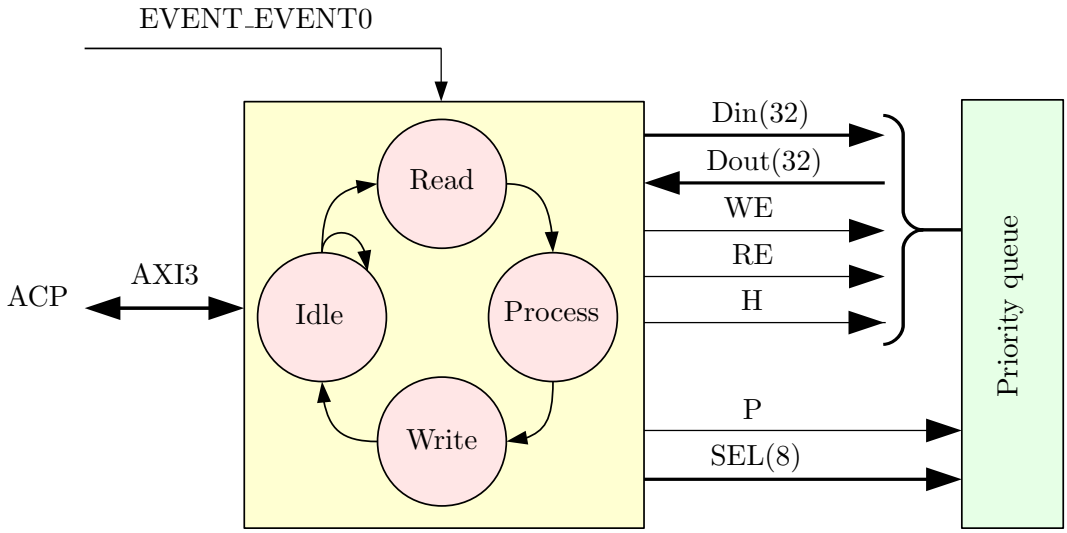


Figure 3.6: Architecture of ACP HW scheduler communication.

AXI bus. The data portion of the transfer is mapped directly to the data input of the priority queue while the command is decoded and routed to the relevant control wires. The priority queue operation is completed by the next clock cycle. The accelerator then sets the priority queue control signals such that the highest priority thread is reported. Finally, the accelerator issues a write transaction to the ACP. This write clears the command that was provided by SW and reports the highest priority runnable thread through SM.

Once a scheduler transaction has completed any command, the first word in the 64-bit SM region is cleared to signal completion and the second word is speculatively filled with the highest priority thread in the schedule. In this way, the CPU can retrieve the highest priority thread from the low-latency cache at any time and then issue the appropriate command to remove this thread from the schedule. By always storing the highest priority runnable thread at a pre-determined location, we make it possible for the CPU to retrieve this thread from the low-latency cache as soon as it is required. The CPU thus continues program execution while the accelerator processes the request and replaces the highest priority runnable thread in preparation for the next scheduling event.

We connected the ACP AXI accelerator described above directly to the ACP AXI3 slave port within the Zynq SoC. Each transaction requires both a read phase (to retrieve the

provided command) and a write phase (to signal completion and update the highest priority runnable thread). Although this extends the time for the accelerator to complete a transaction, communication latency is reduced since data can be transferred asynchronously through the low-latency cache.

3.4.6 Comparison of accelerator structures

The GP-connected accelerator is connected to the CPU via a single GP AXI3 port. In this arrangement, the accelerator is unable to initiate transfer to either CPU or SM because it is a slave peripheral on the connected bus. Each transaction from the CPU to the accelerator is a self-contained query, insertion or removal operation to the priority queue.

The cache-coherent ACP-connected accelerator contains the same functional logic as the GP-connected accelerator, but uses an alternative interfacing approach. This accelerator includes a state machine to manage accelerator-mastered transactions to cache-coherent SM via the ACP. Each transaction has two phases: The accelerator first uses a read transaction to pull the request from SM. Once the priority queue operation has been processed, the accelerator pushes the result back to SM.

The ACP-connected accelerator can be connected to the HP port of the Zynq processor without modification. However, SW must then manage data coherency with the accelerator. This is done by either configuring SM access to bypass the CPU cache, or by executing fine-grained cache maintenance instructions on the CPU for each transaction. While both methods increase the SM access latency of both CPU and accelerator, the latter approach also requires additional CPU cycles to maintain data coherency. Due to these additional overheads, neither method is further considered in this work.

3.5 Evaluation

We consider only IPC performance in this thesis because it is a sufficient example to demonstrate fine-grained communication. The reader can find an evaluation of IRQ latency and its variance in [KD19].

We can invoke the kernel scheduler in a controlled way by performing an IPC from one thread to a thread of lower priority. IPCs of other priorities are handled by the *fastpath* and do not invoke the scheduler. When the *fastpath* is avoided and the scheduler is invoked, the priority of the sending thread has no impact on scheduler execution time.

We used the ARM performance counters to count the number of CPU execution cycles required to perform an IPC from a thread of the highest priority to threads of lower priorities. In our benchmark, the IPC receiver thread was initially blocked, waiting for IPC. The sending thread first read the state of the CPU cycle counter from the performance monitoring unit (PMU) of the CPU and issued the IPC system call. Upon receiving the system call exception, the kernel unblocked the receiver thread and issued a scheduler transaction to insert it back into the schedule. The kernel then blocked the current thread as it waited on the IPC reply and issued a second scheduler transaction to find the new highest runnable thread in the system. By benchmark design, the highest priority runnable thread was always the IPC receiver and hence it was removed and scheduled for execution. Finally, the receiver thread read the CPU cycle counter. We took the resulting execution time of the IPC as the difference between the two readings of the CPU cycle counter.

In our study, the IPC merely served as a synchronisation signal between two threads. The IPC contained no data payload. Since transfer of data occurs in linear time across all scheduler architectures under investigation, varying the payload size was not expected to yield interesting information. Threads also shared the same virtual memory translations (Section 2.4); a virtual address space switch was not performed as part of the study.

We ran the benchmark with a hot cache by using 16 cache warming iterations before collecting 250 samples. We set the HW scheduler AXI bus clock and logic clock to 100MHz

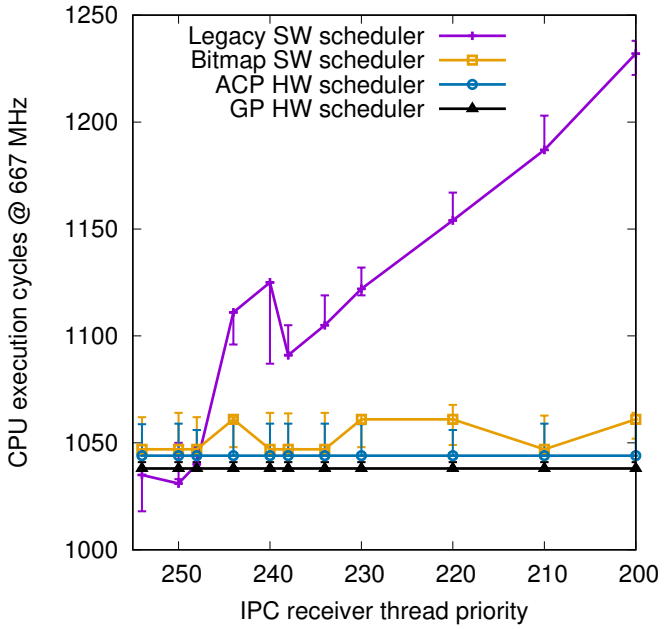


Figure 3.7: Hot cache IPC execution cycles for given receiver thread priorities.

while the CPU operated at its maximum frequency of 667MHz. The SW was compiled with arm-linux gcc 4.7.4.

We initially ran the benchmark with each of the four scheduler architectures described in Section 3.4. The *legacy* scheduler architecture was implemented entirely in SW. The *bitmap* scheduler architecture addresses the scalability issues of the legacy scheduler, but at the cost of maintaining additional data structures. The *GP* scheduler architecture featured a HW-accelerated priority queue that the CPU communicated with directly via the GP AXI port. The *ACP* scheduler architecture used cache-coherent SM to communicate between CPU and accelerator and used the *SEV* signalling mechanism for synchronisation.

We tested receiver priorities in the range of 250 to 0, however, we observed no change in trend below priority 220; we have therefore excluded some of these measurements for clarity. The median results of this benchmark are presented in Figure 3.7. Error bars indicate 1st and 3rd quartiles.

We found that the execution time of the legacy SW scheduler implementation generally increased linearly as the receiver thread priority decreased. This is because the scheduler

traverses the *ksReadyQueues* from the highest priority to the lowest priority until it finds a runnable thread. The lower the priority of the receiver, the more entries the legacy scheduler must examine before it finds the waiting receiver. This behaviour also increases the cache footprint of the scheduler: when the receiver thread is priority 0, the scheduler reads from 256 queue heads. Because head and tail pointers are interleaved, this results in 512 words (2 KB) being loaded into the cache. Another way of looking at this is that the scheduling behaviour results in the eviction of 2 KB of data from the cache in the worst case. This evicted data can be data that is frequently used by an application; system performance will then be further degraded because that data must be reloaded from high-latency main memory when the application is next scheduled.

We investigated the discontinuities in the legacy scheduler curve and found that they correlate with an unusually high number of branch mispredictions (Figure 3.8). During execution, conditional branch instructions prevent the instruction prefetcher from always maintaining a full instruction pipeline. The prefetcher does not know which branch of execution to load until the branch condition is evaluated. In these cases, the branch predictor attempts to predict the path that execution will take. If the branch predictor is correct, the CPU continues uninterrupted. If the prediction is incorrect, the CPU must flush the instruction pipeline and wait for the prefetcher to fill the pipeline with the correct instruction stream. Although this micro-architectural feature improves CPU utilisation and application performance, it can add a significant amount of variance to execution time. Our results show that branch mispredictions on the ARM Cortex-A9 processor can, with high probability in some cases, result in more CPU cycles to perform fewer iterations of a SW loop.

seL4 developers introduced a *bitmap scheduler* to address the scalability issue of the legacy scheduler implementation. This was done by extending the implementation to include a hierarchical bitmap representation of the non-empty *ksReadyQueues*. This optimisation leads to some overheads, but an O(1) lookup complexity was observed (Figure 3.7). The increased execution time of the bitmap scheduler for high-priority receiver threads reflects the additional operations required for the traversal and maintenance of the bitmap. The

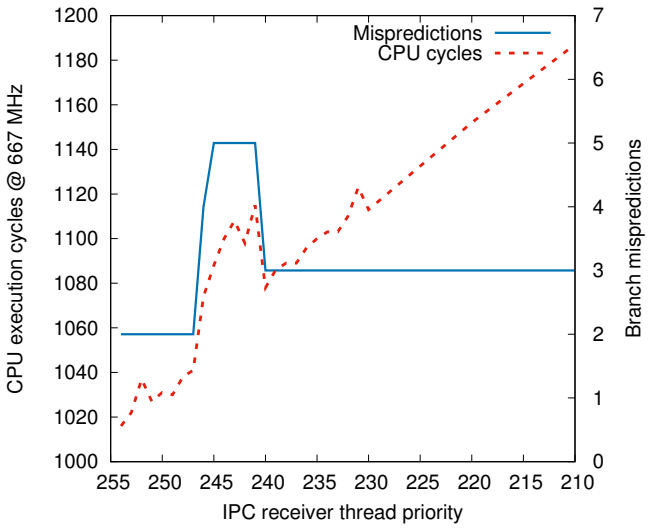


Figure 3.8: Legacy scheduler anomaly investigation.

bitmap scheduler requires 2 additional reads to locate the highest priority thread: one for each level of the bitmap hierarchy. Once the scheduler identifies the thread to be scheduled, the scheduler marks the thread as active and removes it from the scheduling queue. Because this thread is the only runnable thread in the system, the bitmap scheduler additionally marks the *ksReadyQueue* as empty in the second level and marks the priority group as empty in the first level. For the sake of abstraction, the scheduler cannot simply write 0 to these words: it must perform the correct bit operation to clear only the relevant bit at each level. The result is that the bitmap scheduler performs another 2 reads and 2 writes to update the bitmap, however, it is likely that these 4 memory accesses operate on memory in the cache rather than suffering a penalty from loading data from main memory a second time.

We also see that the directly-connected GP accelerator offered only marginal improvements over the optimised bitmap scheduler (Figure 3.7). While the entire task of priority queue management is offloaded to the accelerator, the performance gain is reduced due to the communication latency between the CPU and accelerator. We can see that the legacy SW implementation still outperforms the GP-connected accelerator when the IPC receiver priority is very high. This condition provides the best case performance for the legacy scheduler because the iterative search only needs to examine two priority levels before the

highest priority runnable thread is located.

Surprisingly, when communication is achieved using low-latency, cache-coherent SM rather than direct communication, we observed an increase in execution time relative to the GP accelerator. This is because the signalling mechanism is decoupled from the data transfer. The CPU must execute a DSB in order to stall the CPU until data has reached the L1 cache (Section 2.5.1). Only then can the CPU signal the accelerator with the *SEV* processor instruction. If the CPU executes the *SEV* instruction before this time, the accelerator may read a stale command from SM.

The decision to accelerate a short running OS kernel task scheduler on a high performance CPU was in part motivated by the challenges involved. We anticipated that accelerating the scheduler would be difficult and that it would drive us to find a truly optimal method for fine-grained interactions on an ARM-based CPU-FPGA heterogeneous platform. Our results show that the legacy SW scheduler architecture still performs better than both the GP- and ACP-connected HW scheduler for high-priority IPC receiver threads. We therefore took the best-case performance of the legacy SW implementation as a target for the execution time of our HW-accelerated task scheduler.

We constructed microbenchmarks to investigate why the direct communication of the GP approach outperforms the low-latency SM approach of the ACP-connected accelerator. We measured the median number of CPU cycles required to perform each scheduler operation (Table 3.2). The execution time was measured at the application programming interface (API) level of the scheduler, which includes the cost of validating arguments and packing commands into the appropriate format for each accelerator architecture. The overhead of reading the ARM performance counters was measured and subtracted from the results. Note that such fine-grained benchmarking avoids some performance penalty events, such as cache misses.

From the microbenchmark results we found that the GP-connected accelerator requires the least number of CPU cycles to insert a thread into the schedule. These numbers do not reflect the latency of the transaction; they represent the CPU execution time to issue

Table 3.2: Median scheduler operation cost (CPU cycles).

System	Enqueue	Dequeue highest priority	Total
Legacy (priority 255)	22	49	71
Legacy (priority 0)	22	954	976
Bitmap	30	59	89
GP	12	94	106
ACP	47	74	121
ACP (polling)	30	40	70
Hybrid	10	31	41

the command. The CPU can continue execution as soon as the transaction has reached the write buffers (Section 2.5.1), well before it is received by the accelerator.

We also found that the GP-connected accelerator requires the greatest number of CPU cycles to retrieve the highest priority thread from the schedule. This is because of an immediate dependency on the result of the transfer; we must test the validity of the returned data in case the schedule is empty. When the returned value is *NULL*, the kernel must activate the *idle* thread instead of the highest priority runnable thread. The CPU must stall until this branch in execution is determined.

The ACP results reported a lower execution time for retrieving the highest priority thread than the GP-connected accelerator. This is because the CPU can retrieve the result directly from the low-latency cache instead of waiting for a response from HW. However, the synchronisation time adds significant overhead for insertions. The total time to perform both an insertion and removal of the highest priority runnable thread is thus higher for the ACP scheduler than for the GP scheduler.

To quantify the costs of data synchronisation and signalling, we modified the ACP-connected scheduler to operate in polling mode. In this mode, the accelerator continuously polls SM for a new command. We found that data synchronisation was still required when inserting a thread into the schedule. This is because there was not enough time between thread insertion and thread removal to allow the insertion transaction to complete. The barrier forces CPU buffers to write through to the cache sooner, which reduces the observation time

Table 3.3: Median IPC execution time for priority 254 receiver.

System	Median IPC execution CPU cycles	Speedup
Legacy	1029	2.6%
Bitmap	1057	0.0%
GP	1038	1.8%
ACP	1044	1.2%
ACP (polling)	994	6.0%
Hybrid	999	5.5%

(and therefore completion time) of the accelerator. On the other hand, thread selection is performed as the last scheduler transaction before returning control to the chosen thread. The time required to restore the thread context and leave the kernel is large enough to avoid this memory barrier. The microbenchmark results of the ACP scheduler in polling mode are included in Table 3.2. The polled ACP scheduler requires the least number of CPU cycles to retrieve a thread across all systems that have been discussed so far.

We investigated a hybrid implementation to take advantage of the best performing communication channel for each operation. In this implementation, the scheduler uses the GP port for all priority queue commands. After processing any command, the accelerator provides the highest priority runnable thread via ACP in low-latency, cache-coherent SM.

We repeated the IPC benchmarks that were detailed at the beginning of this section for both the polling-mode ACP-connected accelerator and the hybrid accelerator. Our results focus on comparing the execution time of each architecture with the best case performance of the legacy SW implementation (receiver thread priority 254). The distribution of collected samples is presented as box and violin plots in Figure 3.9 while numerical results are presented in Table 3.3. The violin plot shows the probability density of the collected samples: larger plot widths correspond to values at which we observed more samples.

The results show that the polling-mode ACP-connected accelerator and the hybrid accelerator perform better than both SW approaches in all cases. The hybrid approach offers a 2.9% reduction in median execution time when compared with the best-case performance of the legacy SW scheduler. We observed a 5.5% reduction compared with the bitmap scheduler.

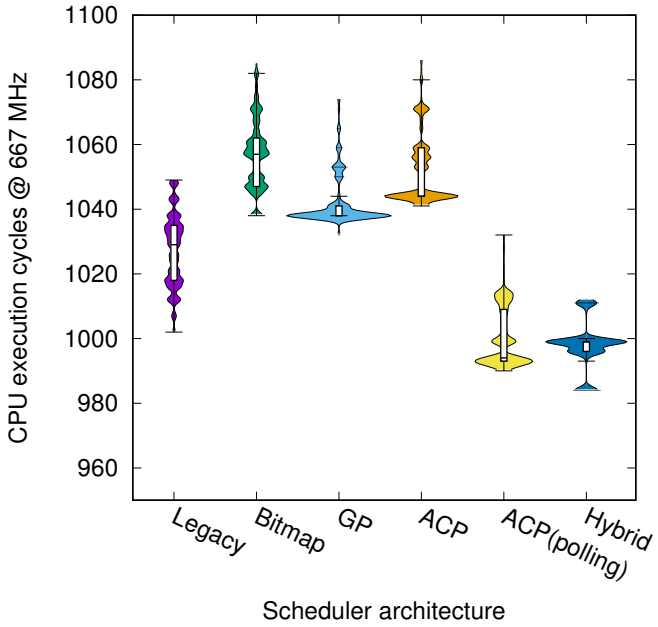


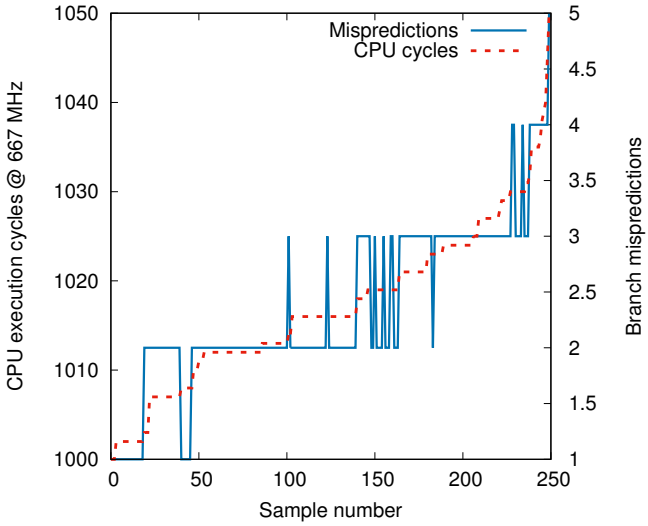
Figure 3.9: Hot cache execution cycles with probability density for IPC from thread priority 255 to 254 for the scheduler architectures studied.

A reduction in scheduler execution time allows more tasks to be scheduled in a given period of time. The time that would otherwise be consumed by kernel execution can now be used by an application to complete sooner, thereby allowing other tasks to begin sooner.

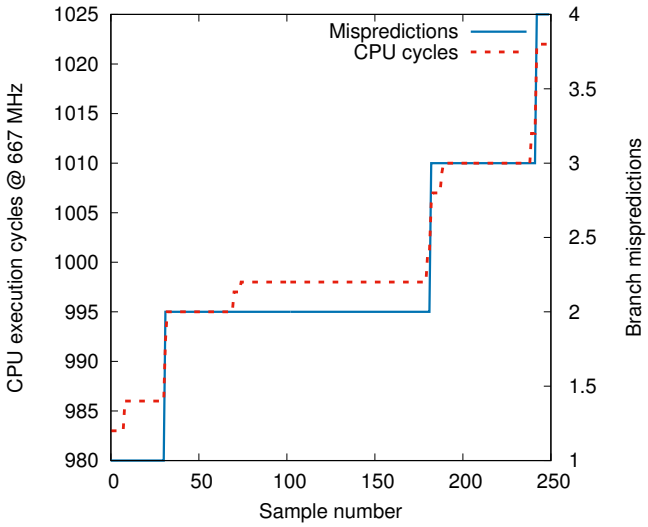
The results in Figure 3.9 also provide information about the execution time variance. The schedulers that use HW acceleration showed well-defined modes. The ARM event counters [ARM05] showed that these modes correlate with branch mispredictions (Figure 3.10). This pattern of mispredictions was also observed in the pure SW implementations. However, noise from other sources of non-deterministic execution masked the effect.

HW acceleration improved the execution time variance in all cases. The hybrid approach, in particular, showed improvements of 58% when compared to the legacy SW implementation and 56% when compared with the bitmap optimisation (Table 3.4).

All other things being equal, the reduction in variance allows more tasks to be scheduled *safely* in a given period of time. Since the execution time is more deterministic, we can reduce the compute power that is reserved for ensuring that a critical real-time task will



(a) Legacy SW scheduler.



(b) Hybrid scheduler.

Figure 3.10: Branch mispredictions correlated with CPU execution cycles. Samples sorted by execution time.

Table 3.4: IPC execution time variance for priority 254 receiver.

System	Variance	Improvement
Legacy	120	0%
Bitmap	120	0%
GP	65	46%
ACP	110	8%
ACP (polling)	80	33%
Hybrid	51	58%

complete on time. One can then either schedule more tasks on the processor, or reduce the size of the compute resource to conserve energy and manufacturing costs.

Although we have reduced the variance of the system, the branch predictor remains a dominant source thereof. The branch predictor is an important micro-architectural feature for enhanced performance. Although it can be disabled to further reduce variance, this causes a significant reduction in CPU utilisation and an increase in program execution time.

3.6 Chapter summary

We have evaluated the potential for tightly-coupled CPU-FPGA systems to improve the CPU execution time and jitter of frequent, short-running operations. The target application, the seL4 kernel SW task scheduler, proved difficult to accelerate and required careful selection of HW-SW communication strategies.

We evaluated communication via MMIO and also cache-coherent SM. No single strategy on its own was able to provide a performance improvement over the best case execution time of the legacy SW. However, a carefully selected combination of these strategies was able to achieve a 5.5% improvement in CPU execution time.

Although the improvement in execution time is small, our results show that the use of FPGA-based accelerators in tightly-coupled systems need not be limited to coarse-grained acceleration. They can also improve the execution time of very short-running tasks, such as appending to and removing from a linked list.

In our study, execution time was reduced when all communication from the CPU to the FPGA was made through MMIO. With that method, write buffering allowed the CPU to quickly continue executing meaningful work. Similarly, execution time was reduced when all communication from the FPGA to the CPU was made through SM. This allowed the CPU to perform more meaningful work by preventing high-latency reads from stalling the

CPU.

In the next chapter, I consider the partitioning of short-running tasks across a CPU and an FPGA for cooperative processing. The collection of communication methods identified in this chapter are used to minimise both communication latency and CPU idle time. These methods improve both overall task completion time and allow smaller workloads to benefit from HW acceleration.

Chapter 4

Cooperative processing of short-running tasks

Due to the cost of repeated data movement between CPU and FPGA, the use of FPGA-based accelerators has traditionally been limited to offloading long-running tasks from the CPU to programmable logic. These tasks are typically processed entirely by the accelerator. While the workload is being processed, SW will either wait until the task is complete or schedule other tasks for processing.

Although modern heterogeneous platforms (Section 2.6), reduce the costs of CPU-FPGA data transfers, the traditional offload model is cemented as the popular choice. For these systems to become truly heterogeneous, the utilisation of all computational resources should be optimised. In particular, the CPU and FPGA should cooperate by dividing the workload between them so as to maximise system throughput.

For long-running tasks, the cost of communication is small relative to processing time. Therefore, communication overhead can be ignored and workloads can be partitioned proportionally to the throughput that each compute resource provides.

In contrast, the cost of communication for short-running tasks is relatively large: commu-

nication costs must be considered when partitioning such tasks across compute resources. Over a fixed period of time, the CPU cycles required for communication reduces the amount of work that can be processed by the CPU. Similarly, communication latencies reduce the amount of work that can be processed by an accelerator. In the extreme case, the communication overhead may outweigh the acceleration provided by HW.

In this chapter, I present a model that predicts if, and by how much, short workloads should be partitioned between HW and SW for cooperative processing. As well as the throughput of each resource, the model considers both the CPU overhead and the latency of communication when partitioning the workload. The model allows engineers to choose a workload partitioning that minimises task completion time.

I then extend the contributions of Chapter 3 by providing a detailed evaluation of the CPU overhead and latency of short transfers between CPU and FPGA on the tightly-coupled Zynq-7020 SoC. Such transfers are essential to efficiently synchronise between cooperating HW and SW tasks and provide key parameters for the model. While these metrics are specific to the Zynq-7020 SoC, similar techniques can be applied to other platforms to find communication metrics.

Finally, I demonstrate how the derived model and the communication metrics can be used to choose the optimum workload partitioning for a stream-based integer accumulator task. The model determines when cooperative processing becomes beneficial within 8% of the optimum, chooses a partitioning that leads to a task completion time within 2% of the optimum, and predicts task completion time with 12% mean relative error (MRE).

4.1 Contributions

The contributions of this work are:

- We derive a model that determines the optimal partitioning of work between a CPU and a connected FPGA for any cooperative workload.

4. Cooperative processing of short-running tasks

- We predict for which cooperative workloads the execution time of a task will be reduced if some portion of the work is processed in HW.
- We measure the communication overheads of short transfers between the CPU and FPGA on the tightly-coupled Zynq-7020 SoC.
- Using a stream-based integer accumulator task, we demonstrate that the model can be used to predict optimum workload partitioning between CPU and FPGA.

4.2 Publications

Parts of this work were accepted via peer review into the International Conference on Field-Programmable Technology (FPT) for presentation in 2018. While most of that work is my own, I credit my coauthor and supervisor, Oliver Diessel, for his overall guidance and editorial support.

[KD18] A. Kroh and O. Diessel. A short-transfer model for tightly-coupled CPU-FPGA platforms. In *2018 International Conference on Field-Programmable Technology (FPT)*, pages 366–369, Dec 2018.

4.3 Prior work

Offline HW/SW functional partitioning of a task has been studied extensively in prior work [Tei12, THM15, LVL03]. The goal of that work is fast and automated design space exploration for minimising execution time, energy consumption and HW costs. Sequential segments of code are identified and grouped into nodes termed basic blocks. Those blocks are connected to form a graph that reflects the flow of program execution. The challenge is in formulating an algorithm that chooses which of those blocks should be executed on which of the available resources [KPPK11, WWLS13, WSC10, CLL+96].

4. Cooperative processing of short-running tasks

Those algorithms optimise partitioning by considering the benefit in execution time, the FPGA area required and the cost of migrating control and data. However, few studies have involved physical HW that impose realistic migration costs [VRKP14, LBK+16].

Offline HW/SW partitioning is difficult to optimise when program execution through the graph is determined at run-time. For example, the number of iterations of a program loop may be determined by the provided dataset. If the iteration count is unknown, the benefit of migration is difficult to balance with its cost. For a range of tasks, Vaz et. al. showed that the iteration count could only be determined offline in 11% of cases, while 46% could be determined online [VRKP14]. The authors propose that partitioning decisions should be made online once the number of iterations is known.

While the prior work mentioned above could extend our research, our focus is on partitioning the dataset itself across compute resources rather than the functional partitioning of a task.

In this work, both CPU and FPGA perform a common function to process the workload. We partition the workload online with loop iteration bounds providing an indicator as to the execution time on each resource. Like the prior work mentioned above, we must consider the migration costs when determining how much work to offload. These migration costs account for synchronisation overheads between CPU and FPGA using short transfers.

While many have evaluated communication performance on various platforms, the evaluations are throughput-oriented and focus on traditional large data transfers between CPU and FPGA. To our knowledge, no research has been conducted into how these throughput models can by applied to an application that uses frequent small transfers.

The achievable PCIe bandwidth for large transfers between system memory and the FPGA was measured to be 3 GB/s, but falls to ∼65 MB/s for 1 KB transfers [JK13]. In a cooperative system with frequent short transfers, we expect transfers as small as one byte.

The throughput of MMIO communication between CPU and FPGA for payload size in the 2 B to 2 MB range has been evaluated on both the Intel Cyclone and Xilinx Zynq

CPU-FPGA devices. That work found that a peak bandwidth of 30 MB/s is achieved for the Cyclone device for packet sizes larger than 8 B [MSFRA15]. On the Zynq device, 90 MB/s can be achieved for reads and 55 MB/s for writes when the packet size is larger than 128 B [CFMRAF17]. Although small transfer sizes were considered, only throughput is evaluated and the range of small transfer sizes evaluated is limit.

Other work has explored the performance of both MMIO and SM communication in order to assist an engineer in the selection of the optimal channel. On Zynq, it is shown that MMIO is best for transfers in the range 16 B to 64 K [SSS15], cache-coherent SM is best for sizes up to 64 KB [SSS15, SWWB13], and uncached SM communication is best for sizes up to 2 MB [SWWB13]. A similar study on Cyclone found that the choice depends on the direction of data transfer [MRAF18], with writes being served better via MMIO and reads from SM.

4.4 Partitioning model

The focus of this work is cooperative computation, in which two compute resources perform a common function on a subset of the provided data. Once all compute resources have processed their individual workloads, a nominated compute resource aggregates the partial results and returns the final result to the application (Figure. 4.1).

Workloads best suited for this model can be flexibly divided into independent datasets that require infrequent synchronisation between them. Examples include image processing and matrix multiplication, where the dataset is typically partitioned into blocks for processing [JSNV13]. Our accumulator study presents a workload that allows fine-grained partitioning. Each integer to be accumulated can be considered independently.

The CPU overheads of migrating a task from SW to HW are the CPU execution cycles required to both initiate processing by the accelerator and to collect the result. The initiation overhead may include preparing memory for FPGA access, the transfer of operating parameters, and a command to the accelerator to indicate that it can begin

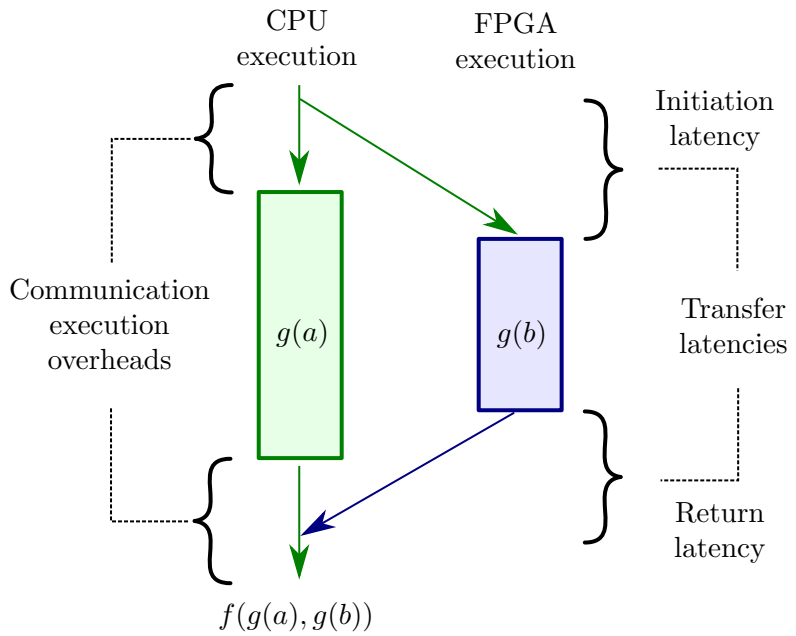


Figure 4.1: Cooperative system execution and overheads.

processing. On the return path, the overhead may include preparing memory for CPU access, retrieving the partial result from the accelerator or memory, and aggregating the partial results of HW and SW processing.

Latency overheads are the combined latency of accelerator initiation and of returning the partial result to the CPU. The initiation latency is the time between when the CPU executes the initiation procedure and when the accelerator can begin processing. Although this includes the latency of the first dataset reaching the accelerator, the latency of further data access is considered in the throughput that the accelerator and its connected data bus provide. The return latency includes the transfer of any outstanding data when the accelerator completes processing and a signal to the CPU to indicate that the result is available.

The execution time T of a cooperative task is determined by the longest completion time, including transfer latencies and overhead, across the accelerator (T_a) and CPU (T_c). If too much work is given to the accelerator, the CPU will become idle as it waits for the accelerator to complete. If too little work is sent to the accelerator, an opportunity for

parallel execution is lost. Therefore, the workload must be carefully partitioned to minimise the completion time, which occurs when $T_a = T_c$.

When the workload, N, is large, the overheads of accelerator initiation and returning the result can be ignored as they are small relative to the computation time. The completion time of the accelerator and CPU partitions can then be calculated using (4.1a) and (4.1b) respectively, where α^* is the fraction of the workload that should be processed by the accelerator. By equating (4.1a) and (4.1b), we see that α^* partitions the workload proportionally to the throughput provided by the accelerator (X_a) and the CPU (X_c) (4.2).

$$T_a = \frac{\alpha^* N}{X_a} \qquad (4.1a) \qquad\qquad T_c = \frac{(1-\alpha^*)\,N}{X_c} \qquad (4.1b)$$

$$\frac{\alpha^* N}{X_a} = \frac{(1-\alpha^*)\,N}{X_c} \qquad \text{sub. (4.1a) and (4.1b) into } T_a = T_c$$

$$\alpha^* X_c = (1-\alpha^*)\,X_a$$

$$\alpha^* (X_c + X_a) = X_a$$

$$\alpha^* = \frac{X_a}{X_a + X_c} \qquad (4.2)$$

For small workloads, cooperative computation requires careful attention to data transfer costs in terms of both transfer latencies (D_a) and CPU overheads (O_c) (Figure. 4.1, Eq. (4.3a) and Eq. (4.3b)). Transfer latencies on both the initiation and return paths reduce the amount of work that the FPGA can complete. Programmable logic must wait for the processing command to arrive and ensure that the result is available to the CPU as soon as it is needed.

$$T'_a = \frac{\alpha N}{X_a} + D_a \qquad (4.3a) \qquad\qquad T'_c = \frac{(1-\alpha)\,N}{X_c} + O_c \qquad (4.3b)$$

The transfer of the processing command is performed by the CPU. The CPU execution cycles required to perform this transfer represent an opportunity cost to the CPU as these cycles could be used to process the workload. By considering both CPU overheads and transfer latencies, the workload can be partitioned by α (4.4) such that the CPU and FPGA complete the processing of their respective parts at the same time.

$$\frac{\alpha N}{X_a} + D_a = \frac{(1-\alpha)\,N}{X_c} + O_c \qquad \text{sub. (4.3a) and (4.3b) into } T_a = T_c$$

$$\frac{\alpha N}{X_a} - \frac{(1-\alpha)\,N}{X_c} = O_c - D_a$$

$$\alpha X_c - (1-\alpha)\,X_a = \frac{X_a X_c\,(O_c - D_a)}{N}$$

$$\alpha\,(X_c + X_a) = X_a\left[\frac{X_c\,(O_c - D_a)}{N} + 1\right]$$

$$\alpha = \frac{X_a}{X_c + X_a}\left[1 - \frac{X_c\,(D_a - O_c)}{N}\right]$$

$$\alpha = \alpha^*\left[1 - \frac{X_c\,(D_a - O_c)}{N}\right] \qquad \text{sub. (4.2)} \qquad (4.4)$$

Workload partitioning improves task completion time only if the time required for communication is masked by a reduction in completion time. To determine the workload, N_L, for which we benefit from using the programmable logic, we must ensure two conditions are met: First, the time to initiate the accelerator to perform some subset αN_L of the workload, to compute $(1-\alpha)N_L$ work on the CPU and to receive the results is less than or equal to the time required to process N_L work on the CPU alone (4.5), where O_0 is the proportion of O_c associated with calling and processing the function in SW with $N = 0$ work. Second, the time to process αN_L on the accelerator should also be less than or equal to the time to process N_L work on the CPU alone (4.6). We find N_L using Eq 4.8 by solving the two inequalities (4.5 and 4.6) simultaneously.

$$\frac{N_L}{X_c} + O_0 \geq \frac{(1-\alpha)\,N_L}{X_c} + O_c \tag{4.5}$$

$$\frac{N_L}{X_c} + O_0 \geq \frac{\alpha N_L}{X_a} + D_a \tag{4.6}$$

$$\frac{N_L}{X_c} - \frac{(1-\alpha)\,N_L}{X_c} \geq O_c - O_0 \qquad \text{rearranging (4.5)}$$

$$N_L - N_L + \alpha N_L \geq X_c\,(O_c - O_0)$$

$$\alpha N_L \geq X_c\,(O_c - O_0) \tag{4.7}$$

$$\frac{N_L}{X_c} + O_0 \geq \frac{X_c\,(O_c - O_0)}{X_a} + D_a \qquad \text{sub. (4.7) into (4.6)}$$

$$N_L \geq X_c \left[\frac{X_c\,(O_c - O_0)}{X_a} + D_a - O_0 \right] \tag{4.8}$$

4.5 Evaluation of communication overheads

To evaluate our model, we must first find values for the CPU overhead (O_c) and latency (D_a) of the available methods for initiating an accelerator and retrieving the result on our target platform. Knowledge of these overheads also assists in choosing the most appropriate communication primitives for our accelerator.

We used the Avnet Zedboard for our study, which features a Zynq-7020 SoC (Section 2.7.3). The Zynq comprises two ARM Cortex-A9 CPU cores with tightly-coupled programmable logic. Data transfer can be achieved by sharing memory in either RAM or with the low-latency L2 cache of the CPU. In the latter case, memory access by the FPGA is coherent with that of CPU (Figure 4.2). Therefore, the CPU overhead of preparing memory for shared access is avoided.

The Zynq also supports CPU mastered direct communication between CPU and FPGA via

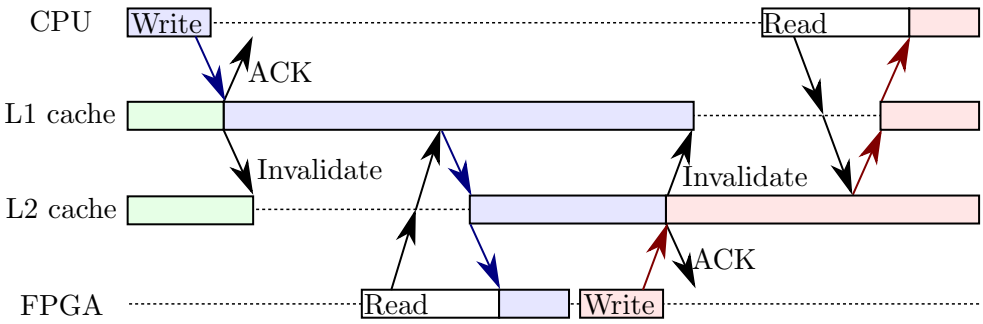
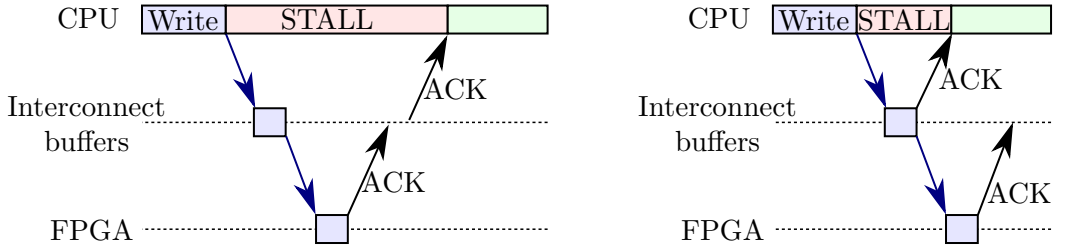


Figure 4.2: Cache-coherent data flow on Zynq-7000 series SoC.



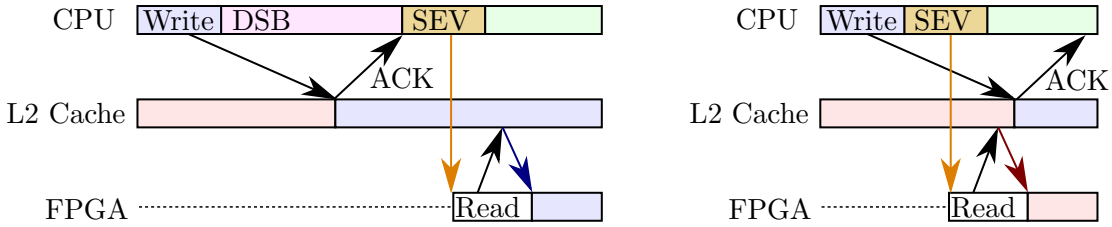
(a) MMIO write transaction with SO memory attribute.

(b) MMIO write transaction with DE memory attribute.

Figure 4.3: MMIO write transactions with SO and DE memory attributes.

MMIO. Write transfers propagate from the CPU to the FPGA through interconnects that each provide internal buffering. When access is configured with the strongly-ordered (SO) memory attribute, the CPU must wait for each write to be acknowledged by the target before another transaction can be issued (Figure 4.3a). On the other hand, the device (DE) attribute allows the CPU to continue as soon as the write is buffered (Figure 4.3b). While the DE attribute reduces the CPU overhead of writes, buffers throughout the interconnect may merge writes to the same address such that some writes never reach their target (Section 2.5.1). SO and DE reads follow the same principles as writes, except that data is returned with the acknowledgement of completion. Because reads modify CPU state (the register file), the number of outstanding transactions before the CPU must stall is reduced.

While both SM and MMIO can be used to transfer data, MMIO additionally provides a signal to the accelerator that data has been transferred. An approach to signalling when using cache-coherent SM is provided by the send event (SEV) instruction. ARM suggests



(a) SEV signalling with data synchronisation.

(b) SEV signalling only.

Figure 4.4: Cache-coherent SM writes with signalling on Zynq-7000 series SoC.

that this mechanism would be useful for synchronising access to memory that is shared between CPU and accelerator [ARM05]. When data has been written to cache-coherent SM, the data synchronisation barrier (DSB) instruction is used to stall the CPU until the data can be observed in the cache by the accelerator. The CPU then executes the SEV instruction to toggle the logical state of the *EVENT_EVENTO* signal in programmable logic. This notifies the accelerator that data has been updated (Figure 4.4a). Although data synchronisation extends the execution time of the transfer, it prevents the accelerator from reading stale data from the cache (Figure 4.4b).

In the subsections that follow, the CPU overhead and latency of short transfers using MMIO and SM are evaluated to acquire the metrics that are used by our model. In the case of MMIO, both SO and DE memory attributes are evaluated. In the case of SM, we also evaluate the overhead of signalling using the SEV instruction. The latency of bulk data transfers is application specific and will be discussed in our HW accumulator evaluation in Section 4.6.

For all of our experiments, the CPU, programmable logic and DDR RAM were configured to operate at 667 MHz, 214 MHz and 533 MHz respectively. The theoretical maximum bandwidths of the ACP, L2 cache, DDR RAM and GP port are 1632 MB/s, 5086 MB/s, 2132 MB/s and 816 MB/s respectively at their configured operating frequencies [Xil14]. All experiments were run as bare metal applications.

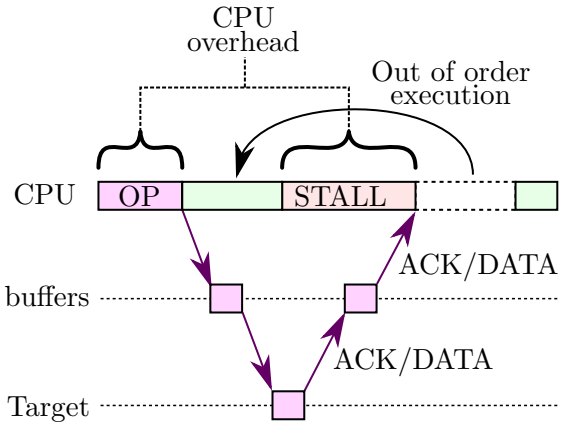


Figure 4.5: CPU overhead and out-of-order execution.

4.5.1 CPU overhead

The CPU overhead of a short transfer is the CPU execution time required to perform that transfer to or from a target memory or accelerator. The execution time depends on the operation, the target, the access attributes and whether or not subsequent instructions can be executed out-of-order. Out-of-order execution reduces the CPU overhead that it attributed to CPU pipeline stalls (Figure 4.5).

We evaluated the CPU overhead of reads and writes to various targets. In our experiments, MMIO communication to programmable logic was issued to an AXI memory controller that we connected directly to the GP port at the CPU-FPGA boundary of the Zynq. This controller was designed to respond immediately to all requests and thereby eliminated latency due to soft interconnects and peripherals.

Communication via cache-coherent SM is via the ACP on the Zynq. An ACP read transaction from programmable logic to SM can be served from any level of the memory hierarchy, including the private L1 cache of either CPU core. On the other hand, an ACP write transaction from programmable logic is always issued to the L2 cache. In this case any corresponding cache lines present in the private L1 cache of each CPU core are invalidated. When the CPU reads data back from SM, read requests are served by the L2 cache of the CPU. Therefore, the CPU overheads of cache-coherent SM communication

Table 4.1: Zynq-7020 CPU overheads, measured in CPU cycles, for short transfers between CPU and various targets.

Target	Words = 1	2	3	4	5	6	7	8
In-order execution								
L2 Cache read	28	41	54	67	80	93	106	119
FPGA MMIO read (DE)	76	86	103	119	145	164	178	188
FPGA MMIO read (SO)	76	146	205	278	343	413	478	542
L1 Cache write	7	7	8	9	10	11	12	13
L1 Cache write; DSB; SEV	24	24	25	26	27	28	29	30
FPGA MMIO write (DE)	14	17	20	23	26	29	79	86
FPGA MMIO write (SO)	85	164	241	317	385	476	565	626
Out-of-order execution								
L2 Cache read	12	25	38	51	64	77	90	103
FPGA MMIO read (DE)	60	70	87	103	129	148	158	172
FPGA MMIO read (SO)	60	130	189	262	327	397	461	526
L1 Cache write	1	1	2	2	3	3	4	5
L1 Cache write; DSB; SEV	24	24	25	26	27	28	29	30
FPGA MMIO write (DE)	1	1	4	7	10	13	63	70
FPGA MMIO write (SO)	69	145	225	298	369	468	549	610

were measured as the CPU execution time required to write to the L1 cache for writes and read from the L2 cache for reads.

We used the CPU cycle counter of the ARM performance monitoring unit (PMU) to measure the CPU overheads for reads and writes of various sizes to and from various targets. Both SO and DE memory access attributes were explored when accessing the FPGA using MMIO. We measured a pessimistic execution time by flushing the CPU write buffers and execution pipeline before and after the transfer instruction(s) were issued. To consider the impact of out-of-order execution, we also measured a best-case execution time by adding 12 instructions that could be executed out-of-order after the transfer instruction(s). In both cases, the overhead of reading the CPU cycle counter and executing the pad instructions (when present) were subtracted from the measured result. The median CPU execution cycles of 100 measurements for both reads and writes were recorded. While key results are summarised in Table 4.1, a complete set of results can be found in Appendix A.

Our results show that CPU overheads for FPGA writes of up to 6 words in size were similar

to those obtained for L1 cache writes when out-of-order execution is possible and the DE attribute is used. This is because the DE attribute allows the CPU to continue to execute after issuing the transaction to the interconnect. While the L1 cache operates at the same frequency of the CPU, the interconnect to the FPGA operates at 1/3 of that frequency. Therefore, once CPU write buffers are full, each additional write requires 3 CPU cycles to execute. Once interconnect buffers are full, additional writes must stall the CPU until a previous write transaction has been processed.

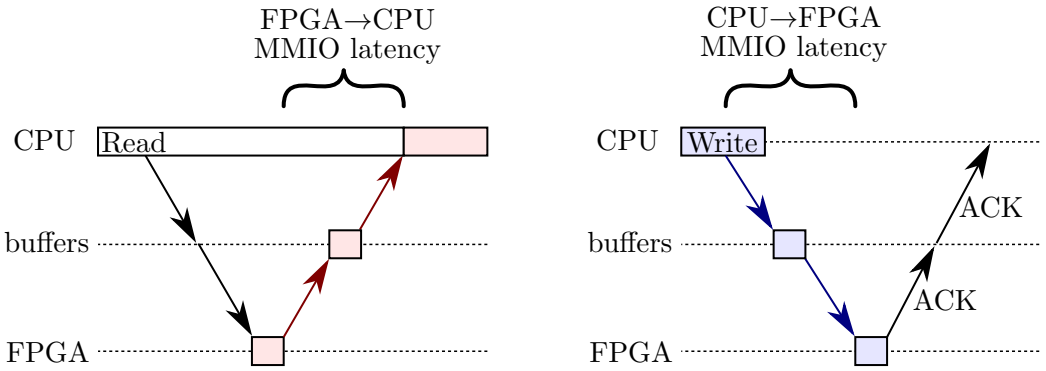
When using the SEV signalling mechanism to notify the accelerator of L1 cache writes, an additional 23 CPU cycles are required when compared to the L1 cache write itself. Such signalling can alternatively be achieved by writing the last word of the transfer directly to the FPGA. In that case, at most 14 CPU cycles are required when the DE attribute is used. Therefore, this method of signalling is only beneficial if it significantly reduces the latency of transfers from CPU to FPGA.

For all memory attributes and execution orderings, the CPU overhead of an FPGA read is much higher than the overhead of an L2 cache read. This is due to CPU pipeline stalls as it waits for the read transaction to complete. The duration of that stall is determined by the latency of the transaction, which is higher for the FPGA than the L2 cache.

4.5.2 Latency

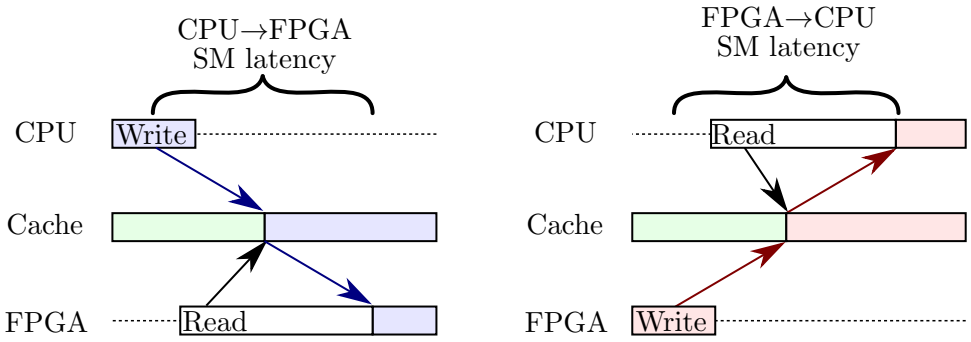
Latency is the time between when a transfer begins and when data arrives at its target. The latency represents time when the accelerator is not able to process data and must be considered when balancing the work between HW and SW. The latency of communication between CPU and accelerator is part of the D_a parameter in our model. The remaining part is the latency of the initial and final bulk data transfers from and to SM (if required).

Prior work has measured only the round-trip latency of a transfer [MSFRA15] [MRAF18] [CFMRAF17] [SSS15] [SWWB13]. Round-trip latency is convenient because a common reference of time can be used to record both the start and end time of the experiment.



(a) FPGA to CPU MMIO latency. (b) CPU to FPGA MMIO latency.

Figure 4.6: MMIO latencies.



(a) CPU to FPGA SM latency. (b) FPGA to CPU SM latency.

Figure 4.7: Cache-coherent SM latencies.

However, our model requires that we exclude the latency of read requests (Figure 4.6a) and write acknowledgements (Figure 4.6b) to measure the latency of the transfer of the data itself.

For cache-coherent SM communication, data transfer propagates through the cache. Therefore, we measure the latency as the time between when the writer transfers data to the cache and when that data is observed in the cache by the reader (Figure 4.7).

To provide a common reference of time, we used the low-latency SEV signalling mechanism provided by the Zynq. We executed the SEV instruction at key points in the instruction stream to observe when neighbouring instructions were executed by the CPU. We assumed that this signal has no latency since it does not propagate through interconnects within

the CPU or programmable logic. We used Vivado's integrated logic analyser (ILA) soft IP core to measure the elapsed FPGA clock cycles between events in programmable logic and the CPU. For consistency, we present our results in CPU cycles.

For MMIO communication from CPU to FPGA, we executed the SEV instruction immediately before a write instruction. We then measured the time from when the SEV instruction was observed until the data arrived at the CPU-FPGA boundary.

For MMIO communication from FPGA to CPU, we executed the SEV instruction immediately after the read instruction. The SEV instruction enforces a serialising barrier in so far as any previous instruction that modifies CPU state completes before the SEV is executed. In our case, the CPU state is the target CPU register of the read. Therefore, these instructions will execute in program order. We measured the latency as the time from when the data arrived at the CPU-FPGA boundary until the SEV instruction was observed.

The latency of SM communication was measured by first initialising a pre-determined word in memory to a known value. The sender was then configured to change the value of that word and the receiver was configured to continuously read that word until it observed that the value had changed. When the FPGA is the sender, we programmed the CPU to execute the SEV instruction when it observed a change in memory content. The latency was then measured as the time from when the write transaction arrived at the CPU-FPGA boundary until the CPU executed the SEV instruction. When the FPGA was the receiver, the CPU executed the SEV instruction immediately before writing to SM. The latency was then measured as the time from when the SEV instruction was observed until the updated value appeared at the CPU-FPGA boundary.

The results of our experiments (Table 4.2) show that the latency of short transfers from the CPU to the FPGA are lowest when communication is direct via MMIO. In Section 4.5.1, we found that writes from the CPU to the FPGA have a similar overhead to L1 cache writes when up to 6 words are transferred. Depending on the throughputs provided by each resource, the CPU overhead of MMIO communication may outweigh the increased

Table 4.2: Zynq-7020 communication latency between CPU and programmable logic.

Direction	Method	Latency (CPU cycles)
CPU→FPGA	MMIO	32
	SM	42
FPGA→CPU	MMIO	36
	SM	36

latency when more than 6 words must be transferred. Therefore, we conclude that MMIO should be used for all transfers less than 7 words in size, with larger transfers dependent on the opportunity cost of CPU overhead and transfer latency to each resource.

Regarding transfers from FPGA to CPU, we found that transfer latency was the same for both MMIO and SM communication. Since the overhead of L2 cache reads is significantly higher than FPGA reads (Section 4.5.1), we conclude that SM should be used for all communication from FPGA to CPU.

4.6 Hardware accumulator evaluation

The objectives for this chapter were to create a model for low-latency communication on tightly-coupled CPU-FPGA platforms and to demonstrate that it supports cooperative processing of short-running tasks that are partitioned across CPU and FPGA. As previously foreshadowed, we chose an accumulator task for this demonstration because it is easily partitioned and cooperatively executed on both CPU and FPGA. Although this application is trivial, it allows us to focus on workload partitioning and communication overhead, rather than on the underlying data processing algorithm. The methods used in this study can be applied to a range of other applications.

The accumulator has a known workload (the number of values to accumulate). As previously explained, the challenge for this application is to partition the workload such that the CPU and FPGA complete their share of the work at the same time. With this application we show that our model determines the optimum partitioning ratio to minimise completion time. We also show that N_L (the minimum workload at which we benefit from HW

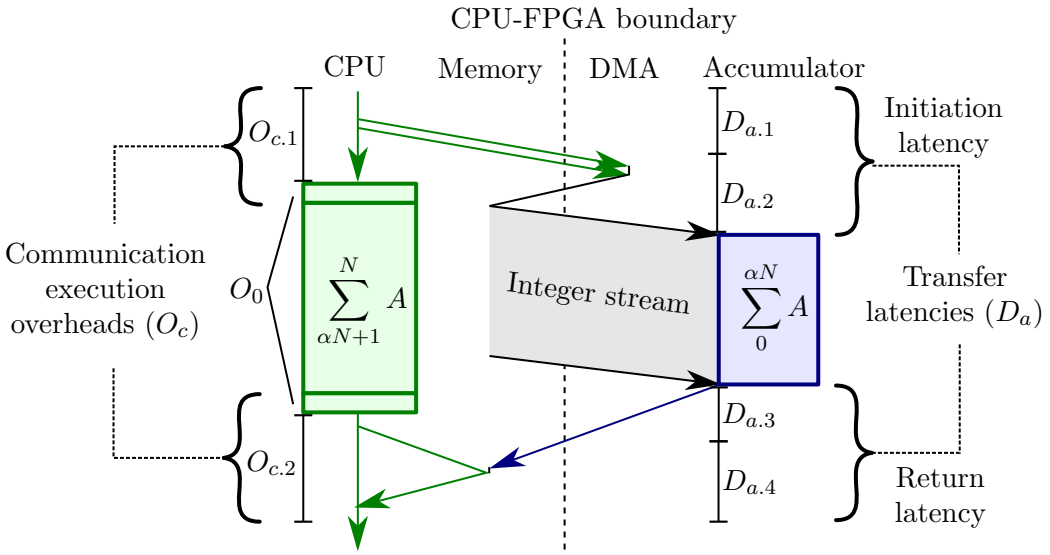


Figure 4.8: Accumulator connection and communication.

acceleration) is accurately predicted by our model.

We connected our accumulator to a DMA controller implemented in programmable logic. The DMA controller provided a stream of integers from SM for processing (Figure 4.8). The DMA engine was configured using two MMIO writes from the CPU to the DMA engine via the GP port. These writes provided the address of the first integer, the number of bytes to transfer and instructed the DMA engine to begin the transfer.

The DMA engine provided independent channels for up- and down-stream transfers. We used the second channel to transfer the result of the computation back to SM once the last integer had been accumulated. We assumed that this needed only to be programmed once to choose a fixed location in SM for the result of all accumulator operations. For this reason, the overhead of programming the second channel was excluded from our experiments.

The DMA engine was connected to the ACP to avoid the overheads of cache-maintenance. Our IP was capable of processing two 32-bit words concurrently in a single cycle to match the 64-bit data bus of the ACP. The system throughput was thus limited by the bandwidth provided by the ACP.

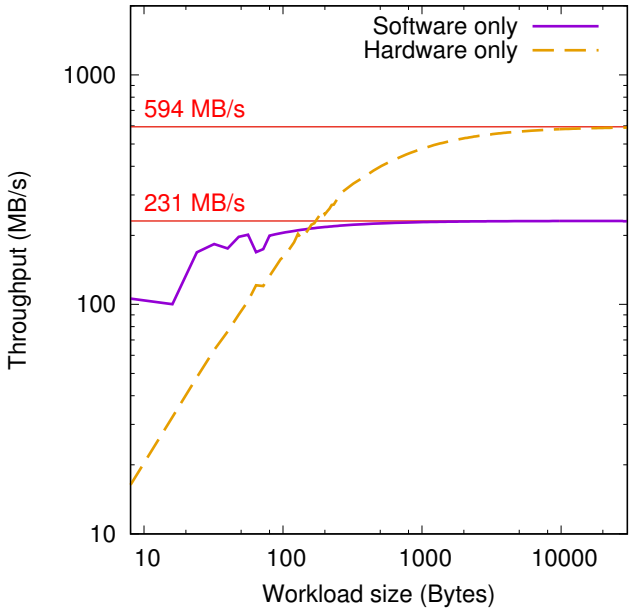


Figure 4.9: Accumulator throughput for software and hardware.

We measured the final component of CPU overhead, O_0 by using the CPU cycle counter to measure the cost of calling the accumulator function with 0 integers to accumulate.

The final component of latency was found by using the ILA to measure the time between the DMA controller transaction arriving at the CPU boundary and the first integer pair arriving at the accelerator. Alternatively, this could be found from the DMA and interconnect IP core specifications.

Finally, the throughput of the compute resources (X_a and X_c) was measured using the CPU cycle counter when calling the accumulator function with very large workloads (Figure 4.9) – the communication overheads can be ignored in this case. The complete set of parameters, including α and N_L, are shown in Table 4.3. For our accumulator application, N_L must be a multiple of 8 bytes to match the 64-bit wide ACP bus.

We measured the completion time of the accumulator when using SW- and HW-only, as well as a cooperative system in which the partitioning was chosen by our model (Figure. 4.10). Included in our results is the expected execution time as predicted by our model. We also include the empirical optimum result for each workload, which was found experimentally

Table 4.3: Accumulator model parameters given a 667 MHz CPU clock frequency and 214 MHz FPGA clock frequency.

Symbol	Value	Source
X_a	594 MB/s	Application dependent (Figure 4.9)
X_c	231 MB/s	Application dependent (Figure 4.9)
O_0	36 ns	Application dependent
$O_{c.1}$	1.5 ns	Table 4.1: FPGA (DE) write of 2 words (1 cycle)
$O_{c.2}$	42 ns	Table 4.1: L2 cache read of 1 word (28 cycles)
O_c	80 ns	$O_{c.1} + O_{c.2} + O_0$
$D_{a.1}$	48 ns	Table 4.2: CPU→FPGA MMIO (32 cycles)
$D_{a.2}$	224 ns	Application dependent
$D_{a.3}$	75 ns	Application dependent
$D_{a.4}$	54 ns	Table 4.2: CPU→FPGA SM (36 cycles)
D_a	401 ns	$D_{a.1} + D_{a.2} + D_{a.3} + D_{a.4}$
α	$0.72 - 56N^{-1}$	Eq. (4.4)
N_L	96 B	Eq. (4.8)

by varying the partitioning ratio from 0.0 to 1.0 in increments of 0.001 and executing the task to completion.

Our results show that our model predicted the workload N_L, for which the cooperative approach begins to outperform the SW-only approach. This prediction was within 8% of the empirical optimal for the accumulator application. Our model was able to predict the completion time of the partitioned cooperative system with a MRE of 12% for workloads in the range 8 B to 1 KB. Over the same range, the MRE between the execution time obtained using the model and the best-case observations was 2%.

4.7 Chapter summary

In this chapter I presented a model for partitioning a task between a CPU and tightly-coupled programmable logic for all cooperatively processed workloads. This model can be applied to short-running tasks as it considers both the CPU overhead and latency of communication between compute resources.

We have also reported the CPU overhead and latency of short MMIO and SM transfers

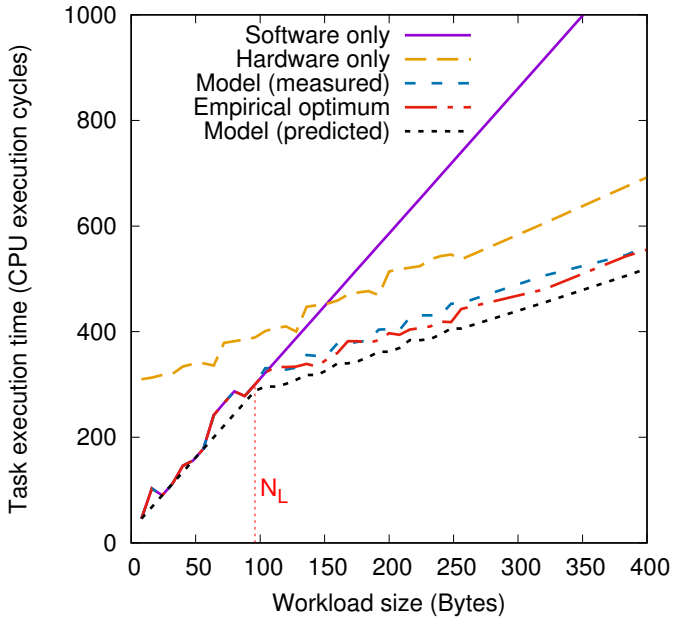


Figure 4.10: Accumulator execution time for α partitioning.

between the CPU and the FPGA on the Zynq-7020 device. These parameters, as well as interconnect delays, were used in our model to estimate the partitioning and completion time of a suitable application.

Our model was able to predict, within 8%, the workload at which we benefit from task partitioning for a generic stream-based application. The task completion time was predicted by our model with an MRE of 12%. We used our model to calculate a partitioning ratio that reduced task completion time to be within 2% of the empirical optimum.

Our model considers a single task that benefits from a single accelerator and assumes that the accelerator has been configured before the task begins execution. In the next chapter, I extend this model to a dynamic multitasking environment in which the programmable logic hosts a number of concurrent, reconfigurable accelerators that are shared between many diverse tasks.

Chapter 5

Accelerator sharing in multi-user environments

This chapter is focused on whether FPGA technology can be used to improve the performance of general-purpose systems, such as personal computers and mobile devices. Unlike special-purpose systems, general-purpose systems execute a variety of short-running tasks in a multitasking environment. When accelerating a short-running task, the overhead of migrating a task from CPU to FPGA and back can quickly outweigh any improvement in execution time. In order to accelerate such tasks, we must minimise the initiation overheads of using the HW. These include configuring an appropriate accelerator and transferring work to it.

A significant task migration overhead is the time required to configure the FPGA to perform a desired function. Ideally, one avoids the recurring configuration overhead by sharing accelerators between applications. In this way, accelerators can stay resident and be reused by other applications. However, accelerators are typically designed to be specific to the application to be accelerated and are difficult to reuse [ISA16].

Shared libraries, on the other hand, are designed to contain functions that are common to many general-purpose applications. Examples of such functions from the C standard

library are *strcmp* for comparing two strings and *qsort* for sorting an array. One might use *qsort* to sort database query results and *strcmp* to compare text fields between records, thereby defining the desired ordering.

Shared libraries are provided by the system and are linked into applications at run time. This reduces application size since common sections of application code and data need only be stored once on disk. Once loaded, read-only sections of these libraries can be shared across applications to conserve memory and cache footprint.

Our motivation for choosing to accelerate shared libraries extends to the inherently high engineering effort of FPGA accelerators. Because shared libraries are shared across many applications, the development costs can be divided across those applications. With lower development costs, there is greater incentive to optimise the design.

Shared libraries also make applications more portable across platforms by providing abstract interfaces to the application. This problem is notoriously difficult to manage for FPGA HW, for which a new bitstream is needed for each combination of application and FPGA device. Shared libraries, on the other hand, can be ported to new HW without modifying the application. This is particularly important when the end user does not have access to the source code of the application, but can control the libraries to which it will be linked at run-time.

A secondary overhead for task migration is transferring data for access by the FPGA. This overhead has been addressed by HW advancements in heterogeneous CPU-FPGA systems. Vendors are providing cache-coherent interconnects that eliminate the need for designers to manage a consistent view of memory content between the FPGA and CPU (Section 2.6). Additionally, the CPU's virtual memory model can be replicated by the FPGA. This provides consistent memory addressing between HW and SW. The overhead of calling the OS to prepare memory for access by the FPGA is eliminated.

5.1 Contributions

The objective for this work is to demonstrate the benefits of accelerating functions that are common across concurrent applications in general-purpose systems.

The core contributions of this work are:

1. A framework for sharing accelerators concurrently across multiple, diverse applications on a multi-core platform (Section 5.4).
2. An efficient method for transferring function arguments between HW and SW (Section 5.4.3).
3. Algorithms for cooperatively partitioning work across CPU and FPGA-based accelerators in a multitasking environment (Section 5.4.7).
4. A performance evaluation of our argument transfer method with respect to traditional methods (Section 5.5.1).
5. An evaluation of the potential for FPGA-based shared library accelerators to improve the overall performance of general-purpose, multitasking systems (Section 5.5.2).

5.2 Publications

The work in this chapter is in the process of being drafted into a paper submission. This work is my own, with shepherding provided by my supervisor, Oliver Diessel.

5.3 Prior work

A number of frameworks for FPGA-based accelerators have been proposed for heterogeneous CPU-FPGA systems [JK13, Egu10, GWC+14, LP09b, LP07, So07, Int19c]. These frameworks

aim to provide a uniform interface to FPGA resources for instantiating, initialising and submitting processing requests to application-specific accelerators [PAA+06, VPI05].

FPGA accelerator frameworks often have competing design goals. RIFFA is designed to provide minimal engineering effort by focusing on cross-platform and cross-OS portability [JK13]. RIFFA achieves this by sacrificing platform-specific performance enhancing features. EPEE, on the other hand, is optimised for performance at the cost of portability [GWC+14]. OS-specific features, such as zero-copy, are used to minimise latency on Linux-based systems.

From the perspective of the application developer, our framework is optimised for both performance and portability. Accelerated shared libraries need only be ported to each platform, rather than for each application-platform combination.

Accelerator frameworks typically provide a blocking interface to their applications. Multi-threading has been proposed as a method to ensure an application can progress when a blocking call to an accelerator is made [Egu10]. This model follows a family of work that proposes the use of light-weight SW threads as wrappers for HW threads [LP09b, So07]. A HW thread is a task that is being performed by an accelerator. The SW thread is used as a proxy between the accelerator and the host application or the OS.

Our framework is similar to a HW thread scheme except that the proxy thread is not expected to block until the HW thread has completed. Our proxy thread assists the HW thread by partitioning the workload, initiating the HW threads, and processing part of the workload itself.

Our framework partitions work between HW and SW for parallel execution on the available resources. A comparable approach to workload partitioning and processing is the foundation of the OpenCL standard [Khr19]. OpenCL considers loops within code as candidates for partitioning and parallel execution. Each iteration of the loop is called a work-item and can be scheduled on any number and combination of computational units, such as CPUs and FPGAs. Using OpenCL, an FPGA has been viewed as a resource that

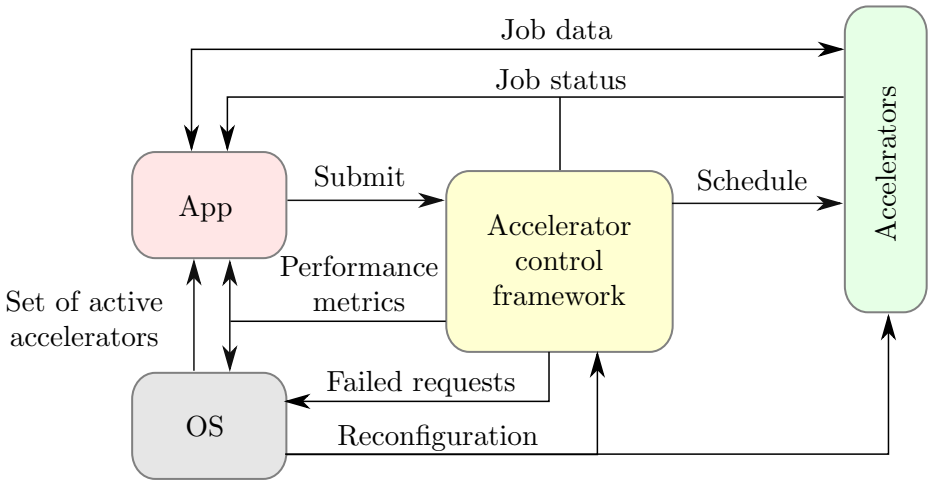


Figure 5.1: Overview of accelerated shared-library framework.

can change its function depending on the nature of the work to be scheduled [VPK19].

FPGA HW has traditionally been used to accelerate long-running tasks. This allows partitioning algorithms to be complex without adding significant overhead to completion time. It also allows for scheduling algorithms with application-domain knowledge and a complete set of sub-tasks to be scheduled over a long period of time [VPK19].

In contrast, our target domain is general-purpose systems that may host a large number and variety of short-lived applications. Task arrival times are sporadic and difficult to predict. It is not possible to formulate a globally optimal schedule for tasks and resources.

5.4 Framework architecture

Our framework provides middleware to connect applications to FPGA-based accelerators (Figure 5.1). The framework and the OS work together to detect and predict application demand for common functions. The OS responds to the anticipated demand by ensuring that a useful set of accelerators are active using DPR. The framework is implemented in HW to avoid OS system call overhead by allowing applications to submit jobs directly to the framework.

5. Accelerator sharing in multi-user environments

In this work, we identify *tasks* as abstract operations that must be performed. We assume that tasks can be partitioned into smaller *subtasks*. When such a subtask is submitted to a compute resource, it is referred to as a *job*. Jobs cannot be partitioned further and must run to completion.

The framework provides performance metrics to the application so that the application may choose the set of compute resources for jobs in order to minimise task execution time. The metrics include the estimated wait time of a submitted job and the current total throughput provided by the configured accelerators. If the application calculates acceleration to be beneficial, part of the workload will be processed by the CPU and the rest will be submitted to our framework for processing by one or more accelerators. When our framework schedules the job on an appropriate accelerator, the designated accelerator gains access to the memory address space of the application. The accelerator updates the job data and status in application memory during processing. If an accelerator is not present for the submitted job, the framework reports an error to the application and notifies the OS of the demand for the new accelerator. We expect most GP tasks to be small enough that they can be processed in SW before an accelerator for that function is configured. Our goal is therefore to ensure that an accelerator is ready for that function's next use when that function is found to be requested frequently.

The HW for our proposed framework (Figure 5.2) can be divided into four parts: ❶ the reconfigurable accelerator slots for cooperative processing; ❷ the IOMMU to provide a unified address space between the CPU and accelerators; ❸ the CSR manager for mapping job parameters to the CSR of the target accelerator and; ❹ the system monitor to collect and report performance statistics and to initiate reconfiguration events. Each of these are detailed along with the operating principles of the system in the following subsections.

5.4.1 Reconfigurable slots

The reconfigurable slots provide areas in the FPGA fabric to instantiate the accelerator

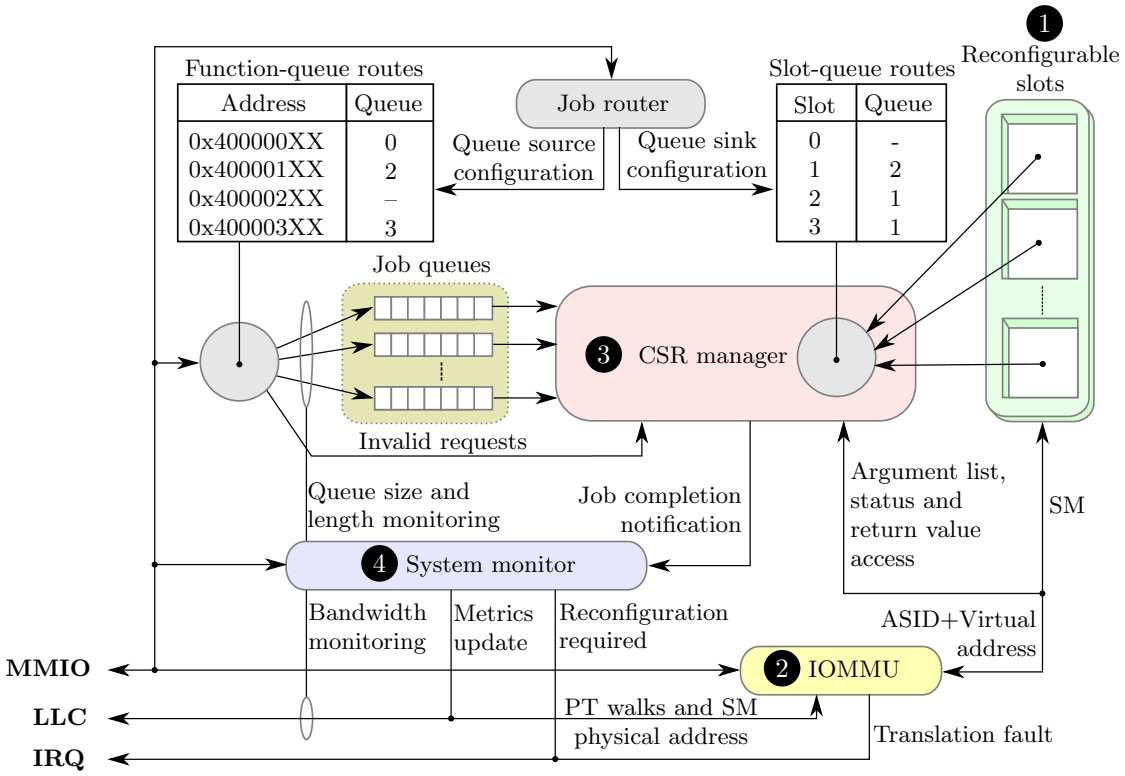


Figure 5.2: Hardware system of accelerated shared-library framework.

logic. Given a large number of functions to be accelerated and a limited amount of FPGA area, we expect accelerators to be instantiated, retired or replicated at run-time in response to changing demands using DPR [VPKG18].

Our framework provides a standard interface to each slot such that accelerators are interchangeable. The interface that our framework provides includes an MMIO port for accelerator control, a cache-coherent SM port for data access and a signal to indicate to the framework when a job can be retired and the next can be scheduled.

The framework does not provide mechanisms for preemptive scheduling of HW jobs due to the associated FPGA area, design complexity and time overhead. Preemptive scheduling involves periodically interrupting a HW job and saving its state so that it can be resumed at a later time [SLM00, MG13, KP05, LHLT10, HTK15]. Another job can then be selected, restored and resumed. To support preemptive scheduling, both framework and accelerator must provide logic for saving and restoring accelerator registers. Unlike preemptive scheduling of SW jobs on a CPU, there is no bound to the number of registers that must be saved and restored. Therefore, there is no bound on the time and memory required to change accelerator state. Instead, we rely on scheduling small amounts of work to completion [VPK19].

Without preemptive scheduling of HW jobs, a single application can monopolise an accelerator by submitting a long-running, uninterruptible job. To mitigate this, a bound is placed on the size of a submitted workload. Long-running tasks are expected to be divided into smaller sub-tasks that can be run to completion in a short period of time. For example, a large image processing task can be partitioned into blocks. Each block can be scheduled for processing by one or more accelerators. This partitioning provides the scheduling flexibility required to frequently switch between tasks without the overhead of saving and restoring a large number of registers.

Partitioning also allows SW to spread a task across multiple accelerator replicas. This is particularly important when the number of accelerators changes after jobs have been submitted. With inadequate partitioning and no mechanism for repartitioning within the

framework, the additional accelerators cannot be used.

For tasks that cannot be broken down into short running subtasks, the programmable logic can be partitioned such that some accelerators operate within our framework and others operate as traditional bare-metal accelerators.

5.4.2 IOMMU

A traditional accelerator will access memory via its physical address rather than the virtual address of the application. A traditional application must therefore perform a manual translation of each contiguous virtual range of memory to provide a set of physical addresses that the accelerator can use. This translation requires a call to the OS and is an overhead that we endeavoured to avoid for our short-running library functions. To that end, our framework provides an IOMMU for the hosted accelerators that performs the same function as the MMU of the CPU.

Our IOMMU accesses OS-provided page table structures to: 1. translate any virtual address provided by the application to a physical address that can be used by the accelerator; or 2. interrupt the CPU such that the OS can respond to failed translations. With the set of valid translations controlled by the OS, memory need not be pinned to ensure residency and memory access can be restricted such that one application cannot use an accelerator to access the memory of another application.

Examining page table structures is a time-consuming process. Therefore MMUs typically include a TLB to cache frequently translated addresses. Unlike prior work [WC17], our IOMMU features a unified TLB: a single translation cache is maintained and serves all accelerators. Translations performed for one accelerator can therefore be reused by another accelerator.

The unified TLB is beneficial to tasks that are partitioned in space or time. Cached translations are reused when a task is partitioned across replicated accelerators. Similarly, translations are reused when a task is partitioned into dissimilar functions that are

executed sequentially. For example, an application may wish to render an encrypted Joint Photographic Experts Group (JPEG) image file. The application must first decrypt, then decompress, and finally transform the file content such that it can be displayed at the correct scale on screen. Each of these operations can be assisted by HW acceleration.

A unified TLB that serves multiple parallel accelerators implies that all transactions and translations are tagged with an address space identifier (ASID). This ASID differentiates a virtual address of one application from that of another application. Since an application may be malicious or contain bugs, we cannot assume that an ASID provided by an application is valid. Instead, it is the framework that must identify which application has submitted each job and associate it with the correct ASID.

We use the address of MMIO transactions to the framework to identify the applications from which they originate and the MMU of the CPU to restrict the addresses that the application can access. This ensures that an application's identity cannot be forged and prevents one application from submitting a job that may access the private data of another. Our framework extracts the ASID from the most significant address bits of the MMIO transaction that is used to access the CSR of the framework. We use the MMU of the CPU to prevent an application from accessing the physical address range that is associated with other ASIDs. Once decoded, the ASID is bound to the submitted job and will be used to tag any SM transfer that is submitted to the IOMMU for the application.

Our approach to binding jobs with ASIDs requires little FPGA area. One must only consider the additional signals and registers required to carry a wider address bus from the CPU to the framework.

5.4.3 Job queues

The job queues hold SW-submitted jobs until they can be scheduled on an accelerator. This frees the CPU to continue execution as soon as the job has been submitted. In its simplest form, a job references a function and contains a set of function arguments. The

arguments provide input data directly and/or provide the location of IO buffers in memory.

In order to minimise OS overhead for short-running tasks, the job queues are accessed directly by the application via MMIO. The job queues thereby define the application binary interface (ABI) between SW and HW.

The shared library SW submits a job by transferring a function identifier (ID), function arguments and a job size to our framework. The function ID must be decoded immediately to select the destination queue for the incoming job. We therefore provide the function ID in the same way that the ASID is provided: by encoding it into the address bits of MMIO transactions to the framework.

It is important that thread safe library functions remain thread safe after they are accelerated using our framework. A thread safe library function is a library function that allows multiple threads within an application to safely execute that function concurrently without additional synchronisation. A thread safe library function typically does not modify resources that are shared across threads, such as global variables. For example, two calls to a library function may safely operate on two independent data sets. However, the accelerated shared library introduces a shared resource: the accelerator framework. Without synchronisation, the configuration of the CSR of the framework by one thread can be interrupted and overwritten by another. One solution to such inter-application concurrency is to replicate the CSR of each accelerator for each application. Unfortunately this does not address intra-application concurrency concerns – each application may contain many concurrent threads.

The ABI hence presents two design problems for shared library accelerators: 1. How should multiple threads in one or more applications submit jobs at the same time? 2. How many function arguments should be transferred via the job queue registers and how many via SM (e.g. the stack)?

As it turns out, our solution to problem 2) is a bi-product of our solution to problem 1). By restricting the amount of data that is transferred to the word size of the MMIO data

Table 5.1: Job queue MMIO address encoding.

Bit range	Number of bits	purpose
31 - 30	2	Constant: FPGA base address.
29 - 21	8	ASID of origin.
20 - 12	8	Function ID.
13 - 2	12	Workload size.
1 - 0	2	0 (Word alignment).
Total	32	

bus, jobs can be submitted in a single atomic MMIO transaction. There is then no need to replicate the CSR of the accelerators for each application, nor to synchronise access from each thread to the CSR using locks. Because the MMIO data width is typically the size of a pointer, the data that is transferred is a pointer to the function's arguments. Although it takes longer for the accelerator to fetch a small number of arguments from memory than it would to transfer them via MMIO (Chapter 4), we avoid CPU thread synchronisation time and using FPGA block RAM (BRAM) resources. A quantitative study of the overheads of this method is presented in Section 5.5.1.

Storing the workload size with the argument list pointer in the job queues allows the framework to estimate the wait time for arriving jobs. When these metrics are exported to the application in real-time, they assist the application in deciding how much work to offload to the accelerator(s). However, to ensure an atomic transfer that incorporates the additional data, an MMIO bus that is two words wide is required. Because the bus provided by our target platform is only one word wide, we encode the workload size in the remaining address bits of the MMIO transaction. Our encoding for the address that an argument list pointer should be transferred to is shown in Table 5.1.

Apart from the arguments to be transferred, we must also consider how accelerator status and function return values are shared between the accelerator and the application. When solving this problem, we must consider that multiple threads within the application may be concurrently submitting multiple parallel jobs to the accelerators. The location of the status of each of these jobs must be known and accessible at any time such that each thread can determine when its submitted jobs are complete. Our solution is to store status

values with the list of arguments. When the job is complete (or an error occurs), the framework updates both the status and the return fields using the argument pointer that was submitted when the job was enqueued.

5.4.4 Job router

The job router defines the path from the application HW/SW boundary to the accelerator for incoming jobs. This path specifies where jobs will be queued and to which slots the job can be scheduled for processing.

The router first decodes incoming jobs by function ID and inserts them into the queue that was assigned for that function. With many more accelerated functions than queues, it is not possible to concurrently route all functions to accelerators. Jobs that arrive for functions that are not being accelerated are submitted in error and will not be queued. Instead, their job status will be updated to report an error. Because the CPU cannot continue until the submitted transaction to the framework have been accepted, such failed jobs are handled with highest priority. Jobs for functions that are routed to full queues are treated in the same way, except that a different error code is returned.

Recording the IDs of functions that were requested but unavailable indicates to the framework a demand for the missing accelerators. When the current set of instantiated accelerators is re-evaluated, the framework may replace an unused accelerator with an accelerator for the requested function. With knowledge of the configured accelerator set, the shared library should not submit work for a function that it finds is not currently being accelerated; it should only submit such a job if it wishes to inform the framework that there will be demand for that function in the near future.

The framework supports the replication of accelerators to improve the combined throughput for functions that are in high demand. This is achieved by allowing each queue to be served by multiple accelerator slots. For scalability, we design this as a one-to-one slot to queue routing rather than a one-to-many queue to slot routing. When scheduling a

queued job to an accelerator, our design replaces an O(n) idle accelerator search for each nonempty queue with an O(1) nonempty queue lookup for each idle accelerator. Since each queue need not store a list of associated accelerators, our scalable design reduces BRAM requirements and simplifies the logic.

Since our framework does not support saving and restoring accelerator state, and to avoid jobs being stalled in a queue indefinitely, we constrain the system such that a slot cannot be reconfigured while it is processing a job, or while it is the only slot that is servicing a nonempty queue.

5.4.5 CSR manager

The CSR manager provides the abstraction needed to connect a traditional MMIO CSR-based accelerator to our framework. Once a job is ready to be scheduled on an accelerator, the CSR manager retrieves the job function arguments from SM using the argument list pointer that was provided by the application. It then copies the arguments to the accelerator CSR. The CSR manager then updates the control registers of the accelerator so that the accelerator begins processing. Once complete, the CSR manager retrieves the return value from the accelerator and copies it back to the application using SM.

The CSR manager assumes that accelerator control registers conform to a specific design pattern (Table 5.2). The pattern matches that of an AXI peripheral that was generated using Vivado HLS. This provides a uniform way in which the manager controls all accelerators, except in the number of data words (function arguments) that the manager must transfer from the argument list pointer to the accelerator CSR.

The CSR manager uses little logic for identifying the next queue-accelerator pair for scheduling (Figure 5.3). Per-slot multiplexers are instantiated to provide a vector to indicate if the associated queue for each slot is empty. The input of those multiplexers is a vector that identifies which queues are empty and the slot-queue routing table is used to select which empty signal is observed at the output of each multiplexer. A bit-wise NOR of

Table 5.2: Assumed accelerator CSR and addressing.

Offset	Symbol	Description
0x00	CTRL	Accelerator control (start/done)
0x04	GIE	Global interrupt enable
0x08	IER	Interrupt enable
0x0C	ISR	Interrupt status
0x10	Ret	Function return value
0x18	Arg0	First function argument
⋮	⋮	⋮
0x18 + $N \times 4$	ArgN	Last function argument

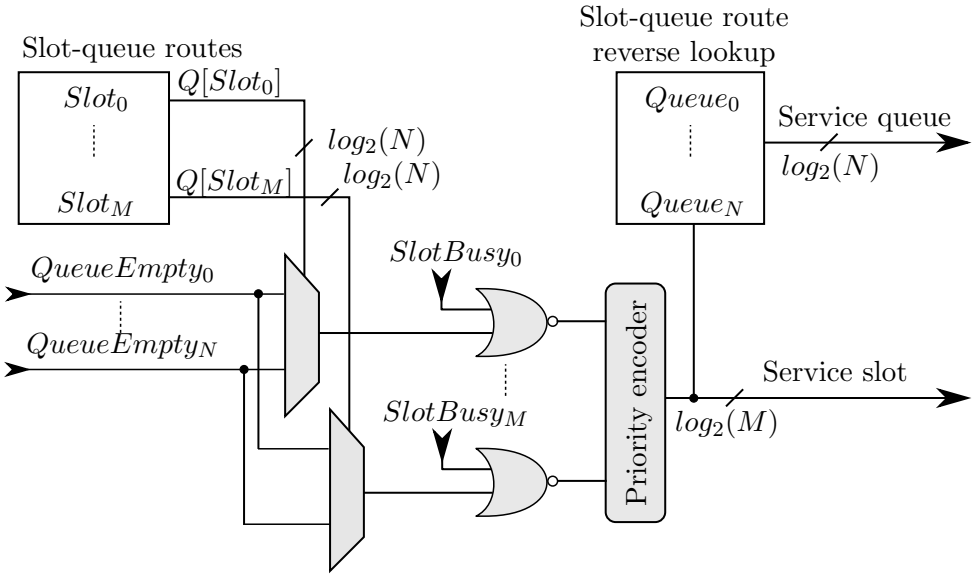


Figure 5.3: Accelerator service queue-slot selection logic.

the multiplexer outputs and the *busy* signals provided by each accelerator indicates which queue-accelerator pairs can be serviced by the CSR manager. A priority encoder is then used to choose which slot will be served next, and a reverse lookup table identifies the queue to which it was bound. Assuming a small number of accelerators and a relatively large processing time for each, we expect our priority encoder to schedule HW jobs in round-robin order in practice.

Once a queue-accelerator pair has been selected, the CSR manager uses the argument list pointer and ASID provided by the queued job to copy function arguments from SM to the CSR of the accelerator. The CSR manager then configures the SM interface of the slot to

tag all transactions with the provided ASID. Thereafter the CSR manager writes to the appropriate control register of the accelerator to start the job. Finally, the CSR manager is free to serve other queue-accelerator pairs.

When an accelerator has finished a job, it must update the job status and, where appropriate, the return value of the accelerated function call in SM. Each accelerator provides a signal to indicate when a job has finished. Those signals are aggregated and processed by a priority encoder to choose which accelerator will be served next. As before, a round-robin scheduling is expected in practice due to the execution delay of a job.

Although the job completion handler runs concurrently with the job initiator, it must be stalled when an invalid job arrives (e.g. a job for which the function is not routed to a queue). In that event, we apply back pressure to the MMIO interface to prevent further jobs from arriving from the CPU. We then wait for any in-flight status and return value updates to complete and report an error status for the submitted job as soon as possible. Once reported, back pressure is released and the CPU can submit more jobs.

5.4.6 System monitor

The monitor provides run-time system performance statistics that assist shared libraries in optimally partitioning workloads. The monitor provides the total size and number of jobs in the queue such that applications can estimate the wait time of any job that the application submits. The monitor also provides the available SM bandwidth such that the expected throughput of memory-bound accelerators can be penalised when memory bandwidth is limited. Lastly, the monitor reports the utilisation of accelerators to the reconfiguration manager (Section 5.4.7) such that the configured set of accelerators can be optimised for the demonstrated demand.

We use worst-case values for HW performance estimation because it is difficult to recover from over-utilisation of HW resources. Once a job has been submitted to our framework, it cannot be modified, repartitioned or removed. If a HW job finishes later than expected,

the application must idle until the job has finished. However, if HW jobs finish earlier than expected, SW can repartition the remaining work across the accelerators using updated performance metrics.

We measure the available SM bandwidth (BW) by counting the number of clock cycles that the SM bus is inactive for over a fixed time period. The application can use the reported available bandwidth to determine if jobs scheduled to n_a accelerators, that each require BW_a bandwidth, will be compute bound or memory bound. An application can then scale the theoretical throughput of each accelerator (X_a) to find the estimated effective throughput (X'_a) per accelerator (Eq. 5.1).

$$X'_a = X_a \times max\left(\frac{BW}{n_a BW_a}, 1\right) \tag{5.1}$$

We estimate the amount of time that a new job will wait in the queue (D_q) to be the sum of two components (Eq. 5.2): 1. The sequential initiation delay D_a of scheduling the existing q queued jobs on the accelerators; and 2. The parallel processing delay of those q queued jobs and n_a running jobs, each of which requires W_i work to be completed. Since accelerators do not provide real-time updates of progress, D_q represents the worst-case queue wait time.

$$D_q = D_a q + \frac{1}{n_a X'_a} \sum_{i=1}^{q+n_a} W_i \tag{5.2}$$

The monitor stores metrics in cache-coherent SM to minimise access latency from applications. The framework need only share metrics for functions that are currently accelerated. Hence metrics could be bound to the queues to which both functions and slots are routed. However, applications are not made aware of reconfiguration events; the framework may reroute queues between the application identifying a function's assigned queue and analysing its metrics. For this reason, metrics are bound to each function and only updated if the function is being accelerated.

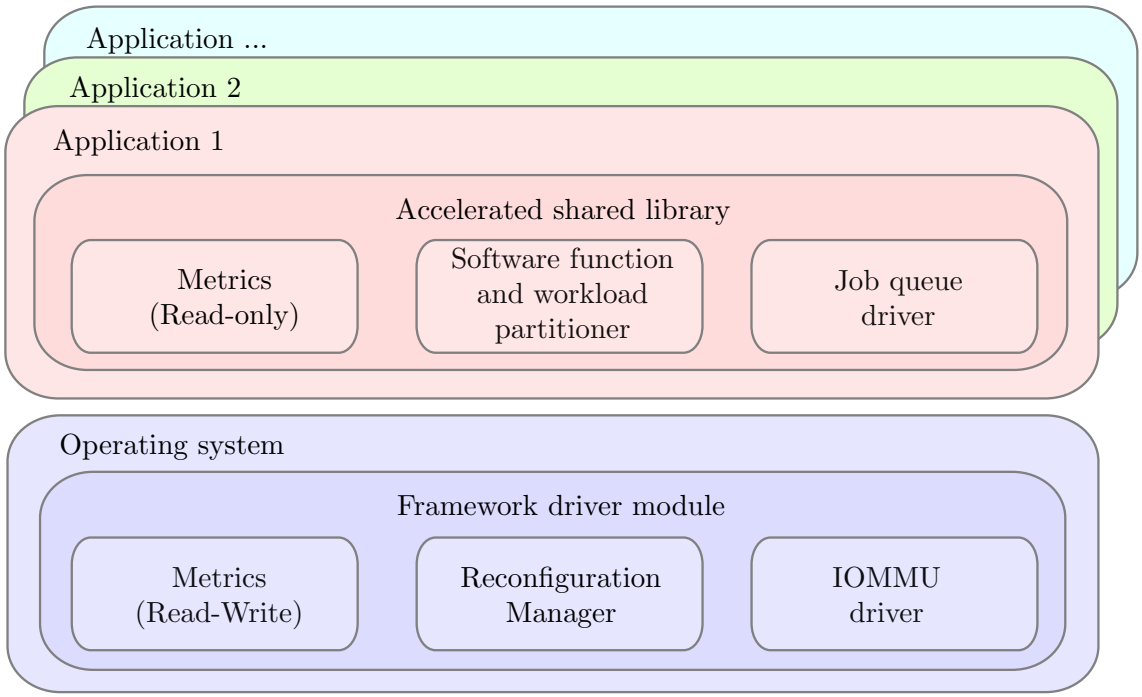


Figure 5.4: Software system of accelerated shared-library framework.

Since it is not feasible to update all metrics for the number of active functions in a single memory transaction, the framework maintains a bitfield of metrics that are scheduled to be updated and updates them sequentially with independent SM transactions.

5.4.7 Software system

The SW system is composed of the OS and one or more applications (Figure 5.4). The OS is modified to include a framework driver module that monitors and controls the configuration of accelerators in HW. The runtime environment that is provided by the OS links applications to the accelerated shared library functions.

In this section, we describe the behaviour of the SW system chronologically from the time the system is booted to the time an accelerated shared library function has run to completion and returns control to the application.

Boot

At boot time, the OS determines whether compatible HW for our framework is connected to the system and, if so, installs the framework driver module into the OS. The module includes the IOMMU drivers, IRQ handlers, reconfiguration manager and a SM page for monitor metrics.

Periodic monitoring and reconfiguration

Once the system has booted, the reconfiguration manager periodically reconfigures the set of accelerators and associated routes to match user demand. The reconfiguration manager determines the demand for each accelerated function by periodically examining two metrics that are reported by the system monitor.

One metric for demand is the aggregate work that remains to be processed. However, the remaining work that is reported by the monitor can be misleading: unless the monitoring period is less than the processing time of a job, the queued work can be empty at one monitoring sample, then filled and drained before the next. In this case, the manager would observe no work in either sample and conclude that the function was unused.

To overcome this limitation, the reconfiguration manager also examines the amount of work that was submitted between two samples. When examining the submitted work, we do not consider whether or not that work has been processed during the monitoring period. However, this metric may exclude a potentially large amount of work that remains to be processed when work was submitted before both samples were recorded.

Therefore, the manager considers both submitted work and remaining work when estimating demand. Specifically, we define the demand for a function over a monitoring period to be the maximum of the remaining work and the submitted work. We then define the demand for an accelerator to be the demand for the associated function divided by the number of accelerators that have been configured for that function.

The reconfiguration manager (Algorithm 1) begins by iterating over all accelerators that are idle. Although these accelerators are idle, they may have been used heavily during the last monitoring period. Therefore, the slot is a candidate for reconfiguration only if the demand for the accelerator is below the constant C_RECONFIG.

Algorithm 1: Periodic reconfiguration algorithm.

Input: Accelerator status and demand; function request bitmap.
Result: Improved set of configured accelerators and job routes; Clears function request bitmap.

```
1  foreach s ∈ idle slots do
2      if demand for s is less than C_RECONFIG then
3          // Search for an overutilised slot to assist
4          s_h := accelerator with highest demand;
5          if demand for s_h is greater than C_REPLICATE then
6              reconfigure s to replicate s_h;
7              route s → queue of s_h;
8          else if function requests ≠ {} and available queues ≠ {} then
9              // Search for a new function to accelerate
10             f := next function request;
11             q := find available queue;
12             reconfigure s for f;
13             route s → q;
14             route f → q;
15             clear request for f;
16         end
17     end
18 end
19 clear all function requests;
```

Functions that are not routed to queues provide little information about function demand. They only provide a single bit that indicates if the accelerator was requested at least once during the sample period. On the other hand, configured accelerators provide a rich set of metrics that better quantify demand. Therefore, we prioritise accelerator replication over new function requests. The reconfiguration manager searches the set of accelerators to find the accelerator that has the highest demand. If the demand for that accelerator is above C_REPLICATE, the idle accelerator will be retired and the slot will be reconfigured to assist the overloaded accelerator.

If the slot has not been reconfigured to replicate another and a new function has been

requested, we reconfigure the accelerator to serve one of the new function requests. Due to the reconfiguration delay of the accelerator, the configured accelerator might be used for the function call that requested it. However, the request implies that the accelerator may be needed again in the future. Because the accelerator provides no information on whether it will be used again in the future, we always reconfigure accelerators that are idle and have a demand below C_RECONFIG for new function requests when a replication candidate is not found.

The reconfiguration manager continues this process until it has considered all slots in the framework. It then clears all function requests to ensure requests are recent. Finally, the reconfiguration manager waits for an interrupt to indicate that the next monitoring period has expired before repeating the process.

Login

When a user logs into the system, the login scripts configure the runtime system to use the accelerated libraries if available. The scripts configure the runtime by replacing the search paths for SW-only libraries with paths to their accelerated alternatives [BNP10]. By providing both SW-only and accelerated versions of the library we avoid overheads, such as checking accelerator availability, for each function call. This method also allows users to opt-out of using the accelerated libraries by overriding the configuration made by the login scripts.

Load

The process of loading an application for execution is unchanged when our framework is used. The dynamic linker identifies unresolved symbols in the application that it binds to shared library symbols. The dynamic linker finds these libraries via the search paths that were adjusted to include or exclude the accelerated libraries at login time. Once all symbols have been resolved, the application is initialised.

Initialisation

Once the application is loaded, the dynamic linker calls the initialisation function of each linked library. If that library is an accelerated shared library, the initialisation function of that library executes a system call to the OS to prepare the application to use the framework. The system call directs the OS to 1. allocate a new ASID for the application; 2. register the ASID and page table of the application with the IOMMU; 3. map the appropriate job queue CSR alias into the virtual address space of the application for direct access; and 4. map the read-only monitor metrics page into the virtual address space of the application.

Call

When an application calls an accelerated shared library function, the function partitions the workload into small jobs and schedules them for execution on the available compute resources. Our partitioning algorithm reduces the overall completion time of the function by exploiting the parallelism provided by the FPGA-based accelerators. Rather than waiting for the accelerators to complete, the function also processes some of the workload on the CPU itself.

Optimum workload partitioning requires an accurate prediction of the completion time of each job. To minimise overall completion time, partitioning aims to ensure that both HW and SW partitions finish execution at as close a time as possible. If the SW partition finishes before HW, CPU execution cycles are wasted while waiting for HW to complete. If HW partitions complete before SW, an opportunity for parallel execution is lost.

SW should *dynamically rebalance* partitions in multitasking environments because the completion time of SW jobs are difficult to predict. Such tasks can be interrupted at any time to allow other tasks to consume their fair share of CPU time. In the extreme cases, the decision to send *all* or *none* of the workload to the accelerator(s) depends on where a process is up to in its scheduling time slice. If the task is at the beginning of its time

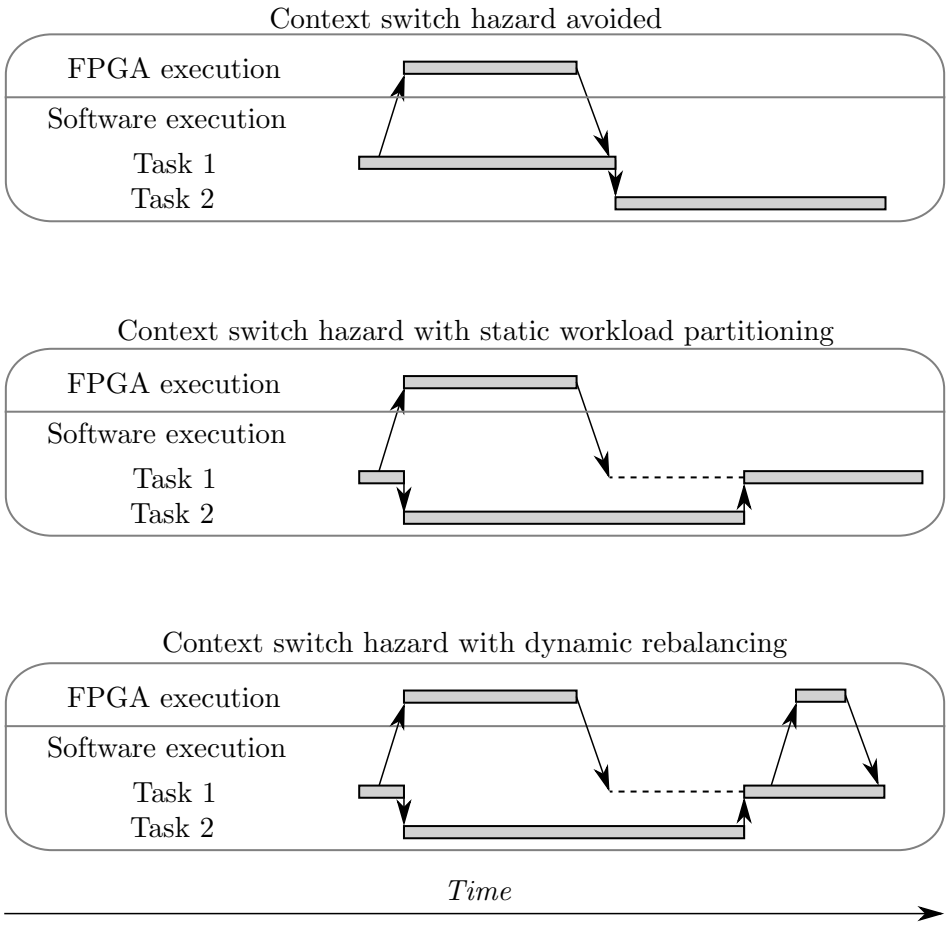


Figure 5.5: Context switch hazards for HW/SW workload partitioning.

slice, a SW-only approach could be optimal due to the initiation delay of HW jobs. On the other hand, if its time slice has almost expired, a HW-only approach could be optimal. In that case, HW jobs continue uninterrupted and could complete before the application is rescheduled for SW execution (Figure 5.5). We refer to this issue as a context switch hazard. Because there is no way of knowing where a task is up to in its time slice, in our framework, SW periodically evaluates the progress of HW jobs and rebalances the work to minimise the completion time of each task.

The completion time of a HW job is also difficult to predict. Although in our framework individual HW jobs are uninterruptible, it is difficult to accurately predict how long a job will spend queued before it is scheduled. The framework may increase the number of accelerators for a function at any time, thereby allowing more jobs to be scheduled in

parallel. With more parallelism, the remaining jobs will wait for less time in the queue. Like the context switch hazard identified above, such mispredictions result in HW completing earlier than expected and can be resolved with periodic monitoring and rebalancing in SW.

To improve scheduling flexibility and parallel execution, in our framework SW submits more jobs to accelerator control than there are resident accelerators. Submitting a single large HW job for each replica allows those jobs to continue processing in the event of a context switch hazard. However, it reduces response time to accelerator replication because jobs cannot be repartitioned to use the additional accelerators once submitted. On the other hand, submitting a single small job to each accelerator allows us to respond quickly to accelerator replication, but it limits the amount of work that can be processed if the application's time slice expires. Further, the monitor will observe that the accelerator is underutilised because we have withheld a lot of work from the framework in the hope that the workload can be better partitioned across replicas when they become available. Our partitioning algorithm hence divides the HW workload further into a number of smaller HW jobs. In this way, dynamic replication can be exploited while maximising the work performed during a context switch hazard.

The partitioning algorithm uses both offline and real-time performance metrics to decide how the workload will be partitioned. Offline metrics, such as throughput and initiation overhead, provide an estimate of completion time in an environment where there is no competition for resources. Online metrics, such as queuing delay and available bandwidth, are provided by the system monitor (Section 5.4.6) and are used to adjust the offline performance metrics for current demand.

For small workloads, the delay and CPU overhead of accelerator initiation, and synchronisation can outweigh the reduction in execution time. Eq. (5.3) (Derived in Section 4.4) can be used to calculate the workload size for which we begin to benefit from using the accelerator (N_L). This equation considers the compute throughput of the CPU (X_c) and the accelerator (X_a) as well as the latency (D_a) and CPU overheads (O_c and O_0) of communication. We adjust Eq. (5.3) to Eq. (5.4) by considering the bandwidth-adjusted

accelerator throughput (X'_a). If the workload size of the submitted task is greater than N_L, but there is no accelerator configured, the application submits an empty job for the function. This job will be rejected by the framework because the function is not being accelerated, but it informs the framework that there might be some future benefit to accelerating that function.

$$N_L \geq X_c \left[\frac{X_c \left(O_c - O_0 \right)}{X_a} + D_a - O_0 \right] \tag{5.3}$$

$$N_L \geq X_c \left[\frac{X_c \left(O_c - O_0 \right)}{X'_a} + D_a - O_0 \right] \tag{5.4}$$

When a task has exclusive access to one accelerator, the function can determine the fraction of the workload that should be processed by the accelerator (α) using Eq. (5.5) (Derived in Section 4.4). That equation considers the throughput ratio of compute resources and compensates for the CPU overhead and the latency of short transfers, particularly when the workload size (N) is small. For large workloads, these overheads are small relative to workload completion time and thus have less impact on the partitioning ratio.

$$\alpha = \frac{X_a}{X_a + X_c} \left[1 - \frac{X_c (D_a - O_c)}{N} \right] \tag{5.5}$$

We modify Eq. (5.5) to consider the available bandwidth, accelerator replication and queuing delay (Eq. 5.6). The effective throughput (X'_a) is scaled by the number of available accelerators (n_a) and the latency considers both the queuing delay (Section 5.4.6) and the sequential scheduling latency and CPU overhead of submitting n_j jobs to the framework.

$$\alpha = \frac{n_a X'_a}{n_a X'_a + X_c} \left[1 - \frac{X_c \left(D_q + n_j \left[D_a - O_c \right] \right)}{N} \right] \tag{5.6}$$

5. Accelerator sharing in multi-user environments

Our dynamic workload partitioning algorithm (Algorithm 2) first checks if acceleration is beneficial and, if not, processes the entire workload on the CPU alone (Line 2). If the work could be accelerated but no accelerator is available, the algorithm periodically submits an empty job to request an accelerator and processes work in C_COWORK_MAX batches in SW until an accelerator is available (Line 7). C_COWORK_MAX should be chosen to balance a quick response to accelerator configuration with the overhead of polling accelerator state. In practice, C_COWORK_MAX can be kept low because the number of accelerators is communicated via low-latency SM rather than an OS system call.

Once an accelerator is available, the algorithm begins to partition work across compute resources until all work has been processed (Line 13). The algorithm submits at least n_j jobs to the framework (Line 20). n_j should be chosen to balance the initiation overhead of each job with scheduling flexibility in case an accelerator is replicated. Our choice of $n_j = 6$ will be explained in Section 5.5.4. While waiting for HW jobs to complete, SW processes work in C_COWORK_MAX blocks (Line 21). Once HW jobs have completed, we may find that a context switch or accelerator replication reduced the amount of work that was processed in parallel by SW. In that case, the algorithm rebalances the remaining workload by repeating the procedure at Line 13.

Our framework also supports multi-core CPU architectures. In this section, we have described our system with a single threaded view only because we detail the process of an isolated call to the accelerated library function. Because the thread safety of libraries is preserved by our framework (Section 5.4.3), the application can use traditional multithreading techniques to take advantage of multi-core CPU architectures.

5.5 Evaluation

The overarching contribution of this chapter is the proposal and evaluation of a framework for accelerating common, short-running tasks. Central to that framework is a low-latency method for transferring tasks between SW and HW. Therefore, we begin our evaluation

Algorithm 2: Workload partitioning algorithm.

Input : N work to be processed;
volatile monitor metrics;
volatile n_a accelerators configured for this function;
n_j jobs to submit;
C_COWORK_MAX work to process in SW before rebalancing;

Result: N work processed

```
1  N_L := calculate minimum workload size using Eq. 5.4;
2  if N ≤ N_L then
3      Process N in SW;
4      return
5  end
6   // Process some work in SW while waiting for an accelerator
7  while N > 0 and n_a = 0 do
8      Submit empty job to request accelerator;
9      N_c := min(N, C_COWORK_MAX);
10     N := N − N_c;
11     Process N_c work in SW;
12 end
13 while N > 0 do
       // Partition work across HW and SW
14     α := calculate partitioning ratio using Eq. 5.6;
15     // Included only in a system without dynamic rebalancing.
16     Process N work in SW;
17     if α > 0 then
18         N_f := N × α;
19         N := N − N_f;
20         Submit n_j jobs of N_f / n_j work each; // n_j = 6 in our exploration
21         while HW jobs are incomplete do
22             // Included only in a system with dynamic rebalancing. While
                  processing, SW monitors HW progress to compensate for context
                  switch hazards and replication.
23             N_c := min(N, C_COWORK_MAX);
24             N := N − N_c;
25             Process N_c work in SW;
26         end
27     else
28         Process remaining N work in SW;
29     end
30 end
```

by measuring the overhead of our transfer method. We then evaluate the performance of our framework in a multi-user environment using synthetic workloads that emulate a concurrent and dynamic set of applications.

We evaluate using real HW to ensure that subtle SW and HW limitations are not overlooked. The Zedboard [Avn19] satisfies the requirements of our HW model (Section 2.8). In our experiments, we configured both CPUs to operate at their maximum frequency of 667 MHz. Programmable logic was configured to operate at 111 MHz (1/6 of the CPU speed).

We chose Xillinux-1.3 [Xil19b] as the OS and root file system for our evaluation because it was easy to setup and provides a complete user environment based on the popular Ubuntu Linux distribution. We implemented our framework as a loadable kernel module rather than modifying the kernel source directly.

5.5.1 ABI overhead

Our framework allows applications to directly and concurrently submit jobs to the accelerators. Our method avoids synchronisation overhead by transferring the memory address of all function arguments in a single atomic transfer (Section 5.4.3). However, the framework must now read the function arguments from memory and copy them to the accelerator's CSR before the accelerator can begin accessing IO buffers (when required) and processing. It is important that we measure these costs.

Besides the number of arguments to transfer, the argument access latency from the FPGA is expected to influence performance. The latency in this case is influenced primarily by the residency of memory translations in the TLB of the IOMMU. With a two-level page table, the page table walker of the IOMMU must make two random access memory reads to translate the application virtual address to a physical address. These reads have a direct data dependency: the first transaction must complete before the second transaction can begin. Once the translation is complete, reading the arguments can be done in a single AXI burst transaction. Memory translation is therefore assumed to be the most significant

influence on latency and is used as a second variable in our experiment.

We designed a custom accelerator to be used for our ABI benchmark. The accelerator performs no operation: it simply returns success once it has started. The execution time of the accelerator, as viewed from the CPU, is therefore only the time required for SW to prepare and submit function arguments and for the completion status to be read by the CPU. The execution time was measured by first reading the CPU cycle count at the beginning of the experiment using the ARM performance counters. The arguments were then transferred to the accelerator using either direct CSR access via MMIO, or by submitting a job via our framework. SW then waited in a loop until the status flag of the accelerator indicated that the job was complete. Finally, a second cycle count was taken. After subtracting the overhead of reading the cycle counter, the difference between the two cycle count readings was accepted as the makespan of the benchmark for each sample collected.

In our experiment, we varied the number of arguments to transfer and whether or not address translation was required. The size of each argument was fixed to 32-bits for simplicity while the number of arguments was varied from 0 to 32. TLB translation residency was controlled by flushing the TLB before starting the experiment. When a cached translation was desired, we ran the experiment twice, discarding the first result. We were thereby sure that the framework used the cached translation during the next iteration. Each experiment was run 1000 times.

The results show that the direct MMIO access to accelerator CSR provides the best performance when the number of arguments is less than five 32-bit values (Figure 5.6). For five or more arguments, transferring all arguments via a single pointer provides the best performance, but only when a translation table walk is not required. In the case where a walk is necessary, we found that 21 or more arguments are required before our method outperforms direct CSR access.

Error bars in Figure 5.6 indicate the 1st and 3rd quartiles of the sample distribution. The high variance in direct CSR access time is due to buffering throughout the interconnect

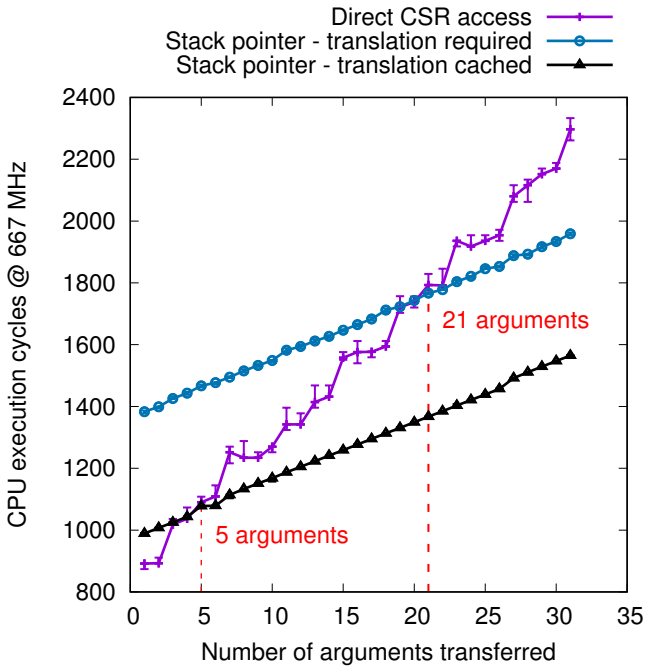


Figure 5.6: Execution time of the ABI accelerator with varying number of arguments.

that leads to multimodal delays.

Our results are promising for our method of low-latency argument transfer. Although we must transfer a number of arguments to the accelerator before outperforming direct CSR access, fewer CPU cycles are required to manage the transfer (Chapter 3). This enables the CPU to process more work in SW while waiting for HW jobs to complete.

We measured the duration of the *gettimeofday* Linux system call to estimate the additional CPU cycles required to synchronise access to the accelerator CSR. This synchronisation could take the form of acquiring a lock or using the OS as a proxy for submitting jobs. The *gettimeofday* system call requires very little work in the OS; it populates a user-provided location in memory with the current time. By providing a NULL pointer as argument, we avoided that data copy and further reduced the execution time of *gettimeofday*. We measured the CPU cycles required in the same way that we measured the cost of submitting a job to an accelerator and found that the *gettimeofday* system call requires 280 CPU cycles on our target platform. Considering that overhead, our method of argument transfer outperforms direct accelerator CSR access for any number of arguments, when address

translations are resident in the IOMMU's TLB.

5.5.2 Framework evaluation

Our aim is to reduce the execution time of general-purpose systems by providing a framework by which common short-running functions can be accelerated across multiple concurrent applications. Although these applications share accelerators, the set of accelerators and the order in which they are used differs across applications. For example, a web browser and photo editor may both decompress and render images, but a photo editor may also apply image filtering to adjust sharpness or colour. Our evaluation of dynamic performance must therefore consider the concurrent execution of a wide range of application personalities.

We are also concerned with the volatility of the set of applications that execute on a general-purpose system. The set of applications is expected to change as users move from one task to another. The optimum set of resident accelerators will change with the active set of applications. Our monitor must quickly retire and replace accelerators with changing demand.

The focus of our work is not on the accelerators themselves; it is the framework. We therefore designed synthetic applications, shared library functions and accompanying FPGA-based accelerators to emulate a wide range of application and accelerator personalities. This method also allows us to change the behaviour of the accelerator at run-time to emulate DPR.

We designed an application that emulates a range of general-purpose applications by sequentially executing randomly selected shared library functions, each with a random amount of work to process. Each library function can be accelerated and contains only a small amount of sequential code for determining which computational units will be used for processing. Therefore, our applications can be considered to be perfectly parallelisable.

To simplify the experiment, random numbers for function and workload selection follow a uniform distribution. We do not favour any particular function when making a selection

and each call is selected independently. We defined a work unit to be 1 CPU cycle for convenience and tuned a loop of SW to execute exactly N CPU cycles of work, where N is provided at run-time. That routine is called by library functions when work is to be processed on the CPU.

Similarly, we designed an FPGA-based accelerator that accepts a run-time configurable execution time. The execution time of our accelerator is set when the framework copies function arguments from SM to the accelerator CSR. Once the prescribed time has expired, the accelerator reports completion and the framework reports the job status and function return values to the application via SM. The number of arguments expected by our accelerator was fixed at 8× 32-bit arguments. The required number of FPGA clock cycles to emulate accelerator execution time is determined by the CPU/FPGA clock ratio and the desired speedup of the accelerator.

We designed both a SW-only and accelerated library. While the accelerated library partitions the requested work across compute resources, the SW-only counterpart uses only the tuned SW loop to emulate workload processing. The desired library is chosen manually before each experiment by adjusting the LD_LIBRARY_PATH user environment variable.

Our constructed system has the key parameters presented in Table 5.3. We used the default Linux SW task scheduler time slice of 10 ms. The workload size of each call to a shared library function is chosen randomly to produce CPU execution times in a range that spans the boundary of that time slice. This ensures a mixed range of context switch hazards (Section 5.4.7 and Figure 5.5). Each experiment processes a total workload that requires 8 seconds to complete when processed only in SW on a single CPU core. That workload is divided between applications to bound execution time when the number of applications is large.

We varied the monitoring period, P, throughout our evaluation. Both the monitoring period and the reconfiguration time combined determine the response time of the system to changing demand. For simplicity, we assume that accelerators require P ms of time to be configured before they can be used to accelerate a function if not already resident.

Table 5.3: Parameters of system under test.

Name	Value
Number of reconfigurable slots	8
Number of library functions	16
Number of applications	1-8 per CPU
CPU time slice	10 ms
Workload per call	1-100 ms†
Total experiment workload	8 s†
Accelerator speedup	0×-10×
Monitoring period	P ms
Idle threshold (C_RECONFIG)	0.5 ms†
Overloaded threshold (C_REPLICATE)	P / 2 ms†
Reconfiguration delay	P ms

† equivalent uniprocessor time.

C_RECONFIG and C_REPLICATE, the workload sizes at which we define accelerators to be under- or overutilised, are set to preserve recently used accelerators, but provide a high replication rate. We leave the exploration of these parameters as future work and select constants for our evaluation. We chose C_RECONFIG such that accelerators may be swapped when demand is reported to be equivalent to 0.5 ms of work or less when processed in SW. C_REPLICATE is chosen to be proportional to the monitoring period because the reported demand depends on the period over which it was measured. We set C_REPLICATE such that accelerators were replicated when the processing time of the reported demand exceeds half of the configured monitoring period.

We modify the speedup that accelerators provide in the range 0× to 10× throughout our experiments. An accelerator with speedup 0× provides no benefit; all work will be processed in SW only. Although HLS has shown speedups between 4× and 126× when compared to an Intel i5 2.67 GHz CPU [LRL+12], a heterogeneous solution has demonstrated speedups in the range 1× to 2× [KPPK11]. That range is inline with the 2.6× throughput of our accumulator case study in Section 4.6.

Our evaluation compares the makespan of a set of random application personalities when using our framework to a system that uses only SW to process the same workload. We evaluate how performance correlates with the number of accelerator slots provided by

our framework and the speedup that the accelerators provide. We also evaluate how well our system scales to a number of concurrent applications that must share access to the provided framework and accelerators.

Our experiment considers a varying number of parallel applications, each with a different seed for the generation of function call lists and workload sizes. To eliminate variations caused by disk IO, we synchronise applications using semaphores. Once all applications are loaded and ready to begin, a master application determines the start time of the experiment using the *gettimeofday* system call and records it in SM. The master then signals all applications to begin sequential execution of their function call list and associated workload sizes. Once complete, applications record their completion time and block on a semaphore until all other applications have completed. Each application then reports the difference between the time reported by the master and its individual completion timestamp. The makespan for the sample is taken as the maximum of the reported execution times.

Each experiment was repeated 25 times and the median result was collected. This was repeated 50 times, with different seeds for each application. The results were aggregated by taking the arithmetic mean over the 50 medians for each set of experimental parameters. Our final result was then normalised to the execution time of a SW-only approach that was measured using the same procedure. The maximum standard deviation across all of our experiments was 5.0%.

5.5.3 Static homogeneous accelerator cluster

We begin by evaluating a system that does not require accelerators to be reconfigured. This eliminates the effect of the monitoring period and reconfiguration delay It allows us to focus on the overhead of our framework, the workload partitioning algorithm and context-switch hazards.

We modified the function list generation of applications to utilise only a single common function across all applications. We explored the impact of varying the number of

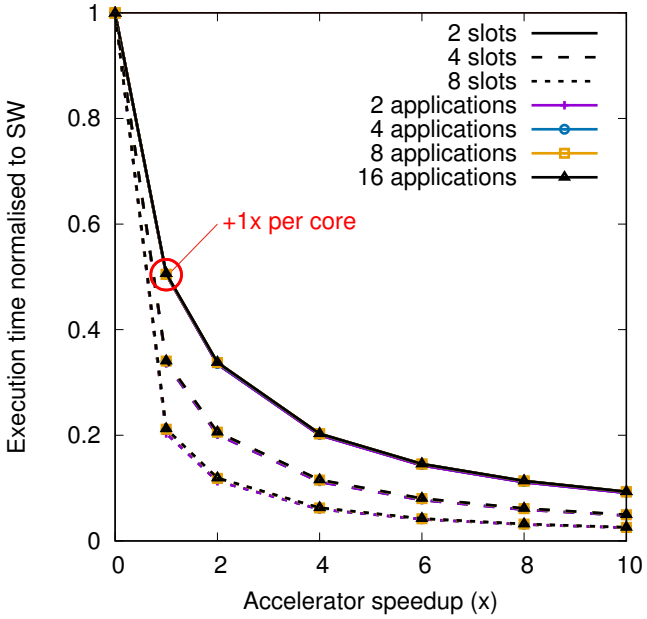


Figure 5.7: Normalised makespan when applications call one type of function.

applications, reconfigurable slots, and accelerator speedup on system performance.

For a system that implements dynamic rebalancing, our results show that the reduction in execution time when our framework is used scales with the number of accelerators and their provided speedup (Figure 5.7). Because applications execute on a dual-core CPU, the CPUs must share the provided accelerators. In the case of two reconfigurable slots, each providing 1× speedup, there is twice the compute power available to each CPU. Hence the execution time is halved.

We calculated the combined utilisation of all CPU cores and accelerators to quantify how well the available resources are used. We define the system utilisation as the ratio of the expected makespan T_e and the observed makespan T_o (Eq. 5.7). The expected makespan is the SW-only makespan scaled by the acceleration potential of the available n_s slots that are shared across N_c CPU cores. Each slot provides a speedup S and cooperates with one CPU core to process the workload (Eq. 5.8).

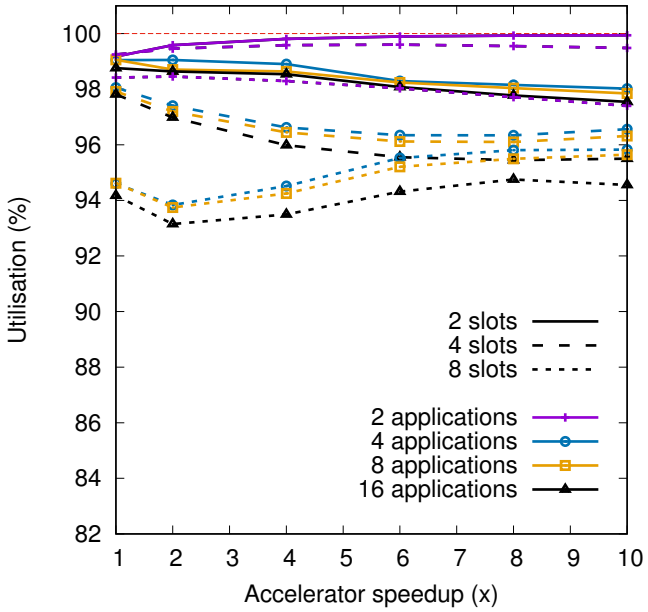


Figure 5.8: System utilisation when applications call one type of function.

$$U = \frac{T_e}{T_o} \tag{5.7}$$

$$T_e = \frac{T_{SW}}{1 + \frac{n_s}{N_c} S} \tag{5.8}$$

The utilisation of our system when accelerating a single function is above 93% in all cases (Figure 5.8). The case where only 2 applications share the accelerators provides the best utilisation overall. This is because context-switch hazards are not present: each application is scheduled on an independent CPU core.

When dynamic rebalancing is not provided, we find a clear separation in utilisation between a system that hosts 2 applications and a system that hosts more (Figure 5.9). This is because context switch hazards are avoided when each application has exclusive use of one of the two CPU cores. In the best case, dynamic rebalancing provides a 9.0% improvement in utilisation. This occurs when accelerator speedup is low. When accelerator speedup is

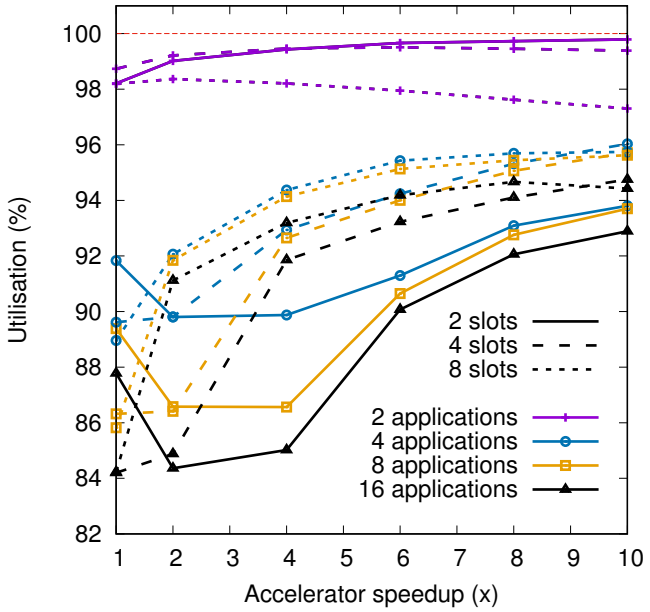


Figure 5.9: System utilisation without dynamic rebalancing when applications call one type of function.

high, functions are more likely to complete within the 10 ms scheduler time slice, thereby avoiding a context switch hazard.

Our static accelerator evaluation shows that our framework performs efficiently when demand is anticipated. Methods for anticipating demand are beyond the scope of this work. We instead turn our focus to our framework's response to the current demand. In a dynamic system, we expect a reduced utilisation since accelerators cannot be used while they are being reconfigured for another function.

5.5.4 Dynamic accelerator cluster

We evaluated the response of our framework to changing application demands by increasing the range of functions that applications execute. A wide range of uniformly distributed, randomly selected functions is unlikely to produce a system that shares accelerators. We therefore limit our system to using just 16 functions – twice the number of reconfigurable slots. The response time depends on the frequency at which our reconfiguration manager

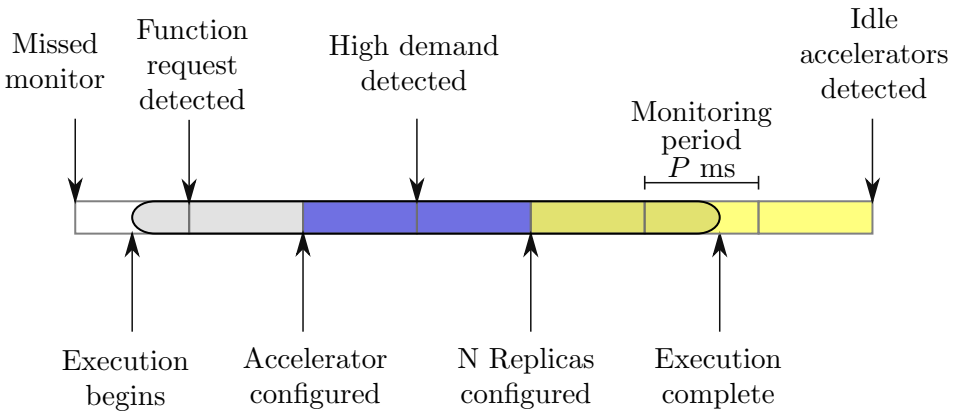


Figure 5.10: Framework response to function request and execution assuming no accelerators have been configured.

monitors demand and the reconfiguration time of accelerators (Figure 5.10). With demand generated at random, we expect it to be difficult to obtain high utilisation. In addition, we expect our system to provide acceleration for functions that already have an accelerator resident, or by reconfiguring accelerators to match the reported demand.

We set n_j to six jobs for our dynamic system experiment. Since the number of resident accelerators are expected to be low for each function the fine-grained partitioning feature of our algorithm is expected to have an impact on performance. Assuming the available 8 slots are shared evenly across the two CPU cores, two jobs can be scheduled sequentially when the first accelerator is configured. The remaining four can then be scheduled in parallel if the monitor reconfigures slots to replicate the accelerator. If slots are not reconfigured, the remaining four will eventually be scheduled and processed sequentially.

Our initial investigation considered a monitoring period of 100 ms. Considering the average area under the curve over the range 0× to 10×, our system was able to reduce the normalised makespan of the application set to 67% (Figure 5.11). However, the acceleration potential of the system is between 5× and 41× per CPU core, depending on the configured speedup of the accelerators. We found an average utilisation of just 15% (Figure 5.12).

The performance of the system improves with the number of applications because the workload is more diverse. With a monitoring period of 100 ms and a scheduling time

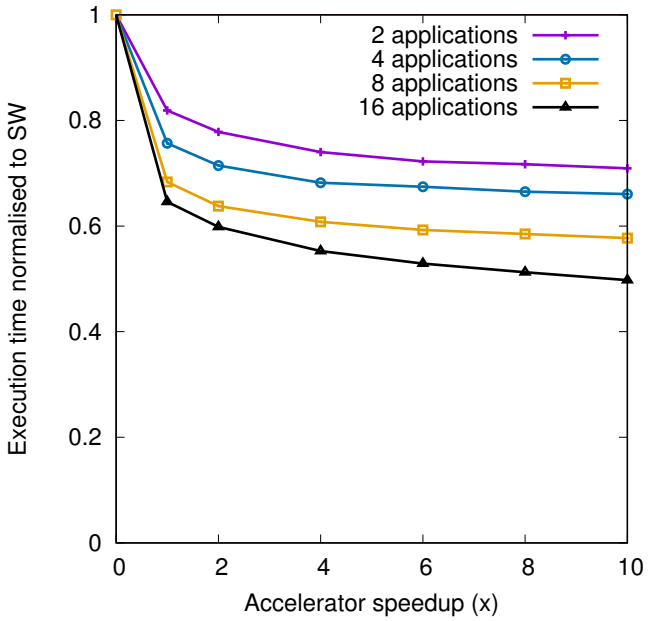


Figure 5.11: Normalised makespan with 8 accelerator slots, 100 ms monitoring period and applications calling 16 types of functions.

slice of 10 ms, all 16 applications will execute across the two cores before accelerators are (re)configured. With each application executing a random function, the likelihood of one of those functions finding a resident accelerator is increased.

The utilisation of the system decreases as accelerator speedup increases because high-throughput accelerators have a greater impact on performance. An idle accelerator is associated with a cost that is proportional to the throughput that it would otherwise provide. Further to this, accelerators finish faster and thereby spend relatively more time idle or being reconfigured.

We further explored the behaviour of our framework by tracing its execution and reconfiguration events. Figure 5.13 shows the execution of two applications across the available 16 functions when 8 accelerators provide a speedup of 10×. For these results, in contrast to all others, we used the same seed to generate the workload of each application. The trace includes the anticipated execution time of each function call if it were executed only in SW. Underlying that trace is a heat map that shows the number of accelerators that are available to each function as the execution progresses.

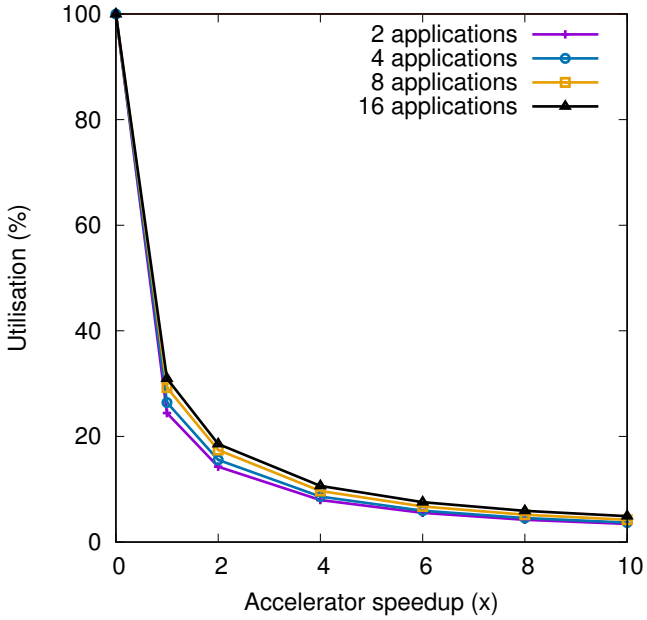


Figure 5.12: System utilisation with 8 accelerator slots, 100 ms monitoring period and applications calling 16 types of functions.

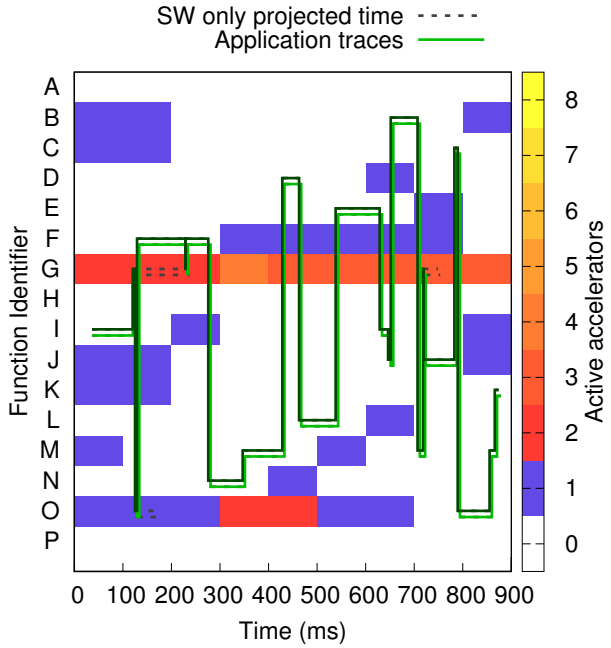


Figure 5.13: Execution and reconfiguration trace of 2 applications with 100 ms monitoring period.

The execution trace shows that accelerators are frequently configured too late to be used by the function calls that they were configured to accelerate. However, function calls still benefit from acceleration at 120 ms, 130 ms and 720 ms, when the accelerator for that function was available at the time the function was called. Further, the delayed reconfiguration results in a diversity of resident accelerators, which increases the probability that at least one accelerator will be available to accelerate a future function call. Nevertheless, many accelerators remain idle, and accelerated functions seldom use the full compute potential of the available slots.

We examined the response of our framework to different monitoring periods by repeating the above experiment. With a monitoring period of 1 ms (Figure 5.14), our framework responds faster to the measured demand and frequently allocates all 8 slots to the function being executed. We see that, although function calls make better use of the available resources, due to workload demand and consequent accelerator replication, the range of resident accelerators becomes limited. If a short-running task for a non-resident accelerator arrives when our monitoring period is 1 ms, it waits an average of 1.5 ms before an accelerator is configured and a further 2 ms before it is replicated.

We reran our makespan and utilisation experiments of Figures 5.11 and 5.12 with a monitoring period of 1 ms to investigate the response in a more dynamic environment. The average normalised makespan of our application set improved by 43% to be 23% (Figure 5.15). The average utilisation increased by 21% to be 35% (Figure 5.16). Although our framework now responds more quickly to changing demands, high-throughput accelerators are difficult to utilise. A typical job in our experiment ($\sim$50 ms) will complete quickly once multiple accelerators are resident (Figure 5.17). Therefore, high-throughput accelerators spend a relatively short time processing the workload when compared to their reconfiguration time.

Figures 5.18 and 5.19 show the performance of systems with different monitoring periods when 8 applications execute concurrently. For our remaining experiment, we use the performance of a system that hosts 8 applications and a 1 ms monitoring period as a baseline for comparison.

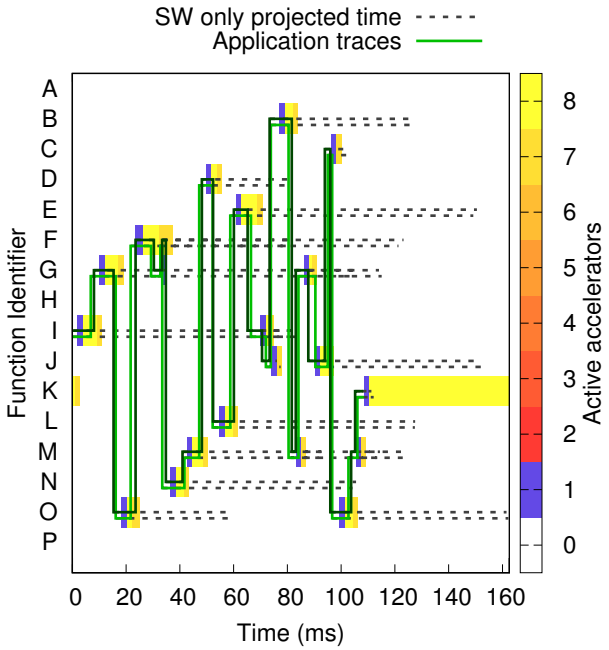


Figure 5.14: Execution and reconfiguration trace of 2 applications with 1 ms monitoring period.

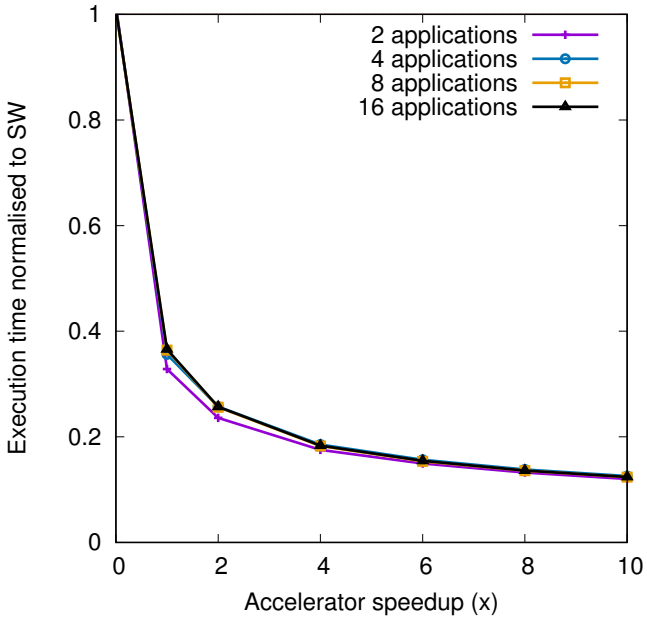


Figure 5.15: Normalised makespan with 8 accelerator slots, 1 ms monitoring period and applications calling 16 types of functions.

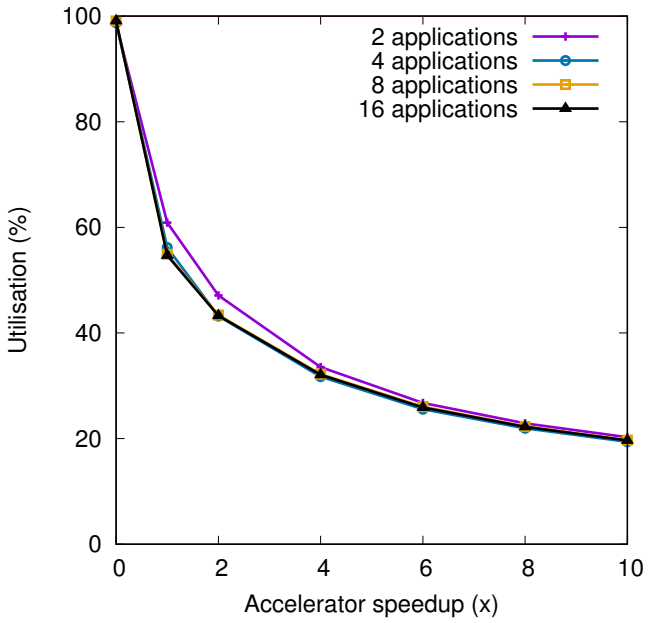


Figure 5.16: System utilisation with 8 accelerator slots, 1 ms monitoring period and applications calling 16 types of functions.

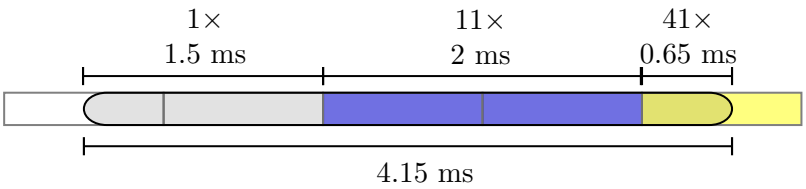


Figure 5.17: Execution time of a typical 50 ms workload using 10× accelerators with monitoring period 1 ms. Refer to Figure 5.10 for an explanation of timeline events.

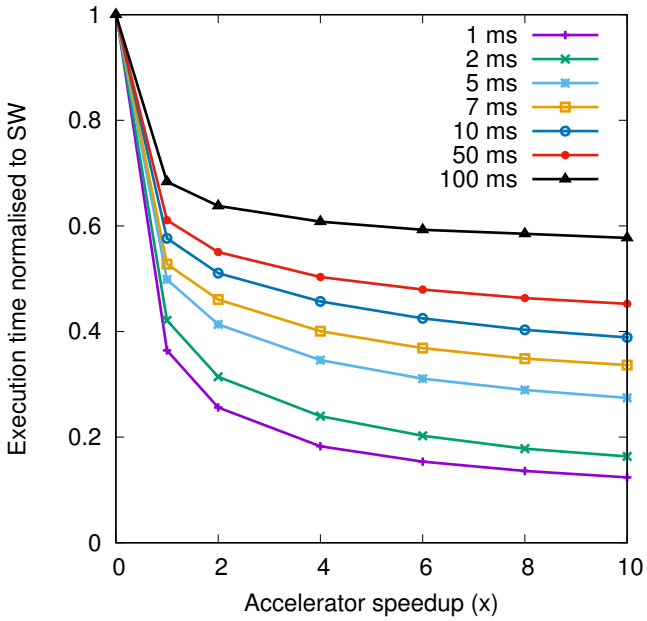


Figure 5.18: Normalised makespan with 8 accelerator slots and 8 applications that call 16 types of functions as the monitoring period is varied.

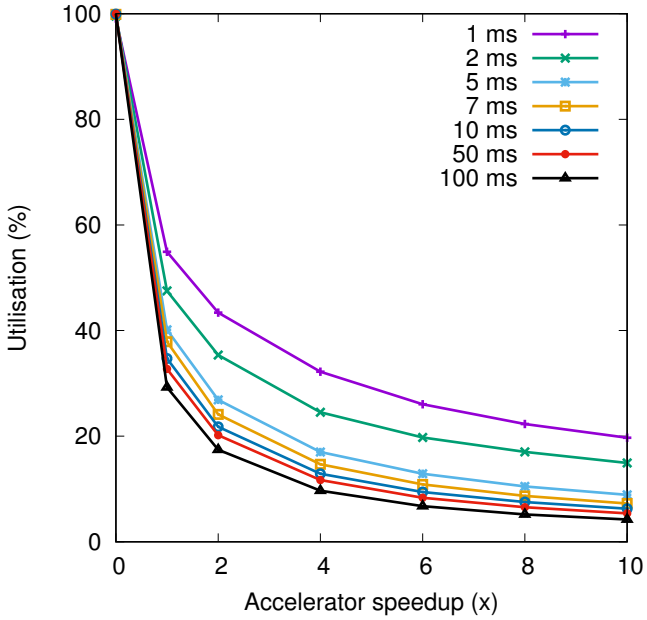


Figure 5.19: System utilisation with 8 accelerator slots and 8 applications that call 16 types of functions as the monitoring period is varied.

Our results show that short-running tasks can be accelerated in a dynamic environment. For the range of workloads that we studied, applications benefit particularly well when the framework responds more quickly to changing demand.

5.5.5 The impact of fine-grained job partitioning

Our final experiment evaluates the benefit of dividing a single HW job into smaller jobs when the application observes a single resident accelerator. We modified the shared library functions to submit only one job for each resident accelerator and repeated our makespan and utilisation experiments. We observed that fine-grained job partitioning improved the average normalised makespan by 7.7% over accelerator speedups in the range 0× to 10× (Figure 5.20). The average utilisation was reduced by 9.0% over the same range (Figure 5.21). On closer inspection, we found that, although our framework responds quickly to demand, applications respond slowly to replication. Jobs were rarely scheduled on accelerator replicas, particularly when the number of applications was 2. This is because the application must first wait for the initial large job to complete and, in the absence of context switch hazards, both SW and HW finish processing the entire workload at almost the same time.

5.6 Chapter summary

In this chapter we have described and evaluated our framework for accelerating general-purpose tasks in a multi-user environment. The framework allows applications to share a set of accelerators to avoid reconfiguration overheads for short-lived tasks. Function demand is monitored and responded to by optimising the available set of accelerators using DPR. When an accelerator is overloaded, a replica may be reconfigured and routed to the job queue that serves that function.

Our results show that the optimum method for transferring a small number of function arguments on the Zynq-7020 is direct CSR access. However, such a design pattern does not

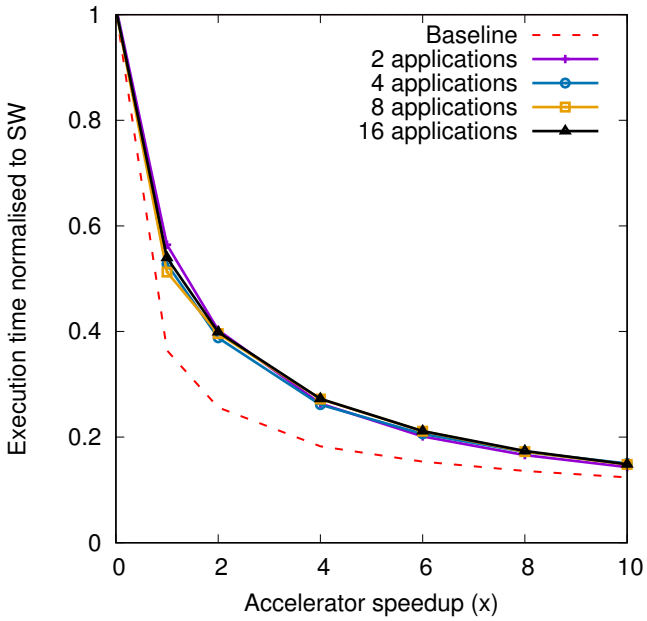


Figure 5.20: Normalised makespan with 8 accelerator slots, 1 ms monitoring period and applications calling 16 types of functions. Only one job is submitted per resident accelerator.

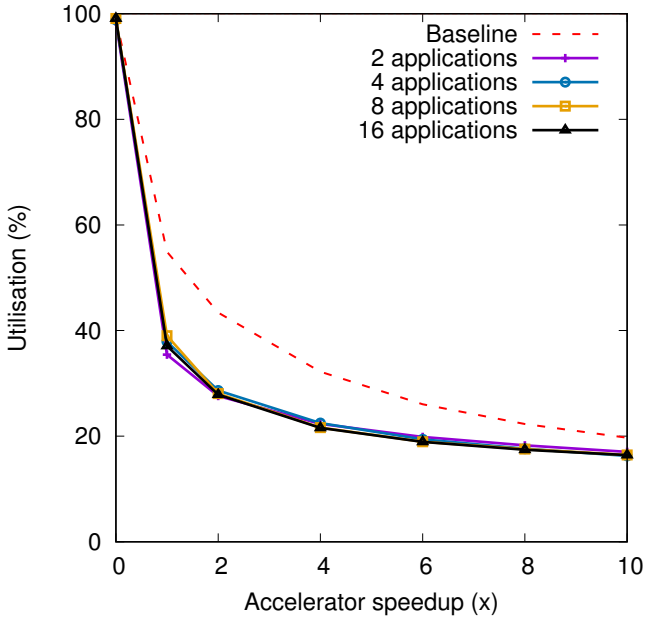


Figure 5.21: System utilisation with 8 accelerator slots, 1 ms monitoring period and applications calling 16 types of functions. Only one job is submitted per resident accelerator.

address the concurrency concerns of sharing accelerators across threads and applications.

We have shown that the alternate approach provided by our framework both solves the concurrency concerns and provides the best performance when five or more 32-bit values are passed between SW and HW and previous translations remains in the TLB.

Our framework reduces the makespan of a random set of application personalities when the demand monitoring period is 1 ms. In that case, applications complete 77% faster than an equivalent SW-only solution. While our results show less improvement for a longer monitoring period (33% faster), the set of resident accelerators is then more diverse. That improves the likelihood that a short-running function will have an accelerator available as soon as it is called.

In Section 5.4.7, we identified context switch hazards as a challenge for HW/SW workload partitioning. Our results show a 9.0% improvement in utilisation when the workload is rebalanced dynamically to compensate for such hazards.

Our results show that dividing a HW partition into many small jobs and simultaneously submitting them to the framework improves performance by 9.0% when accelerator replication is expected.

Our research shows that short-running tasks can be accelerated in multitasking environments if the response time to changing demand is minimised, and also if a diverse set of resident accelerators is maintained.

Chapter 6

Conclusion

This chapter highlights how my research outcomes meet the research aims set out in Chapter 1. It then concludes by proposing directions for future work.

6.1 Applications for tightly-coupled systems

In Chapter 3, I detailed a case study of accelerating the OS SW task scheduler. The seL4 scheduler is heavily optimised for a short execution time, hence we were sceptical of the benefit that HW acceleration would provide. Nevertheless, we were able to reduce execution time by 5.5% and the variance in the execution time by 58%.

While that chapter presents just one concrete application, the study demonstrates that a task that could otherwise not be accelerated, can benefit when a tight-coupling of CPU and FPGA is provided.

6.2 Communication models for engineers

In Chapter 3, I presented an in-depth evaluation of the available communication methods provided by the tightly-coupled Zynq-7020 SoC. That was followed in Chapter 4 with metrics that quantify the latency and required CPU execution cycles for short transfers between CPU and FPGA. With that work, SW engineers can choose the most appropriate method of communication for their application.

In Chapter 4, I presented a model for partitioning a workload between HW and SW in a single-user environment. Given the metrics provided in that chapter and the expected throughput of both CPU and accelerator, I demonstrated that SW engineers can calculate the HW/SW partitioning ratio that leads to a completion time that is within 2% of the empirical optimum. SW engineers can also use the developed model to predict, with 12% MRE, the execution time of workloads of varying size.

In Chapter 5, I extended the model presented in Chapter 4 to include many accelerators and queue wait time. Although that model worked well in the absence of CPU core time sharing, we identified SW context-switch hazards as a limitation of our model. Since we cannot predict when and for how long SW execution will be stopped, we must periodically monitor progress and repartition the work if a context switch is detected.

Along with that work, I provided and evaluated a model for transferring function arguments between SW and HW. That model may be used by SW engineers to select an optimal argument transfer method based on their required transfer size and synchronisation requirements.

6.3 Accelerator sharing for concurrent short-running tasks

Chapter 5 proposed a framework that uses our task partitioning methodology to accelerate many concurrent short-running tasks. With 8 accelerator slots that provide speedups in the range 0× to 10× for perfectly parallel workloads with SW-only execution times in the

range 1 ms to 100 ms, we found that our framework improves overall system performance with an average speedup of:

- 88% when one type of task is accelerated;
- 33% when a dynamic set of tasks are accelerated; and
- 77% when the response time to demand is reduced.

Although the best performance was observed when the response time was quick, a slower response time led to a more diverse set of resident accelerators. The more diverse set of resident accelerators increased the likelihood that an accelerator would immediately be available to the task, thereby avoiding reconfiguration delay.

6.4 Future work

In dynamic, multitasking environments, smaller workloads may benefit from an increased monitoring period. Our experiments have evaluated the response of our framework to changing demands, but smaller workloads will require an unreasonably quick response time to instantiate an accelerator before execution completes in SW. It may be beneficial to maintain the residency of a diverse range of accelerators to improve the likelihood that an accelerator is available for such functions before they are called. Our results show that an increased monitoring period reduces utilisation but increases the diversity of resident accelerators. One avenue for further investigation is to find the optimum monitoring period for different workload sizes, or to find the ideal balance between replication and diversity.

Augmenting the current demand with an anticipated demand may further improve the probability that an accelerator will be available at the time it is needed. We have considered application personalities that are generated using uniformly distributed random numbers. In practice, we expect some shared library functions to be more popular than others. A trivial metric for anticipating demand is the most frequently used function. Others may include function call dependencies, heuristics or machine learning.

Although we have shown an average makespan reduction of 77%, we have also found that system resources are underutilised.

We chose a trivial reconfiguration algorithm (Section 5.4.7) to evaluate our framework. This algorithm could be improved by applying heuristics for deciding whether or not an accelerator should be replaced to serve a new function. A trivial improvement to this algorithm is to consider the historic demand for each accelerator.

Knowledge of function dependencies can be used to reduce idle time by speculatively reconfiguring slots. For example, image decompression may commonly be followed by image resizing for rendering. The accelerator for resizing the image can be instantiated early such that it is ready for use as soon as the resize function it called. Dependencies can be determined on- or offline and controlled by either the kernel module or the shared library. The kernel module can monitor the utilisation of the decompression accelerator and ensure that a resize accelerator will also be configured. The library itself could request the appropriate accelerator for the next processing stage shortly before the current stage completes.

Our partitioning algorithm imposes a maximum CPU execution time before rebalancing the work between SW and HW (Section 5.4.7). Future work might investigate changing the maximum workload to explore the trade-off in computational overhead and resource utilisation. Although the accelerators may spend less time idle, the CPU overhead of using the framework would be reduced, freeing the CPU to process more meaningful work.

If the execution time overhead of our argument transfer approach is significant on other platforms, a future direction of research could be alternate approaches for concurrent, multi-user CSR access. Such work would investigate the optimum level of CSR replication and the synchronisation primitives that should be used to protect them. Of course, the overhead of synchronisation may be higher than the overhead of an OS system call – we may find it best to move CSR access back into the OS.

References

[ARM05] ARM limited. *ARMv7-A Architecture Reference Manual DDI 0406C.b*, 2005.

[ARM11] ARM limited. *AMBA AXI and ACE Protocol Specification IHI 0022D (ID102711)*, 2011.

[ARM12] ARM limited. *ARM Cortex-A9 MPCore Technical Reference Manual DDI 0407H*, 2012.

[ARM13] ARM limited. *ARM Cortex-A53 MPCore Processor Technical Reference Manual DDI 0500D*, 2013.

[Avn19] Avnet. Zedboard. http://zedboard.org/product/zedboard, 2019. [Online; accessed 18-September-2019].

[BBB+07] R. Baxter, S. Booth, M. Bull, G. Cawood, J. Perry, M. Parsons, A. Simpson, A. Trew, A. McCormick, G. Smart, R. Smart, A. Cantle, R. Chamberlain, and G. Genest. Maxwell - a 64 FPGA supercomputer. In *Proceedings of the Second NASA/ESA Conference on Adaptive Hardware and Systems*, AHS '07, pages 287–294. IEEE Computer Society, 2007.

[BCD69] A. Bensoussan, C. Clingen, and R. Daley. The Multics virtual memory. In *Proceedings of the second symposium on Operating systems principles*, pages 30–42, 1969.

[BDH+97] J. Burns, A. Donlin, J. Hogg, S. Singh, and M. De Wit. A dynamic reconfiguration run-time system. In *Proceedings. The 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines Cat. No. 97TB100186)*, pages 66–75. IEEE, 1997.

[Bea16] A. Bean. *Improving memory access performance for irregular algorithms in heterogeneous CPU/FPGA systems.* PhD thesis, Imperial College, London, 2016.

[Bit09] R. Bittner. Bus mastering PCI express in an FPGA. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '09, pages 273–276. Association for Computing Machinery, 2009.

[BNP10] T. Beisel, M. Niekamp, and C. Plessl. Using shared library interposing for transparent application acceleration in systems with heterogeneous hardware accelerators. In *ASAP 2010 - 21st IEEE International Conference on Application-specific Systems, Architectures and Processors*, pages 65–72, July 2010.

[BP06] Z. Baker and V. Prasanna. An architecture for efficient hardware data mining using reconfigurable computing systems. In *2006 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 67–75. IEEE, 2006.

[Bre96] G. Brebner. A virtual hardware operating system for the Xilinx XC6200. In *Proceedings of the 6th International Workshop on Field-Programmable Logic, Smart Applications, New Paradigms and Compilers*, FPL '96, pages 327–336. Springer-Verlag, 1996.

[BRS13] D. Bacon, R. Rabbah, and S. Shukla. FPGA programming for the masses. *Communications of the ACM*, 56(4):56–63, 2013.

[BS13] S. Breßand G. Saake. Why it is time for a HyPE: A hybrid query processing engine for efficient GPU coprocessing in DBMS. *Proceedings of the VLDB Endowment*, 6(12):1398–1403, 2013.

[BSC+11] B. Blackham, Y. Shi, S. Chattopadhyay, A. Roychoudhury, and G. Heiser. Timing analysis of a protected operating system kernel. In *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, pages 339–348, Nov 2011.

[CA07] S. Craven and P. Athanas. Examining the viability of FPGA supercomputing. *EURASIP Journal on Embedded systems*, 2007(1), 2007.

[CCC+16] M. Chang, Y. Chen, J. Cong, P. Huang, C. Kuo, and C. Yu. The SMEM seeding acceleration for DNA sequence alignment. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 32–39, May 2016.

[CCF+16] Y. Choi, J. Cong, Z. Fang, Y. Hao, G. Reinman, and P. Wei. A quantitative analysis on microarchitectures of modern CPU-FPGA platforms. In *Proceedings of the 53rd Annual Design Automation Conference*, DAC '16, pages 109:1–109:6. ACM, 2016.

[CFMRAF17] L. Costas, R. Fernández-Molanes, J. Rodríguez-Andina, and J. Fariña. Characterization of FPGA-master ARM communication delays in Zynq devices. In *2017 IEEE International Conference on Industrial Technology (ICIT)*, pages 942–947, March 2017.

[CH07] Tom Van Court and Martin C. Herbordt. Families of FPGA-based accelerators for approximate string matching. *Microprocessors and microsystems*, 31 2:135–145, 2007.

[CKPP15] G. Charitopoulos, I. Koidis, K. Papadimitriou, and D. Pnevmatikatos. Hardware task scheduling for partially reconfigurable FPGAs. In *International Symposium on Applied Reconfigurable Computing*, pages 487–498. Springer, 2015.

[CLL+96] C. Carreras, J. Lopez, M. Lopez, L. Sanchez, C. Delgado-Kloos, and N. Martinez. A co-design methodology based on formal specification and high-level estimation. In *Proceedings of the 4th International Workshop on Hardware/Software Co-Design*, CODES '96, pages 28–35. IEEE Computer Society, 1996.

[CP16] R. Chen and V. Prasanna. Accelerating equi-join on a CPU-FPGA heterogeneous platform. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 212–219, May 2016.

[CPM97] L. Chunho, M. Potkonjak, and W. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of 30th Annual International Symposium on Microarchitecture*, pages 330–335, Dec 1997.

[CSZ+14] F. Chen, Y. Shan, Y. Zhang, Y. Wang, H. Franke, X. Chang, and K. Wang. Enabling FPGAs in the cloud. In *Proceedings of the 11th ACM Conference on Computing Frontiers*, CF '14. Association for Computing Machinery, 2014.

[DE97] O. Diessel and H. ElGindy. Run-time compaction of FPGA designs. In *International Workshop on Field Programmable Logic and Applications*, pages 131–140. Springer, 1997.

[Den65] Jack B Dennis. Segmentation and the design of multiprogrammed computer systems. *Journal of the ACM (JACM)*, 12(4):589–602, 1965.

[DG12] E. Dodiu and V. Gaitan. Custom designed CPU architecture based on a hardware scheduler and independent pipeline registers – Concept and theory of operation. In *Electro/Information Technology (EIT), 2012 IEEE International Conference on*, pages 1–5, May 2012.

[DMBS12] T. Davidson, M. Merlier, K. Bruneel, and D. Stroobandt. A dynamically reconfigurable pattern matcher for regular expressions on FPGA. In *Parallel Computing with FPGAs 2011 (ParaFPGA 2011)*, volume 22, pages 611–618. IOS Press, 2012.

[DT13] J. Dahlstrom and S. Taylor. Migrating an OS scheduler into tightly coupled FPGA logic to increase attacker workload. In *Military Communications Conference, MILCOM 2013 - 2013 IEEE*, pages 986–991, Nov 2013.

[Egu10] K. Eguro. SIRC: An extensible reconfigurable computing communication API. In *2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 135–138, May 2010.

[FAL+16] H. Fernandes, M. Aslam, J. Lobo, J. Ferreira, and J. Dias. Bayesian inference implemented on FPGA with stochastic bitstreams for an autonomous robot. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, Aug 2016.

[FBCS12] J. Fowers, G. Brown, P. Cooke, and G. Stitt. A performance and energy comparison of FPGAs, GPUs, and multicores for sliding-window applications. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '12, pages 47–56. Association for Computing Machinery, 2012.

[Goo83] J. Goodman. Using cache memory to reduce processor-memory traffic. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, ISCA '83, pages 124–131. Association for Computing Machinery, 1983.

[GRE+01] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, pages 3–14, Dec 2001.

[GSB+00] S. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. Taylor. PipeRench: A reconfigurable architecture and compiler. *Computer*, 33(4):70–77, Apr 2000.

[GWC+14] J. Gong, T. Wang, J. Chen, H. Wu, F. Ye, S. Lu, and J. Cong. An efficient and flexible host-FPGA PCIe communication library. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–6, Sept 2014.

[HTK15] M. Happe, A. Traber, and A. Keller. Preemptive hardware multitasking in ReconOS. In *International Symposium on Applied Reconfigurable Computing*, pages 79–90. Springer, 2015.

[HW97] J. Hauser and J. Wawrzynek. Garp: A MIPS processor with a reconfigurable coprocessor. In *Proceedings. The 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines Cat. No.97TB100186)*, pages 12–21, April 1997.

[IBM14] IBM. *Coherent Accelerator Processor Interface User's Manual Version 1.1*, 2014.

[IK07] A. Ioannou and M. Katevenis. Pipelined heap (priority queue) management for advanced scheduling in high-speed networks. *IEEE/ACM Transactions on Networking*, 15(2):450–461, 2007.

[Int16] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3 (3A, 3B, 3C & 3D): System Programming Guide 325384-060US*, 2016.

[Int19a] Intel. *Arria V Device Handbook Volume 1: Device Interfaces and Integration*, 2019.

[Int19b] Intel. *Intel Acceleration Stack for Intel Xeon CPU with FPGAs Core Cache Interface (CCI-P) Reference Manual MNL-1092*, 2019.

[Int19c] Intel Corporation. OPAE. https://01.org/opae, 2019. [Online; accessed 18-September-2019].

[ISA16] Z. István, D. Sidler, and G. Alonso. Runtime parameterizable regular expression operators for databases. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 204–211, May 2016.

[JHE+13] K. Jozwik, S. Honda, M. Edahiro, H. Tomiyama, and H. Takada. Rainbow: An operating system for software-hardware multitasking on dynamically partially reconfigurable FPGAs. *International Journal of Reconfigurable Computing*, 2013, 2013.

[JK13] M. Jacobsen and R. Kastner. RIFFA 2.0: A reusable integration framework for FPGA accelerators. In *2013 23rd International Conference on Field programmable Logic and Applications*, pages 1–8, Sept 2013.

[JSNV13] W. José, A. Silva, H. Neto, and M. Véstias. Analysis of matrix multiplication on high density Virtex-7 FPGA. In *2013 23rd International Conference on Field programmable Logic and Applications*, pages 1–4. IEEE, 2013.

[KAA+17] K. Kara, D. Alistarh, G. Alonso, O. Mutlu, and C. Zhang. FPGA-accelerated dense linear machine learning: A precision-convergence trade-off. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 160–167, April 2017.

[KAE+14] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems*, 32(1):2:1–2:70, feb 2014.

[KBT10] D. Koch, C. Beckhoff, and J. Torresen. Zero logic overhead integration of partially reconfigurable modules. In *Proceedings of the 23rd Symposium on Integrated Circuits and System Design*, SBCCI '10, pages 103–108. Association for Computing Machinery, 2010.

[KD15] A. Kroh. and O. Diessel. Towards OS kernel acceleration in heterogeneous systems. *First International Workshop on Heterogeneous High-performance Reconfigurable Computing (H^2RC)*, 2015.

[KD18] A. Kroh and O. Diessel. A short-transfer model for tightly-coupled CPU-FPGA platforms. In *2018 International Conference on Field-Programmable Technology (FPT)*, pages 366–369, Dec 2018.

[KD19] A. Kroh and O. Diessel. Efficient fine-grained processor-logic interactions on the cache-coherent Zynq platform. *ACM Trans. Reconfigurable Technol. Syst.*, 11(4):25:1–25:22, January 2019.

[Khr19] Khronos Group. OpenCL. https://www.khronos.org/opencl, 2019. [Online; accessed 18-September-2019].

[KHT07] D. Koch, C. Haubelt, and J. Teich. Efficient hardware checkpointing: Concepts, overhead analysis, and implementation. In *Proceedings of the 2007 ACM/SIGDA 15th International Symposium on Field Programmable Gate Arrays*, FPGA '07, pages 188–196. Association for Computing Machinery, 2007.

[KL08] J. Kelm and S. Lumetta. HybridOS: Runtime support for reconfigurable accelerators. In *Proceedings of the 16th International ACM/SIGDA Symposium on Field Programmable Gate Arrays*, FPGA '08, pages 212–221. ACM, 2008.

[KP05] H. Kalte and M. Porrmann. Context saving and restoring for multitasking in reconfigurable systems. In *International Conference on Field Programmable Logic and Applications, 2005.*, pages 223–228, Aug 2005.

[KPPK11] T. Kenter, C. Plessl, M. Platzner, and M. Kauschke. Performance estimation framework for automated exploration of CPU-accelerator architectures. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, pages 177–180. Association for Computing Machinery, 2011.

[KSM03] P. Kuacharoen, M. Shalan, and V. Mooney. A configurable hardware scheduler for real-time systems. In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms*, pages 96–101. CSREA Press, 2003.

[LBK+16] A. Lösch, T. Beisel, T. Kenter, C. Plessl, and M. Platzner. Performance-centric scheduling with task migration for a heterogeneous compute node in the data center. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 912–917, March 2016.

[LHLT10] T. Lee, C. Hu, L. Lai, and C. Tsai. Hardware context-switch methodology for dynamically partially reconfigurable systems. *Journal of Information Science and Engineering*, 26(4):1289–1305, 2010.

[LMAH18] A. Lyons, K. McLeod, H. Almatary, and G. Heiser. Scheduling-context capabilities: A principled, light-weight operating-system mechanism for managing time. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–16, 2018.

[LP07] E. Lubbers and M. Platzner. ReconOS: An RTOS supporting hard- and software threads. In *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pages 441–446, Aug 2007.

[LP09a] E. Lubbers and M. Platzner. Cooperative multithreading in dynamically reconfigurable systems. In *2009 International Conference on Field Programmable Logic and Applications*, pages 551–554, Aug 2009.

[LP09b] E. Lübbers and M. Platzner. ReconOS: Multithreaded programming for reconfigurable computers. *ACM Transactions on Embedded Computing Systems (TECS)*, 9(1):1–33, 2009.

[LRL+12] Y. Liang, K. Rupnow, Y. Li, D. Min, M. Do., and D. Chen. High-level synthesis: Productivity, performance, and software constraints. *Journal of Electrical and Computer Engineering*, 2012:1–14, 2012.

[LSV05] B. Lai, P. Schaumont, and I. Verbauwhede. A light-weight cooperative multithreading with hardware supported thread-management on an embedded multi-processor system. In *Conference Record of the Thirty-Ninth Asilomar Conference onSignals, Systems and Computers, 2005.*, pages 1647–1651, October 2005.

[LVL03] M. López-Vallejo and J. López. On the hardware-software partitioning problem: System modeling and partitioning techniques. *ACM Trans. Des. Autom. Electron. Syst.*, 8(3):269–297, July 2003.

[MB02] V. Mooney. and D. Blough. A hardware-software real-time operating system framework for SoCs. *Design Test of Computers, IEEE*, 19(6):44–51, Nov 2002.

[MC15] V. Mirian and P. Chow. Evaluating shared virtual memory in an OpenCL framework for embedded systems on FPGAs. In *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–8, Dec 2015.

[MG13] A. Morales-Villanueva and A. Gordon-Ross. On-chip context save and restore of hardware tasks on partially reconfigurable FPGAs. In *2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 61–64, April 2013.

[MLC14] H. Muhuan, K. Lim, and J. Cong. A scalable, high-performance customized priority queue. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4. IEEE, 2014.

[MNM+04] T. Marescaux, V. Nollet, J. Mignolet, A. Bartic, W. Moffat, P. Avasare, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins. Run-time support for heterogeneous multitasking on reconfigurable SoCs. *Integration*, 38(1):107–130, 2004.

[MRAF18] R. Molanes, J. Rodríguez-Andina, and J. Fariña. Performance characterization and design guidelines for efficient processor-FPGA communication in Cyclone V FPSoCs. *IEEE Transactions on Industrial Electronics*, 65(5):4368–4377, May 2018.

[MSFRA15] R. Molanes, F. Salgado, J. Fariña, and J. Rodríguez-Andina. Characterization of FPGA-master ARM communication delays in Cyclone V devices. In *IECON 2015-41st Annual Conference of the IEEE Industrial Electronics Society*, pages 004229–004234. IEEE, 2015.

[NRL07] A. Nácul, F. Regazzoni, and M. Lajolo. Hardware scheduling support in SMP architectures. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '07, pages 642–647. EDA Consortium, 2007.

[OK17] J. Ordaz and D. Koch. Making a case for an ARM Cortex-A9 CPU interlay replacing the NEON SIMD unit. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, Sep. 2017.

[OLAH13] S. Ong, S. Lee, N. Ali, and F. Hussin. SEOS: Hardware implementation of real-time operating system for adaptability. In *Computing and Networking (CANDAR), 2013 First International Symposium on*, pages 612–616, Dec 2013.

[PAA+06] W. Peck, E. Anderson, J. Agron, J. Stevens, F. Baijot, and D. Andrews. Hthreads: A computational model for reconfigurable devices. In *2006 International Conference on Field Programmable Logic and Applications*, pages 1–4, Aug 2006.

[PCC+14] A. Putnam, A. Caulfield, E. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Xiao, and D. Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 13–24, June 2014.

[PH90] D. Patterson and J. Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 1990.

[PS14] T. Preußer and R. Spallek. Ready PCIe data streaming solutions for FPGAs. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, Sep. 2014.

[PS15] A. Powell and D. Silage. Statistical performance of the ARM Cortex A9 accelerator coherency port in the Xilinx Zynq SoC for real-time applications. In *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–6, Dec 2015.

[PWP+03] K. Puttegowda, W. Worek, N. Pappas, A. Dandapani, P. Athanas, and A. Dickerman. A run-time reconfigurable system for gene-sequence searching. In *16th International Conference on VLSI Design, 2003. Proceedings.*, pages 561–566, Jan 2003.

[RJ16] S. Ravi and M. Joseph. Open source HLS tools: A stepping stone for modern electronic CAD. In *2016 IEEE International Conference on Computational Intelligence and Computing Research (ICCIC)*, pages 1–8, Dec 2016.

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

[SFJ+19] R. Skhiri, V. Fresse, J. Jamont, B. Suffran, and J. Malek. From FPGA to support cloud to cloud of FPGA: State of the art. *International Journal of Reconfigurable Computing*, 2019:1–17, 12 2019.

[SIOA17] D. Sidler, Z. István, M. Owaida, and G. Alonso. Accelerating pattern matching queries in hybrid CPU-FPGA architectures. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 403–415. ACM, 2017.

[SKH16] T. Sewell, F. Kam, and G. Heiser. Complete, high-assurance determination of loop bounds and infeasible paths for WCET analysis. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–11, April 2016.

[SLM00] H. Simmler, L. Levinson, and R. Männer. Multitasking on FPGA coprocessors. In *International Workshop on Field Programmable Logic and Applications*, pages 121–130. Springer, 2000.

[SMT+12] B. Sukhwani, H. Min, M. Thoennes, P. Dube, B. Iyer, B. Brezzo, D. Dillenberger, and S. Asaad. Database analytics acceleration using FPGAs. In *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 411–420, Sep. 2012.

[So07] H. So. *BORPH: An Operating System for FPGA-Based Reconfigurable Computers*. PhD thesis, EECS Department, University of California, Berkeley, Jul 2007.

[ŠRS12] P. Škoda, B. Rogina, and V. Sruk. Fpga implementations of data mining algorithms. In *2012 Proceedings of the 35th International Convention MIPRO*, pages 362–367. IEEE, 2012.

[SSS15] J. Silva, V. Sklyarov, and I. Skliarova. Comparison of on-chip communications in Zynq-7000 all programmable systems-on-chip. *IEEE Embedded Systems Letters*, 7(1):31–34, March 2015.

[SWP04] C. Steiger, H. Walder, and M. Platzner. Operating systems for reconfigurable embedded platforms: Online scheduling of real-time tasks. *IEEE Transactions on Computers*, 53(11):1393–1407, Nov 2004.

[SWWB13] M. Sadri, C. Weis, N. Wehn, and L. Benini. Energy and performance exploration of accelerator coherency port using Xilinx ZYNQ. In *Proceedings*

of the 10th FPGAworld Conference, FPGAworld '13, pages 5:1–5:8. ACM, 2013.

[Tei12] J. Teich. Hardware/software codesign: The past, the present, and predicting the future. *Proceedings of the IEEE*, 100(Special Centennial Issue):1411–1430, May 2012.

[THM15] J. Tang, Y. Hau, and M. Marsono. Hardware/software partitioning of embedded system-on-chip applications. In *2015 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, pages 331–336, Oct 2015.

[VB13] W. Vanderbauwhede and K. Benkrid. *High-performance computing using FPGAs*, volume 3. Springer, 2013.

[VF18] K. Vipin and S. Fahmy. FPGA dynamic and partial reconfiguration: A survey of architectures, methods, and applications. *ACM Comput. Surv.*, 51(4), July 2018.

[VKVF16] M. Vesper, D. Koch, K. Vipin, and S. A. Fahmy. JetStream: An open-source high-performance PCI Express 3 streaming library for FPGA-to-host and FPGA-to-FPGA communication. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–9, Aug 2016.

[VMB19] P. Vogel, A. Marongiu, and L. Benini. Exploring shared virtual memory for FPGA accelerators with a configurable IOMMU. *IEEE Transactions on Computers*, 68(4):510–525, April 2019.

[VPI05] M. Vuletid, L. Pozzi, and P. Ienne. Seamless hardware-software integration in reconfigurable computing systems. *Design Test of Computers, IEEE*, 22(2):102–113, March 2005.

[VPK19] A. Vaishnav, K. Pham, and D. Koch. Heterogeneous resource-elastic scheduling for CPU+FPGA architectures. In *Proceedings of the 10th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies*, HEART 2019, pages 1:1–1:6. ACM, 2019.

[VPKG18] A. Vaishnav, K. D. Pham, D. Koch, and J. Garside. Resource elastic virtualization for FPGAs using OpenCL. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 111–1117, Aug 2018.

[VRKP14] G. Vaz, H. Riebler, T. Kenter, and C. Plessl. Deferring accelerator offloading decisions to application runtime. In *2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14)*, pages 1–8, Dec 2014.

[VSL08] F. Vahid, G. Stitt, and R. Lysecky. Warp processing: Dynamic translation of binaries to FPGA circuits. *Computer*, 41(7):40–46, July 2008.

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

[WGY+17] C. Wang, L. Gong, Q. Yu, X. Li, Y. Xie, and X. Zhou. DLAU: A scalable deep learning accelerator unit on FPGA. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(3):513–517, March 2017.

[WH95] M. Wirthlin and B. Hutchings. A dynamic instruction set computer. In *Proceedings IEEE Symposium on FPGAs for Custom Computing Machines*, pages 99–107. IEEE, 1995.

[WMW+16] G. Weisz, J. Melber, Y. Wang, K. Fleming, E. Nurvitadhi, and J. Hoe. A study of pointer-chasing performance on shared-memory processor-FPGA systems. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '16, pages 264–273. ACM, 2016.

[WP02] H. Walder and M. Platzner. Non-preemptive multitasking on FPGAs: Task placement and footprint transform. In *Proceedings of the 2nd International Conference on Engineering of Reconfigurable Systems and Architectures (ERSA)*, pages 24–30. CSREA Press, 2002.

[WPAH16] J. Weerasinghe, R. Polig, F. Abel, and C. Hagleitner. Network-attached FPGAs for data center applications. In *2016 International Conference on Field-Programmable Technology (FPT)*, pages 36–43, Dec 2016.

[WPC+16] Z. Wang, J. Paul, H. Y. Cheah, B. He, and W. Zhang. Relational query processing on OpenCL-based FPGAs. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–10, Aug 2016.

[WSC10] J. Wu, T. Srikanthan, and G. Chen. Algorithmic aspects of hardware/software partitioning: 1D search algorithms. *IEEE Transactions on Computers*, 59(4):532–544, April 2010.

[WUZY19] R. Watanabe, S. Ura, Q. Zhao, and T. Yoshida. Implementation of FPGA building platform as a cloud service. In *Proceedings of the 10th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies*, pages 1–6, 2019.

[WWLS13] J. Wu, P. Wang, S. Lam, and T. Srikanthan. Efficient heuristic and tabu search for hardware/software partitioning. *The Journal of Supercomputing*, 66(1):118–134, 2013.

[WXH+16] Y. Wang, J. Xu, Y. Han, H. Li, and X. Li. DeepBurning: Automatic generation of FPGA-based learning accelerators for the neural network family. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2016.

[Xil] Xilinx. *Zynq-7000 SoC (Z-7007S, Z-7012S, Z-7014S, Z-7010, Z-7015, and Z-7020): DC and AC Switching Characteristics DS187 (v1.20.1).*

[Xil14] Xilinx. *Zynq-7000 All Programmable SoC Technical Reference Manual UG585 (v1.9.1)*, 2014.

[Xil19a] Xilinx. *Zynq Ultrascale+ Device Technical Reference Manual UG1085 (v2.1)*, 2019.

[Xil19b] Xillybus. Xillinux. http://xillybus.com/xillinux, 2019. [Online; accessed 18-September-2019].

[Xil20a] Xilinx. *Versal Architecture and Product Data Sheet: Overview DS950*, 2020.

[Xil20b] Xilinx. *Zynq UltraScale+ MPSoC Data Sheet: DC and AC Switching Characteristics DS925 (v1.17)*, 2020.

[XPN16] T. Xia, J. Prévotet, and F. Nouvel. Hypervisor mechanisms to manage FPGA reconfigurable accelerators. In *2016 International Conference on Field-Programmable Technology (FPT)*, pages 44–52, Dec 2016.

[YMHB00] Z. Ye, A. Moshovos, S. Hauck, and P. Banerjee. CHIMAERA: A high-performance architecture with a tightly-coupled reconfigurable functional unit. *SIGARCH Comput. Archit. News*, 28(2):225–235, May 2000.

[YOY17] M. Yoshimi, Y. Oge, and T. Yoshinaga. Pipelined parallel join and its FPGA-based acceleration. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 10(4):1–28, 2017.

[YRS05] P. Yiannacouras, J. Rose, and J. Steffan. The microarchitecture of FPGA-based soft processors. In *Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '05, pages 202–212. ACM, 2005.

[ZZJB13] Y. Zhang, F. Zhang, Z. Jin, and J. Bakos. An FPGA-based accelerator for frequent itemset mining. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 6(1):1–17, 2013.

Appendix A

Table A.1: Zynq-7020 CPU overheads, measured in CPU cycles, for short SM writes followed by a DSB to ensure completion and SEV signalling.

Target	Words = 1	2	3	4	5	6	7	8
In-order execution								
L1 Cache	24	24	25	26	27	28	29	30
L2 Cache	36	33	35	35	37	37	39	39
OCM (DE)	34	38	40	44	46	50	52	56
OCM (SO)	43	62	85	104	127	146	169	188
RAM (DE)	31	34	37	40	43	47	58	61
RAM (SO)	63	102	145	184	225	264	305	344
FPGA (DE)	31	34	37	40	43	46	91	98
FPGA (SO)	97	179	256	326	409	485	565	641
Out-of-order execution								
L1 Cache	24	24	25	26	27	28	29	30
L2 Cache	36	33	35	35	37	37	39	39
OCM (DE)	33	37	39	43	45	49	51	55
OCM (SO)	42	64	84	106	126	148	168	190
RAM (DE)	31	34	37	40	43	47	58	62
RAM (SO)	63	104	144	185	224	265	304	345
FPGA (DE)	31	34	37	40	43	46	96	99
FPGA (SO)	96	175	249	331	408	490	570	643

Table A.2: Zynq-7020 CPU overheads, measured in CPU cycles, for short reads from various targets.

Target	Words = 1	2	3	4	5	6	7	8
In-order execution								
L1 Cache	7	8	9	10	11	12	13	14
L2 Cache	28	41	54	67	80	93	106	119
OCM (DE)	25	29	31	35	46	50	52	56
OCM (SO)	25	47	67	89	109	131	151	173
RAM (DE)	73	78	85	91	142	149	156	159
RAM (SO)	73	139	210	281	343	416	485	556
FPGA MMIO (DE)	76	86	103	119	145	164	178	188
FPGA MMIO (SO)	76	146	205	278	343	413	478	542
Out-of-order execution								
L1 Cache	1	1	2	2	3	3	4	5
L2 Cache	12	25	38	51	64	77	90	103
OCM (DE)	9	13	15	19	30	34	36	40
OCM (SO)	9	31	51	73	93	115	135	157
RAM (DE)	57	63	69	73	126	130	139	145
RAM (SO)	57	125	194	260	327	400	462	540
FPGA MMIO (DE)	60	70	87	103	129	148	158	172
FPGA MMIO (SO)	60	130	189	262	327	397	461	526

Table A.3: Zynq-7020 CPU overheads, measured in CPU cycles, for short writes to various targets.

Target	Words = 1	2	3	4	5	6	7	8
In-order execution								
L1 Cache	7	7	8	9	10	11	12	13
L2 Cache	19	16	18	18	20	20	22	22
OCM (DE)	16	20	22	26	28	32	34	38
OCM (SO)	25	47	67	89	109	131	151	173
RAM (DE)	14	17	20	23	26	30	40	45
RAM (SO)	45	86	125	166	205	246	285	326
FPGA (DE)	14	17	20	23	26	29	79	86
FPGA (SO)	85	164	241	317	385	476	565	626
Out-of-order execution								
L1 Cache	1	1	2	2	3	3	4	5
L2 Cache	14	1	2	2	4	4	6	6
OCM (DE)	2	4	6	10	12	16	18	22
OCM (SO)	9	31	51	73	93	115	135	157
RAM (DE)	1	1	4	7	10	14	24	29
RAM (SO)	29	70	109	150	189	230	269	310
FPGA (DE)	1	1	4	7	10	13	63	70
FPGA (SO)	69	145	225	298	369	468	549	610