

EFFICIENT SUPPORT FOR  
PARTIALLY RECONFIGURABLE  
ACCELERATORS IN AN FPGA SoC FOR  
THE GNU/LINUX OPERATING SYSTEM

BY

IOANNIS GALANOMMATIS

SUPERVISOR    PROF. D. PNEVMATIKATOS  
COMMITTEE    ASSOC. PROF. V. SAMOLADAS  
                  PROF. A. DOLLAS

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

OF

MASTER OF SCIENCE

IN

ELECTRICAL ENGINEERING

TECHNICAL UNIVERSITY OF CRETE

JUNE, 2018



2018, IOANNIS GALANOMMATIS

Licensed under the Creative Commons BY-SA 4.0 license.

You may modify and redistribute this work in any medium for any purpose, provided you give credit to the author and retain the same license. Full license text is found at:

<https://creativecommons.org/licenses/by-sa/4.0/legalcode>

The source code repository for this work is publicly available at:

 <https://github.com/igalanommatis/zdma>

# Efficient Support for Partially Reconfigurable Accelerators in FPGA SoC for the GNU/Linux Operating System

## ABSTRACT

Currently there is a rising trend in accelerating specialized computation, mostly driven by the growing need for machine learning. The solutions can be classified to two groups: novel processor architectures, tailored for the targeted computation problem, and hardware acceleration, which is represented mostly by the FPGAs due to their capability to adapt or re-purposed to a different environment.

Partial reconfiguration further extends the FPGA flexibility by allowing the reconfiguration of a part of the device during run-time without interrupting the overall system operation. This technology makes possible to create a system that offers multiple accelerator cores that can be reconfigured on-demand during normal system operation.

This work implements such a system using the Zynq SoC from Xilinx. It consists of the following parts:

- Hardware implementation of one homogeneous low-latency and one heterogeneous high-throughput accelerator system for Zynq-7000 SoC, as well as one homogeneous and balanced system for an UltraScale+ SoC.
- A Linux device driver supporting any system design under its specifications.
- A system library that provides a user-friendly API for managing the accelerators.
- An application in image processing that implements some common accelerators.

In this modular system, each component is isolated and provides an abstracted interface to the others. At the user level, the system provides a simple API that hides all hardware details. Using this API, the user can request a computation from the system which will be scheduled for execution when a hardware resource is available. The system administrator may add, remove or restrict the accelerator availability to system slots or may configure the behavior the scheduler and modify security policies, all without interrupting the normal operation of the system.

Finally, the system takes advantage of the system interconnect to provide concurrent data transfers and parallel access to memory, maximizing total I/O throughput. In order to permit the exploration for the optimal design, we made the system flexible enough to accept any accelerator arrangement and any memory and interconnect layout without any software modification.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Motivation . . . . .	2
1.2	Our Approach . . . . .	3
1.3	Contributions . . . . .	4
<b>2</b>	<b>Related Work</b>	<b>7</b>
<b>3</b>	<b>Background</b>	<b>13</b>
3.1	The Hardware Platform . . . . .	13
3.2	The Communication Protocol . . . . .	15
3.2.1	The AMBA AXI Family . . . . .	16
3.2.2	The AXI Implementation . . . . .	18
3.3	The Physical Interconnect . . . . .	21
3.3.1	The Zynq 7000 Interconnect Architecture . . . . .	22
3.3.2	The Zynq UltraScale+ Interconnect Architecture . . . . .	25
3.4	Exchanging Data with the Programmable Logic . . . . .	28
3.4.1	Programmed I/O from a Processor . . . . .	28
3.4.2	Using the “hard” DMA controller in the PS . . . . .	29
3.4.3	Implementing a DMA controller in the PL . . . . .	30
3.5	Design Components . . . . .	34
3.5.1	The DMA controller . . . . .	34
3.5.2	The Interconnect . . . . .	36
3.6	Partial Reconfiguration . . . . .	43
3.6.1	The Partial Reconfiguration Workflow . . . . .	46
3.6.2	Floorplanning . . . . .	48

<b>4</b>	<b>Hardware Architecture</b>	<b>51</b>
4.1	The Implemented Designs . . . . .	51
4.1.1	An Accelerator Performance Oriented Approach . . . . .	52
4.1.2	An Accelerator Count Oriented Approach . . . . .	52
4.1.3	The Zynq UltraScale+ Port . . . . .	55
4.2	Enabling Partial Reconfiguration . . . . .	57
4.2.1	Challenges . . . . .	57
4.2.2	Implementation . . . . .	59
4.2.3	Partition Sizing . . . . .	61
4.2.4	Partition Heterogeneity . . . . .	64
4.2.5	Decoupling the Reconfigurable Logic . . . . .	65
4.3	Accelerator Configuration . . . . .	65
4.4	System Debugging . . . . .	67
4.5	Describing the Hardware with a Device Tree . . . . .	67
4.5.1	Writing a Device Tree for the System . . . . .	69
4.5.2	Lying about the AXI DMA Interrupt Lines . . . . .	72
<b>5</b>	<b>Software Framework</b>	<b>75</b>
5.1	System Initialization . . . . .	75
5.2	The System Library . . . . .	76
5.2.1	The System-Wide API . . . . .	77
5.2.2	The Task-Specific API . . . . .	78
5.3	Communicating with the Hardware . . . . .	80
5.4	Performing DMA from the kernel . . . . .	81
5.4.1	Allocating DMA'able Memory . . . . .	82
5.4.2	Controller and Channel Selection . . . . .	84
5.4.3	Termination of a DMA Transaction . . . . .	85
5.5	Zero-Copy Transfers . . . . .	86
5.6	Security and Error Handling . . . . .	87
5.7	Configuring the Accelerators . . . . .	88
5.8	The Memory Allocator . . . . .	89
5.9	The Scheduler . . . . .	95
5.10	Partial Reconfiguration . . . . .	99

5.10.1	Using the devc fg Interface . . . . .	100
<b>6</b>	<b>Application and Evaluation</b>	<b>103</b>
6.1	Accelerator Description . . . . .	103
6.1.1	Trivial Pixel Transformations . . . . .	105
6.1.2	Contrast and Brightness Transformations . . . . .	107
6.1.3	The Sharpen, Emboss and Outline Filters . . . . .	107
6.1.4	The Sobel/Scharr Filter . . . . .	108
6.1.5	The Gaussian Blur Filter . . . . .	109
6.1.6	Resource Utilization and Latency . . . . .	110
6.2	Accelerator Interface . . . . .	112
6.3	Evaluation . . . . .	116
<b>7</b>	<b>Conclusion and Future Work</b>	<b>123</b>
7.1	Challenges and Lessons Learned . . . . .	124
7.1.1	The Implementation Workflow . . . . .	124
7.1.2	The Tool Quality . . . . .	125
7.1.3	The Efficiency of HLS . . . . .	126
7.2	Future Work . . . . .	127
7.2.1	Integration with FPGA Manager . . . . .	127
7.2.2	Use of Advanced DMA Modes . . . . .	128
7.2.3	Rethinking the Accelerator - DMAC Relationship . . . . .	129
7.2.4	Task Buffer Migration . . . . .	130
7.2.5	Accelerator Control . . . . .	132
7.2.6	Accelerator Interrupts . . . . .	132
7.2.7	Portability . . . . .	133
7.2.8	Scheduler Improvements . . . . .	133
7.2.9	Bitstream Size . . . . .	134
7.2.10	Clock Management . . . . .	135
7.2.11	Extending Heterogeneity . . . . .	135
7.2.12	Random-Access Model . . . . .	135
7.2.13	Scaling to Multiple Boards . . . . .	136
	<b>Appendices</b>	<b>137</b>

<b>A</b>	<b>Partial Reconfiguration Scripts</b>	<b>139</b>
A.1	Creating static design Device Checkpoint . . . . .	139
A.2	Generating the Project Files . . . . .	140
A.3	TCL Client Script . . . . .	141
A.4	Partial Bitstream Manipulation . . . . .	145
<b>B</b>	<b>HLS Compiler Scripts</b>	<b>147</b>
B.1	Generating and Exporting an Acccelerator Module . . . . .	147
	<b>Glossary</b>	<b>149</b>
	<b>Bibliography</b>	<b>161</b>

# Chapter 1

## Introduction

There is little a modern processor cannot do. They are powerful machines, and this power comes at low prices. Indeed, the abundance of processing power made the IoT revolution possible, where a quite capable processing core may be embedded in the simplest and cheapest everyday devices.

So, if processing power is abundant, why do we often need to resort to specialized computation machines? A general rule is that a general-purpose computing machine would be less efficient in performing a task than a highly specialized machine that was tailor-made for the execution for this specific task. In most cases we can tolerate this inefficiency and this is why general-purpose computers do exist. There are other cases however, where we are prepared to put a lot of effort in order to possess the most efficient computer possible.

Before we become more specific with the FPGAs, let us first define our areas of interest regarding efficiency:

- **Energy:** A general purpose processor utilizes many structural units to coordinate a computation and typically requires more support logic at system level. All this additional logic consumes energy. Even worse, the additional generated heat must be dissipated, and therefore we must spend additional energy to cool the computer.

All of these are less insignificant on a personal computer but they do matter in battery-powered portable devices. It is an equally important matter in scientific

computation, where the cost of energy consumption is usually more important than the cost of the machine itself.

- **Performance:** A general-purpose has its silicon architected for offering optimal overall performance. A specific computation could be benefited more by some other design decisions or it may not benefit at all from the majority of the silicon of a general-purpose computer. In contrast, a highly specialized computer has defined its architecture and has dedicated all of its silicon for the sole purpose of doing a specific task efficiently, making it the best machine to solve this problem.

Now, one may pose the question conversely: Why is not everything specialized processing machines? The answer is that the cost of development and manufacturing favor mass production. A highly specialized machine will likely be produced in lower quantities and therefore the effort put for producing a single unit is higher.

It is this trade-off that the FPGA disrupts. The FPGA silicon itself is mass produced and is pre-validated on transistor-level. The FPGA designer will need to design and validate only their own functionality, thus greatly lowering the non-recurring engineering effort to produce specialized silicon.

## 1.1 Motivation

There is a consensus that FPGAs are a powerful tool for computation. However there are certain unsolved problems that impede their spread. One of these is the lack of a standardized interface to the operating system and a common framework to support the building of accelerators.

At the operating system front, it is only recently that the Linux kernel gained awareness of the dynamically reconfigurable hardware (see section 5.10) and the supporting framework is under development. A generic userspace API and ABI is yet to be defined and currently, at its initial stage, it focuses on basic functionality like the abstracted view of reconfiguration interface, the correct automatic probing and removal of dependent device drivers and the software isolation during partial reconfiguration, etc. Clearly, a lot of things will change in the near future – but for now, there is no way to have an environment of dynamically reconfigurable accelerators without writing kernel code.

At the system software front, the situation is worse. There is no universally accepted method of time scheduling hardware accelerators in a dynamically reconfigurable environment. There is some work done in proprietary systems. As for open source frameworks, there exist a couple of frameworks that tackle the issue of parallel processing in a heterogenous environment, but as their scope includes all types of processing devices, they have grown large and complex. The few academic approaches to a simple and open dynamically reconfigurable acceleration typically pass the burden of scheduling to the end-user.

At the hardware front, fortunately, there is significant academic work to support a reconfigurable accelerator framework, albeit dynamic partial reconfiguration is not often used. All academic work is focused to the model of a PCIe attached FPGA. In recent years, we saw the rise of integrated FPGA SoCs where the programmable logic is closely connected to the processor, sharing access to a common memory controller. To our knowledge, no academic work has yet published utilizing this novel architecture.

## 1.2 Our Approach

In this work we will explore the aforementioned shortcomings by creating a hardware and software framework that supports on-demand loading of custom accelerators. We decided to include a run-time scheduler to manage the tasks and the accelerator cores, in order to relieve the end user of such a responsibility. A user request is posted to the system and served asynchronously, in order to allow the user to perform their own computation in parallel to the hardware processing.

The target usage is a server environment that offers hardware acceleration capability to user applications that feature an a priori known set of computational kernels. This environment was inspired by the upcoming Xeon hybrid CPU+FPGA processors, but since we had no access to such hardware, our initial target platform was set to be the Xilinx Zynq-7000 All Programmable SoC. Additionally, we will try to port our system to the newer and more advanced Zynq UltraScale+.

Due to the nature of the intended environment, the system must be multi-user and security will matter. The end user software API must be simple and offer an abstracted view of the hardware system. The user shall be allowed to configure their task's affinity to the system accelerator slots. The system administrator shall be able to add or remove

an accelerator to the system, as well as to configure its priority and its availability to the system slots. Additionally, they shall be able to configure the scheduling algorithms and security policies. Changes shall be effective immediately and without interruption of normal operation.

The system adaptability will be realized by the use of dynamic partial reconfiguration. We will attempt to create high performance architectures by exploiting the opportunities for parallelization that the hardware platform offers.

It is of paramount importance that the system would be flexible to support any accelerator arrangement and any memory and interconnect layout with minimal effort.

### 1.3 Contributions

The implementation of such a system was successful, as we achieved most of our goals.

We implemented a kernel driver that assumes the responsibility of programming, configuration and exchange of data between the accelerator and the user. We opted for an in-kernel scheduler which manages resources and synchronization. Several scheduling algorithms were implemented, for selecting the most appropriate free slot and for finding the best victim for eviction.

The system was made multi-user. A user posts a task for execution and the system will schedule it. The user has no direct access to the scheduler and cannot steal priority. No sensitive data cross the user-kernel barrier, so the user cannot discover other user's data. System-wide operations can be restricted to the system root user, in order to prevent a user to affect system scheduling decisions in their favor or to unload an accelerator that is desired by another user.

Nonetheless, our security model has weaknesses. The system is aware only of tasks and accelerator resources. It is not aware of the task owner, the user, so in current form, no user or group quotas may be assigned. This makes the system susceptible to DoS attacks (see section 5.6). Also, proper operation by unprivileged users still needs some work.

The software API is made simple and is grouped to task operations, available to all users, and to system operations, available to the system administrator. All intended functionality was implemented.

In order to support efficient interconnect and memory usage, we implemented a

segmented memory model. Available memory is divided in zones, which serve as configurable pools for allocation. The natural use of these zones is to match the interconnect architecture. This way we achieve balancing of data traffic among the ports that connect the programmable logic to the processor system which includes the memory controller. We made possible to define a memory zone's bandwidth capacity, in order to indirectly affect its desirability for accelerator usage.

The system manages a degree of accelerator slot heterogeneity. Not all slots are required to be equal and certain accelerator types may not fit certain slots. The scheduler will take into account this fact in order to assess how attractive a system slot is, and the memory allocator will use this information to determine the potential scheduling freedom a memory zone selection will provide.

The segmented memory model combined with the affinity capability allow the implementation of basic quality-of-service and assignment of isolated or dedicated memory paths to a set of accelerators. These are desirable capabilities for a mixed criticality system.

Finally, the goal of flexibility was achieved with the proper use of the Flattened Device Tree (FDT). A designer may implement any accelerator arrangement, any interconnect architecture and any memory layout. The implementation tool will generate an hardware description file, out of which the FDT can be generated. Nonetheless, the designer will still need to manually describe a certain amount of additional details regarding the organization of the hardware design. The final FDT will be passed to the kernel and is sufficient to fully describe the hardware – no source code modification is needed. For more information, please see section 4.5.1.

A shortcoming of our system is that memory layout, as well as the implemented static hardware design cannot change without system reboot. This is due to older Linux kernel limitations on modifying an active FDT. Please see section 5.10 for details.

Finally, the port to Zynq UltraScale+ had to be halted. We completed the hardware design and the partial reconfiguration workflow, but we could not support it by the driver. The reason is that Xilinx has yet to offer software support for the partial reconfiguration capability through the PCAP interface.



# Chapter 2

## Related Work

The concept of allowing a user software application an abstracted interface to the programmable logic on an FPGA is not new. The first attempts to offer such a functionality involved commercial entities offering highly integrated solutions. Such solutions consist a complete system that includes not only their own proprietary software but also their hardware, or even, their own languages. Therefore, apart from being non-free, the cost of purchase can be very high. The best known system vendors are Maxeler, Convey (now Micron), Pico Computing (also aquired by Micron) and Impulse Accelerated Technologies, the latter being the only that targets commodity FPGAs. Xillybus is also a commercial product, but unlike all others it only sells an IP core that drives Xilinx PCIe endpoint block. Additionally, it offers their backing device driver as open source which is actually mainlined in Linux kernel.

There exist free and open source (F/OSS) solutions however. OpenCPI and OmpSs try to create heterogeneous computation environments that make use any available computing element be it a CPU, a GPU, an FPGA, etc. They utilise commodity hardware and their existing languages and toolchains. They are ambitious projects with far-reaching vision, being in development for several years and been grown to large and complex systems.

A newer and even more ambitious project is ReconOS [1]. In this work they attempt to unify the software and the hardware processing. They propose the idea of the “hardware thread”, a hardware processing that is represented as a common thread. That thread acts and is acted upon with the available multithreading programming model.

The first simple F/OSS framework came, ironically, from Microsoft Research by K.Eguro. The SIRC [2] enables the communication of a PC with an FPGA through an Ethernet link, which limits its attainable throughput and latency. Unsurprisingly, it only supports MS Windows. These two major disadvantages impede most practical use.

The MPRACE [3] framework is an attempt to create a library with useful hardware cores (including an open source DMA engine) and corresponding device drivers that would enable data exchange between an FPGA and the host PC. The MPRACE repository has no activity after paper publication.

A major step forward was the RIFFA Framework [4] as it is the first to conceive the idea of having multiple independent generic accelerator engines in a single FPGA platform. RIFFA abstracts the data transfer between a user application and a PCI-e attached FPGA accelerator by the means of named pipes. It is minimalistic by design but significant effort was put to offer the highest performance the data transport medium offers, but equally importantly, to support as many environments as possible. It can use Xilinx Integrated Block for PCI Express, Altera IP Compiler for PCI Express and Altera HardIP For PCI Express, effectively covering all modern large PCIe capable FPGAs. A driver is provided for both GNU/Linux and MS Windows, and an API is offered for C, C++, Java, Python and MATLAB. It offers zero-copy data transfer through the means of creating a scatterlist of the user buffer. It supports multiple FPGA boards but no partial configuration. Apart from the two “big” systems, it appears to be the only one framework that receives continued development and have three papers published. The API is simple and the source code is very well written and commented. They also offer extensive documentation and step-by-step instructions for using their work. The combination of these characteristics rendered it as a baseline model for all future attempts as well as a foundation to build upon.

Two years after SIRC publication, in 2012, R.Bittner of Microsoft Research published a new system, Speedy [5], which solved the performance bottleneck by supporting the PCI-e bus and it uses a random-access interface to the end-user utilizing the on-board DDR of Xilinx ML505. Authors even designed their own DDR2 controller, to challenge the Xilinx MIG performance. However, as its forerunner, it is Windows-only – even the source code is packed in a Windows executable, so it could not be studied. Generally, due to its restrictions and narrower application it is overshadowed by the

significantly more extensive RIFFA, which predated it by a few months.

The EPEE [6] diversifies itself from RIFFA by implementing a user-controllable register and user-defined hardware interrupts. The latter will interrupt a blocked system call – no asynchronous notification mechanism (i.e. POSIX Signals) is offered. It offers zero-copy by mapping a kernel buffer to the user and it also supports buffered operations. It has a partial reconfiguration “plugin” that is a simple user function that reads a partial bitstream and programs it to the FPGA. The system is not aware of the reconfiguration and therefore no synchronization, queueing, scheduling or other kind of management of modules takes place. In all fairness, they do not advertize their system as partially reconfigurable. The API is simple and source code is clean, but it appears the authors have no interest in further development of their work.

All published works make use of the Xilinx PCIe endpoint block to drive the PCIe protocol. The Gen2 block for 6-series [7] and 5-series [8] offers a low-level interface that requires significant effort and knowledge of the PCIe protocol in order to be driven by the designer logic. Indeed, the majority of the effort invested by the aforementioned works focuses at the hardware block that interfaces the designer logic with the Xilinx PCIe endpoint. Xilinx has since then released a Gen3 block for its 7-series [9] and UltraScale devices [10] as well as the older 6-class [11]. These blocks present a significantly different user interface compared to the Gen2 block. A significant addition is the offering of a DMA bridge for PCIe [12] which drastically simplifies connectivity as it can be set up to offer a generic Advanced eXtensible Interface (AXI)4 or AXI-Stream interface.

The introduction of these cores allow the creation of a PCIe-based accelerator framework with minimal effort. A designer may still prefer a lower level interface if they wish a finer control over the data transfer, or cannot tolerate the large amount of logic [13] the DMA/PCIe bridge consumes. Or they may skip both the bridge and the endpoint entirely if they require a fully custom or a pure F/OSS solution. But now, there is little incentive to develop yet another one PCIe endpoint driver if none of the above stand true.

The ffLink [14] is the first F/OSS work to be published that uses the new Gen3 endpoint and consequently offer PCIe 3.0 support. The authors found the resource utilization of DMA/PCIe bridge unacceptable and they used the generic AXI DMA IP cores to drive the Gen3 endpoint. Currently, ffLink is no more available as a standalone project – it is integrated in ThreadPoolComposer [15], which in turn was superseded

by TaPaSCo [16], a bigger project at TU-Darmstadt which appears to be current and well maintained.

A very interesting work is done by Vipin et al, 2013 [17]. It is an attempt to create a generic CUDA-like programming interface to the FPGAs. They abstract access via PCIe, FPGA-based DRAM and Ethernet to generic AXI-Lite and AXI-Stream interfaces. Additionally, they offer some FPGA device-related functionality such as programming and restarting the programmable logic, and even provide diagnostic information (power, voltage, temperature). An interesting feature is that they abstract the workflow of three high level synthesis tools, Vivado HLS, Bluespec and SCORE. The interface is very similar to CUDA or OpenCL: The user writes the HLS code inside the application as an accelerator core to load. The framework assumes the responsibility to synthesize and implement the code. It is an open-source work but source code could not be found.

A year later, Vipin released DyRACT [18]. In this work they offer the functionality of partial reconfiguration, more or less with the same limited way the EPEE does. They have implemented a custom ICAP controller that offers superior performance to Xilinx implementation, achieving a throughput of 91% of the maximum ICAP capability. The framework still does not support zero-copy transfers nor multiple boards. The API is clean and the source is well-written and commented.

Another published work is JetStream [19]. This work attempts to cover all desired features: partial reconfiguration, multiple DMA transfer modes, multiple board support and direct FPGA to FPGA data transfer. Regarding the transfer modes, the nomenclature they use is confusing, contradictory and even invalid, but they do support both buffered and zero-copy DMA. The feature of direct transfer between FPGAs simply means that data will move via the PCIe root complex and will not traverse the memory controller – there is no direct dedicated link between the accelerators as the name would suggest. Finally, the partial reconfiguration feature is not described in the paper and could not be found in the sources – it is certain however that there is no accelerator scheduler. An inspection of the source code reveals that the driver frequently delegates a lot of the work to the end user, leaking kernel data structures to the user-space or exporting functionality that should not be at a user application's authority. Furthermore, the software code is completely undocumented, contains several rough edges and the repository shows no activity since the paper publication. In our opinion,

it is half-finished and, at least regarding the software, a work of questionable quality.

A newer work is RACOS [20]. This work introduces the novel idea of multi-core and multi-threaded accelerators. A multi-core accelerator consists of two discrete accelerators sharing the same reconfigurable partition. A multi-threaded accelerator is an accelerator that can save its context internally and context-swap to another user – in both cases, the connection to the DMA controller is shared. RACOS is especially focused in reconfiguration speed achieving a 99.6% efficiency in ICAP reconfiguration throughput, while they have overclocked the interface from 100 to 125 MHz. However, the most important addition is that RACOS implements a task queue and a scheduler that process it, so the end-user can post a request to be executed asynchronously. The key disadvantage of the system is that it is the only academic work mentioned here that is closed source.

Juxtaposing this work with the aforementioned published works, the most apparent differentiation point is that the data transport is the AMBA AXI bus. This changes the working environment and the role of the FPGA from an attached accelerator to a closely connected co-processor that may have shared access to the main memory.

Feature-wise, our implementation of partial reconfiguration includes an asynchronously executing scheduler, just like RACOS. Our implementation of zero-copy DMA uses kernel allocated buffers mapped to the userspace. Compared to building a scatterlist of user allocated pages, the choice of RIFFA and some others, it is simpler and offers higher performance from doing many small DMAs, even if the DMA controller has Scatter-Gather support. However, the main reason we followed this way is so we can build a custom memory allocator that has full control of all allocated buffers. This was essential for supporting our novel feature of having segmented memory to control and balance the flow of data throughout the interconnect.

Another important feature is that our system was designed from the beginning as multi-user, in order to be applied in a FPGA accelerated server. Among other works, only the architecture of RACOS may be capable of this feature, but this subject was not touched in the paper. Despite that our system is not yet secure (see 5.6), its flaws can be corrected.

In figure 2.1 we display a comparative table of the frameworks that were designed with the multiple accelerator model in mind, partially reconfigurable or not.

Framework	F/OSS	Transport	Partial Reconfiguration	Accelerator Scheduler	Zero Copy	User Clock	Accelerator Configuration	Accelerator Interrupt	Multiple FPGAs	FPGA to FPGA direct link	O/S		FPGA	
											GNU/Linux	Windows	Xilinx	Altera
RIFFA v2.2	✓	PCIe 3.0 x4	✗	✗	✓	✗	✗	✗	✓	✗	✓	✓	✓	✓
EPEE	✓	PCIe 2.0 x8	✗	✗	✓	✓	✓	✓	✗	✗	✓	✗	✓	✗
ffLink	✓	PCIe 3.0 x8	✗	✗	✓	✗	✓	✓	✗	✗	✓	✗	✓	✗
DyRACT	✓	PCIe 3.0 x4	✓	✓	✗	✓	✓	✗	✗	✗	✓	✗	✓	✗
RACOS	✗	PCIe 2.0 x4	✓	✓	✗	✗	✓	✗	✗	✗	✓	✗	✓	✗
JetStream	✓	PCIe 3.0 x8	?	✗	✓	✗	✓	✗	✓	✓	✓	✗	✓	✗
This work	✓	AXI	✓	✓	✓	✗	✓	✗	✗	✗	✓	✗	✓	✗

**Figure 2.1:** Comparison of present FPGA accelerator frameworks.

# Chapter 3

## Background

On this chapter, we will discuss key technologies that were used in this work. The target hardware platforms will be introduced. An emphasis will be given to the communication protocol and the interconnect architecture as they were a pivotal point for this system's design.

Afterwards, we will analyze and compare the strategies for exchanging data between the programmable logic and the processor. We will present the available IP cores that implement the communication and there will be a discussion on their operation modes, the possible configurations, and the design trade-offs involved in their implementation.

Finally, we will discuss the Partial Reconfiguration workflow, describing the benefits and the pitfalls this technology encompasses.

### 3.1 The Hardware Platform

The primary hardware target of this work was the Zynq-7000 All Programmable SoC family from Xilinx. An “All Programmable SoC“ in Xilinx terminology or an “SoC FPGA” according to Altera/Intel and Microsemi, is a device that combines a “hard” (i.e. implemented in silicon, not soft-IP) System-on-Chip and the fabric of an FPGA.

The notion of the “SoC” typically encompasses all the basic functional elements of an embedded computer system capable to run a modern general purpose operating system, excluding the main memory and storage. Still, vendors deviate from this definition – Microsemi uses a microcontroller grade processing system based on Cortex-M3

which does not feature an MMU restricting operating system choice<sup>\*</sup> whereas Altera and Xilinx use the Cortex-A9 and A53 application processors. Still, the assortment of peripherals differ: Some devices omit even the GPU while others include secondary independent real-time processors.

The integrated FPGA fabric is comparable that of the standalone FPGAs these vendors offer. For example, the lower members of the Zynq-7000 family contain Artix-7 class fabric, while the higher members contain a Kintex-7 class.

As a secondary hardware platform, a port for the Zynq UltraScale+ was made. The port was intended to explore the significantly different (compared to Zynq 7000) mechanics of partial reconfiguration as well as affirming the system's portability to a 64-bit SoC.

In the following table, the most important technical specifications of these two platforms are described.

	Zynq 7000	Zynq UltraScale+
Board	Development Board	ZedBoard
	Device Model	Z-7020 (-1)
	PS Memory	512MiB DDR3 (fixed)
	PL Memory	n/a
Processing System	Main Processor	ARM Cortex-A9
		32bit, 2 cores, 866MHz
		L1 32kiB i/d per core
		L2 512KiB, OCM 256KiB
	Real-Time Processor	n/a
	Graphics Processor	n/a
Programmable Logic	FPGA fabric	Artix-7 (28nm)
	LUT	53.2k
	FF	106.4k
	BRAM (Mb)	4.9
	DSP (slices)	220
	Transceivers	n/a
		24 GTH (16.3Gbps)

**Figure 3.1:** Technical specifications of the target hardware platforms.

The internal interconnect of the Processing System (PS) and its connectivity with

<sup>\*</sup>There is a port of GNU/Linux, the  $\mu$ Clinux Project, that enables support of systems that do not feature a Memory Management Unit. The port has been included in the mainline kernel.

the Programmable Logic (PL) will be covered in detail in section 3.3. Before that, it is important to describe the communication protocol that all the on-chip interconnect utilizes.

## 3.2 The Communication Protocol

In order for two (or more) entities to exchange data, there must be a well-defined protocol. With the growth of FPGA ecosystem, a need for a common and widespread communication protocol arose in order to replace custom solutions that deemed too inflexible for bigger designs comprising IP from different project teams and different companies. The ARM's proposal is the Advanced Microcontroller Bus Architecture (AMBA) protocol suite which, as the name suggests, was initially deployed for microcontroller use but later expanded to SoCs gaining momentum as a result of ARM's dominance in the smartphone market. Since all modern FPGA-SoCs from Xilinx use ARM cores, AMBA became a natural choice for the company. Earlier products from Xilinx, like Virtex-II Pro which featured a PowerPC core, used IBM's CoreConnect bus architecture. The other big contender is the free and open source "Wishbone Protocol", which, not unexpectedly, is the favorite of "OpenCores" open-source hardware community.

The Zynq 7000 and the newer Zynq UltraScale+, the two platforms that are targeted by this work, both feature ARM cores and are designed around the AXI protocol, part of the AMBA suite. As Xilinx tries to promote IP reuse with its IP Integrator tool, it has expanded the use of AXI in its FPGAs that contain no ARM IP. The AXI infrastructure and several basic AXI peripherals are offered by Xilinx in Vivado at no additional cost. Therefore, AXI was chosen for the development of this system.

It is important to note that AMBA protocols are for on-chip interconnect only. Although they can transverse the PS-PL border, they are never exposed outside of the chip. This stands true not only for Xilinx's – or Altera's – FPGAs, but also for all ARM-based microcontrollers and SoCs.

### 3.2.1 The AMBA AXI Family

The AXI itself is essentially a group of protocols that support different topologies, as well as feature levels that position themselves differently at the trade-off between performance and functionality versus silicon area. However, they all share a fundamental bus concept: a multi-channel non shared-bus architecture that contains separate channels for each transaction type. It can be considered as a complementary to Advanced High-performance Bus (AHB) protocol, also member of AMBA suite, which is a single-channel multiple-master shared-bus architecture. Comparing these two families would give an edge to AXI in throughput performance and clock frequency requirements, while AHB would favor better in terms of latency, wire count and power requirements.

The AXI family has several members, but for this system the following three were used:

- **AXI**, was the initial and only member in AMBA 3.0. With the advent of AMBA 4.0 which introduced two new members described below, it is now usually referred as “Full AXI” or “AXI Memory Mapped”. The characterization of “full” contrasts to the reduced capability AXI-Lite and “memory mapped” contrasts to AXI-Stream which has no notion of address spaces.

The AXI comprises five channels: Read Data (R), Read Address (AR), Write Data (W), Write Address (AW), Write Response (B). An AXI link may be unidirectional, discarding the unneeded channels.

The addressing information must be communicated before a transfer to take place, which consists a performance barrier. To amend this, AXI supports burst mode, where sequential beats of data may be transferred without re-transmitting any addressing information. Between the two communicating endpoints, an intermediary “AXI Interconnect” must be inserted.

Its typical use in the FPGA realm is transferring data between memory resources, like (BRAM)s, processor RAM, FPGA memory controllers, etc.

- **AXI-Lite**. Introduced with AMBA 4.0, the AXI-Lite, as the name implies, is a reduced capability version of AXI. The most notable simplification is the lack of support for burst transfers. In exchange, it offers a much lower silicon footprint.

It is best suited for low intensity traffic, typically for configuration or status registers.

- **AXI-Stream.** Also introduced with AMBA 4.0, AXI-Stream is a data streaming protocol, which means that it has no notion of memory addressing. This greatly simplifies implementation and reduces wire count. Data flows from the one endpoint to the other, in one direction, without the need of any intermediary interconnect. Transmission size is not known in advance; data will flow indefinitely until a control signal (TLAST) is asserted.

AXI-Stream allows the addition of user defined out-of-band data, typically for synchronization, and it supports sender and receiver IDs, which enables stream routing for virtual circuits.

None of these protocols supports cache coherency. In AMBA 3.0, ARM proposed the Accelerator Coherency Port (ACP), an AXI slave port that connects an AXI master directly to the processor. The coherency logic inside the processor will monitor the transactions and update its caches accordingly. However, since the AXI master is not aware of the cache coherency logic, ACP is only an IO Coherency mechanism; the processor caches may be coherent but the accelerator's are not.

In AMBA 4.0, ARM extended the AXI protocol with AXI Coherency Extensions (ACE), which allows full coherency between the processor and the accelerator, and ACE-Lite, an IO Coherent version. The latter differs from ACP in that its coherency is managed by the interconnect and therefore the port requires no proximity to the processor. These protocols are supported in the newer UltraScale+ but not in the Zynq-7000.

Finally, in latest AMBA version, 5.0, ARM added Coherent Hub Interface (CHI), which targets the multiprocessor's local interconnect hub.

In the data streaming model that this work targets, there exists no spatial or temporal locality. Cached transfers are not only useless but harmful, since they will cause cache thrashing.

Indeed, the kernel driver uses the Linux DMA Streaming API which bypasses all processor caches by marking the allocated DMA'able pages as non-cacheable. Therefore, cache coherency will not matter our discussion any further; however, the hard-

ware interconnect that implements these cache-coherent protocols may be of our use and therefore it will be examined.

### 3.2.2 The AXI Implementation

The AXI implementation in Xilinx products consists of the hardware implementation in Zynq 7000 and Zynq UltraScale+ devices, the soft-IP protocol infrastructure offered in IP Integrator, and the AXI compatible IP building blocks. Additionally, Xilinx offers automation for creating custom cores with AXI interfaces. It is worth to cover this functionality as part of understanding the connectivity of the system.

#### The Zynq Hard IP

The Zynq 7000 is built around AMBA 3.0. The interconnect will be presented at the next section, but it is important to mention here that the use of AXI of this AMBA version carries two important restrictions: The original specification of AXI, as is present in AMBA 3.0 has a maximum burst size of 16. Any AXI master residing in PL that connects to the PS through a slave port, will have to obey this limit or use a protocol converter. Secondly, AMBA 3.0 does not support AXI-Lite or AXI-Stream, therefore all ports in Zynq-7000 are Full AXI.

The Zynq UltraScale+ is AMBA 4.0 compliant. Therefore, both of the aforementioned issues do not apply. Still, the AXI FIFO Interface (AFI) that supports the HP ports is version 3 only, and protocol conversion takes place in silicon. This process is transparent to the designer but is still important as it affects performance.

#### The Xilinx Soft IP

At the PL front, Xilinx offers a suite of IP cores that manipulate the AXI traffic. It offers cores for conversion (stream combining, width conversion), buffering (clock conversion, stream pipelining, FIFO buffers) and routing (stream broadcasting, stream switching, AXI crossbar). Additionally there are some higher level AXI building blocks that automate the interconnect of AXI endpoints. Due to their importance, it is worth to be mentioned separately:

- **AXI Interconnect.** It can connect  $M$  AXI masters to AXI  $N$  slaves communicating with either the Full AXI, both version 3.0 and 4.0, or the AXI-Lite pro-

TOCOL. The interconnect can be configured in full crossbar mode (shared address, multiple data) for high performance, or in shared access mode (shared address, shared data) for low area use, issuing only one transaction at a time.

The AXI Interconnect is built around the AXI Crossbar. The AXI Crossbar implements the core switching functionality and the AXI Interconnect wraps it with the appropriate port couplers that may perform necessary clock and width conversion and/or append register slices or FIFO buffers, to help timing closure and smooth traffic, respectively.

The AXI Crossbar allows the definition of a connectivity matrix for sparse crossbar implementation. However this feature is not used by AXI Interconnect and if desired, the designer must instantiate the AXI Crossbar and its couplers manually.

- **AXI SmartConnect.** This core is a newer design with functionality analogous to AXI Interconnect. It is advertised to be highly optimized to mitigate wire delays in UltraScale+. However that comes at some cost in FPGA resources. If the design uses a slow clock for the targeted FPGA or if the slaves are AXI-Lite configuration registers, the older AXI Interconnect should be preferred.
- **AXI Stream Interconnect.** The equivalent interconnect for AXI-Stream, as it can interface  $M$  AXI-Stream masters to  $N$  slaves. Likewise its sibling, it is built around the AXI-Stream Switch with the appropriate couplers in each of its interfaces.

The stream routing may either be defined externally through a configuration register or by sender / receiver IDs. It should be stressed that in contrast to its Full AXI counterparts, it is not an essential core if only a single master is connected to a single slave. Its use arises on shared physical links and/or where virtual circuit switching is needed.

Xilinx offers a few DMA controllers, compatible with both the full AXI and the AXI Stream. If implemented in the programmable logic, they can move data between an addressable memory resource and either another one, or alternatively stream it to a memoryless component. These are the solutions offered:

- **AXI DataMover:** This is the central component of all DMA controllers. Its role is to move data between the memory mapped and the streaming domain. It needs external logic for control, a role that is assumed by the three DMA controllers described below.
- **AXI DMA:** The AXI DMA can generate AXI-Stream compatible streams from a full AXI compatible addressable memory resource. Its core are two unidirectional or a single bidirectional \* AXI DataMover. It is configured by an AXI-Lite interface. The core has an optional scatter-gather engine that can continuously fetch and execute transfer descriptors without any pause as well as optional multichannel support.
- **AXI Video DMA:** This core is a variation of the AXI DMA specialized in video streams. Among other optimizations, it takes advantage of the user-defined out-of-band channel of AXI-Stream for frame synchronization.
- **Central DMA:** The Central DMA, probably a misnomer, can move data between two Full AXI compatible slaves, e.g. the processor memory and an AXI BRAM controller. It is implemented by a bidirectional DataMover and the control logic, with an optional scatter-gather engine.

### The User IP

As it becomes clear, Xilinx does offer a significant amount of AXI infrastructure IP and AXI compatible peripherals to support the implementation of an AXI-based system. Still, implementation of AXI compliant custom logic is a non-trivial task to undertake. Depending on the workflow and the designer's experience and demands, there are five options available.

- **Custom Implementation:** In case that maximum flexibility and performance is desired, a custom implementation is the way to go. Xilinx offers an AXI Verification IP which helps the designer to verify the functionality of an RTL design.

---

\*This configuration is not supported by Xilinx HSI for DeviceTree generation, which is the standard method describing hardware to the Linux kernel

- **IP Integrator:** Xilinx offers a “Create and Package IP” wizard in its IP Integrator tool. The designer may define the desired AXI parameters and the wizard will generate the corresponding RTL code to create the AXI interface. The designer can afterwards tweak the code to adapt their needs.
- **IP Interfaces (IPIFs):** IPIFs are IP cores that alleviate the burden of AXI conformance from the end designer by performing the complex AXI signaling themselves while offering a simple memory-like interface on the other end. Xilinx provides two such cores, one for Full AXI supporting burst transactions, and one for AXI-Lite.
- **Bridges:** In case the user IP is already developed with an alternative protocol, it may be possible to be bridged to the AXI interconnect, if the additional overhead can be tolerated. Xilinx provides only a handful of bridges, mostly of use within the AMBA family, e.g. for AHB-Lite (both slaves and masters) and for Advanced Peripheral Bus (APB) (slaves only). However, additional bridges may be found at OpenCores or in other open-source libraries. In the simplest case possible, a designer may even opt for the AXI GPIO core that can provide up to two 32 bit general purpose I/O lines.
- **HLS:** If the designer uses the Vivado High-Level Synthesis workflow, the tool is able generate AXI compliant IP using simple HLS directives. This is particularly useful to implement the HLS-core control protocol over AXI-Lite.

### 3.3 The Physical Interconnect

So far, we discussed the communication protocol and its implementation at both the programmable logic and the silicon domains. The next logical step would be to examine the underlying physical interconnect that supports it on the SoC-FPGAs that this work targets.

Granted, in the FPGA fabric there is infinite flexibility and any topology may be created. The presence of a hard IP however, presents a constraining factor. In both Zynq 7000 and UltraScale+ series, there is a single multi-port memory controller that resides on the PS side. Therefore, any traffic from/to the PL must first cross the PL-

PS boundary, then be routed inside the PS interconnect, and finally reach a memory controller port.

Understanding the nature of this path is not a trivial matter. Nonetheless, it imposes a number of hardware and software decisions in this work's implementation, and therefore it needs to be analyzed.

Since the architectural details of these two SoC-FPGA families are significantly different, they will be covered separately.

### 3.3.1 The Zynq 7000 Interconnect Architecture

The 7000 series glues the PL and PS together with a number of high speed ports of varying functionality. Most are slave ports to the PL, which means the transaction initiator must reside in the programmable logic. A couple of them are master ports, to be driven by either the ARM cores, the PS DMA controller or some I/O peripheral. One of them is able to provide IO Coherency, but there is no support for two-way, full coherency.

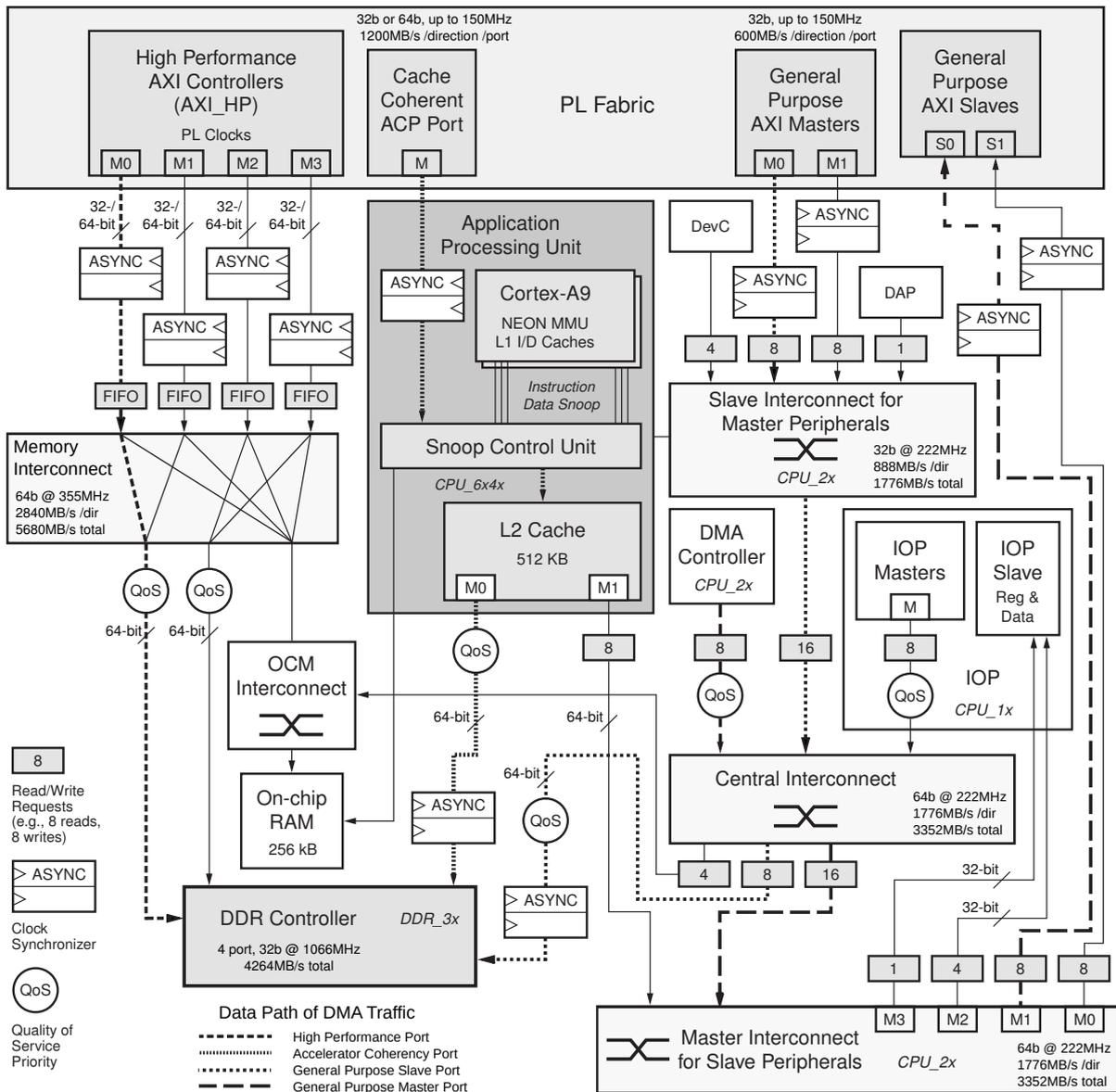
The figure 3.2 presents the system architecture of Zynq 7000 series, emphasizing the interconnect. Technical details are sourced from [21].

#### The High Performance Ports

The Zynq 7000 provides four HP ports, HP0 to HP3. These are all slave ports to the PL, conforming to AXI version 3. If the AXI master is implemented in AXI4, as is the usual case, protocol conversion must take place in the PL.

The ports are clocked by a PL clock at up to 150MHz, and can be 32 or 64 bit wide. As the AXI protocol mandates, they have separate wires for each direction, offering a per direction bandwidth of 1200MB/s for each port.

The HP ports are connected to the memory controller through the Memory Interconnect which in turn drives two of the four ports of the memory controllers, as well as one port of the On-Chip Memory (OCM) interconnect. The port clock will be converted to 355MHz, offering a switching speed of 2840MB/s per direction, 5680MB/s total. The switching scheme routes the traffic from the first two HP ports to the first memory port, and the other two HP ports to the second. Any port can be routed to the OCM.



**Figure 3.2:** The Zynq 7000 system architecture. Note that port naming follows the controller role, not the port's. For example, the "GP AXI Masters" are connected to the "GP AXI Slave Ports", titled "M0" and "M1". Conversely, the "GP AXI Slaves" are connected to the "GP AXI Master Ports", titled "S0" and "S1". Modified image from [21].

The memory controller offers an aggregate bandwidth of 4264MB/s shared among its four ports, irrespectively of the data flow direction. Therefore, if all HP ports are used and configured at their maximum ratings, the maximum theoretical bandwidth of 9600MB/s will saturate the memory interconnect, and if the OCM path is not used, will be further constricted by the memory controller.

### **The Accelerator Coherency Port**

The ACP port, compared to the HP ports, has equivalent performance specifications. The connectivity to the memory subsystem, however, is totally different. As it was mentioned in 3.2.1, the ACP port needs to be in close proximity to the processor in order to provide cache coherency to traffic generated from a non-coherent AXI master. Indeed, in Zynq-7000 series the ACP port is connected directly to the Snoop Control Unit (SCU) of the L2 Cache. From there, it can access one dedicated port of the memory controller. This is a low latency path to memory, but its tight relationship with the processor will complicate the potential usage scenarios.

### **The General Purpose Slave Ports**

The GP slave ports offer the half of the data width of the HP ports as they are 32 bit only, but they can operate at up to the same frequency of 150MHz. However, in order to reach the memory controller they follow a much more complicated path.

Firstly they reach the Slave Interconnect. They occupy two of its four slave ports, the other being dedicated to the Device Configuration controller (devc) and the Device Access Port (DAP). Note that the former will be heavily used at run-time as it is responsible for programming the FPGA during partial reconfiguration. The Slave Interconnect operates at 222MHz, offering an aggregate bandwidth of 888MB/s per direction. Its master port is connected to the Central Interconnect, which operates also at 222MHz but has a width of 64 bits, totaling at 1776MB/s per direction.

The Central Interconnect is shared by another two masters: The PS DMA controller and the I/O Peripherals (the flash memory interfaces, the USB and Ethernet controllers, etc). Itself is a master to three peripherals: The OCM interconnect to leads to OCM memory, the Master Interconnect which connects the PS masters to the fabric via the M\_GP ports, and finally, one port of the memory controller.

It is obvious that in order for S\_GP to reach the memory controller, it has to cross two rather busy interconnects and resource competing could become an issue.

### **The General Purpose Master Ports**

The GP master ports are the functional opposite of GP slave ports; they can connect PL slaves and they route the traffic through the Master Interconnect which in turn is a slave to the Central Interconnect. It is important to note that they are the only master ports from the PS side. If any transaction has to be started with the initiative of the PS, it *must* pass from these ports.

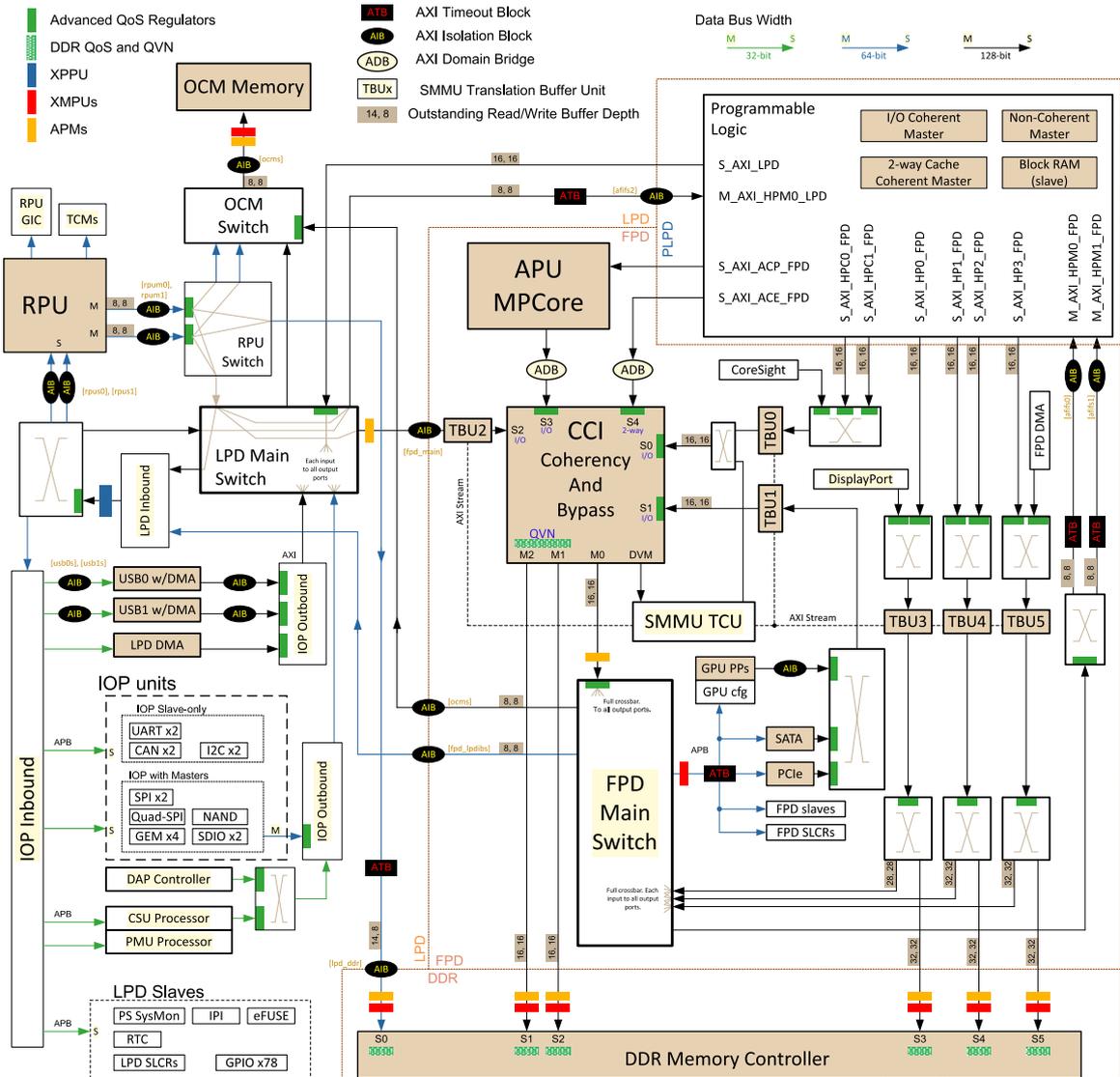
### **3.3.2 The Zynq UltraScale+ Interconnect Architecture**

The UltraScale+ series has significantly improved the system interconnect. Apart from the expected increase in number and bandwidth of the PS-PL ports and their pathway to the memory, there is much better support for cache-coherent peripherals. Unlike the 7000 series, in UltraScale+ all ports are of equal maximum width and frequency; they are all 128 bit wide and may operate at up to 333MHz. Nonetheless, since they have different connectivity and offer different functionality, subsequently they will also differ in access latency to the memory controller as well as produce distinct side effects. Additionally, they are mapped differently in the processor address space and have varying address widths; all of them however, are at least 32 bits. The UltraScale+ is updated to AXI4, with the exception of the AFI.

An overview focused on interconnect is displayed in figure 3.3. The reference for technical information presented here is [22].

### **High Performance Ports**

The UltraScale+ features four HP ports that reside in the Full Power Domain. As one can realize from figure 3.3, the path from an HP port to the memory controller is not identical for all ports. Indeed, the memory port S3 is shared between the HP0 and the DisplayPort controller while the S5 is shared between HP3 and the FPD DMA. The interfaces HP1 and HP2 share exclusive access to memory port S4, a property to be considered if the lowest latency or a deterministic performance is desired. Additionally, if the DisplayPort controller is not used, the HP0 will have full bandwidth access to the



**Figure 3.3:** The Zynq UltraScale+ system architecture, image from Xilinx [22]. The naming nomenclature denotes (in order) the master or slave role, the protocol (AXI), the port name with the modifier “C” for “coherent” or repeating “M” for “master” then followed by the port number, and finally the power domain designation.

S3 memory port. The HP3 would be the least attractive to use, since the memory access pattern of the FPD DMA may not always be known in advance.

### **High Performance Coherent Ports**

As the name suggests, the HPC ports are cache-coherent versions of HP, albeit IO coherent only, much like the ACP port. However, in contrast to the ACP, the coherency is not ensured by the SCU but by the CCI.

Decoupling the port from the processor has both its benefits and its shortcomings. The HPC ports have higher latency than ACP to the memory controller, even higher than HP ports since they have to cross CCI. On the other hand, it does not share a path with the processor to the cache, alleviating the resource competing in a such an important pathway. Xilinx labels the ACP as a “legacy” port, showing its preference in HPC.

There are two such ports in UltraScale+. They share access to a single port of the CCI, which they can reach after crossing two switches. From there, they can be routed to the two memory ports that are visible to the CCI.

### **High Performance Master Ports**

Essentially, the HPM ports inherit the role of Zynq-7000’s M\_GP ports. They are masters to the PL, so they are the gateway for any traffic generated with the initiative of the PS, that being the ARM cores or the PS DMA controllers.

There are three such ports. Two of them are in the FPD and one in the LPD. Unlike Zynq-7000, their performance is not inferior to HP ports.

### **The AXI Slave Port of LPD**

There is a single slave AXI port that connects the PL to the Low Power Domain. The port is connected to the LPD Main Switch, and from there is routed to the RPU after crossing two further switches. Along with the LPD HPM, they are the only means of accessing the PL from RPU side without crossing the Full Power Domain.

### **Accelerator Coherency Port**

Retained albeit unfavored from the Zynq-7000 series, the ACP port is upgraded to match the performance of the other UltraScale+ ports. Only one such a port is available.

### **AXI Coherency Extensions Port**

The ACE port is a unique addition to this FPGA-SoC family. With the help of CCI it can offer two-way, full cache coherency. That is, it can support a cache-enabled peripheral and maintain coherency of both the processor's and the peripheral's cache.

## **3.4 Exchanging Data with the Programmable Logic**

The diversity of the physical interconnect creates a number of possible methods for transferring data between PS and PL. Each method may benefit a specific transfer pattern or may be more appropriate for an application. The temporal characteristics of the transfer, the amount of data to be transmitted, the requirements for latency or throughput, the power consumption and the need of a higher level of determinism, all of them will derive the optimal solution for each problem.

### **3.4.1 Programmed I/O from a Processor**

The Zynq 7000 features two ARM Cortex-A9 processor cores that can generate load/store requests. These may target the DDR memory and the OCM directly from the cache ports and the SCU respectively, or the PL through the means of Master Interconnect (see 3.2).<sup>1</sup>

The UltraScale+ is a bit more complicated since it features two processor clusters that use different pathways. The high-power A53 cores in APU are connected to the CCI and from there they may reach either the DDR memory controller, or be routed from FPD Main Switch either to one of the High Performance Master ports to reach the PL, or the OCM Switch to reach the OCM. The low-power R5 in RPU can only send to the RPU Switch. From there it can reach three targets: The OCM Switch that provides access to the OCM, the DDR memory controller directly, and the LPD HPM that gives access to the PL. Additionally, it has a link to the CCI through the LPD Main

Switch from where it can be routed anywhere in the Full Power Domain, including the FPD HPMs.

Overall, the main advantage should already be obvious: The Programmed I/O method can low-latency access to any component of the PS and the PL, without the need of any PS or PL third-party actor, like a DMA controller. Thus, it saves resources on both PS and PL.

The second big advantage is simplicity. From software point of view, it is sufficient that the program issues the appropriate load / store instructions, with a possible memory barrier – no initialization of any component. As for hardware, a slave peripheral is sufficient to implement an AXI-Lite interface, which is low demanding in complexity and resources.

The major drawback derives from the very nature of programmed I/O and is not Zynq specific. The fact that the traffic is generated by load / store instructions is a twofold problem: First it keeps the processor busy issuing the instructions preventing it to perform any other useful work. The problem aggravates as the number of slaves increases, as obviously there cannot be more parallel transfers than the number of processors executing I/O. Secondly, the load / store instructions cannot generate burst transfers, essentially degenerating a full AXI to AXI-Lite. According to Xilinx ([21], v1.11, pg. 656) one should expect transfer rates of around 25MB/s in Zynq-7000.

The most common use case for this method is writing configuration registers, reading status or other initialization work. Such small data exchanges are latency sensitive, not throughput, making a perfect fit for programmed I/O.

### 3.4.2 Using the “hard” DMA controller in the PS

The Zynq-7000 features an eight-channel DMA controller on the PS side. Looking back at figure 3.2 we can see that it is connected with a single link to the Central Interconnect. From there, it may reach the DDR Controller directly, the OCM memory through the OCM Interconnect, and finally, through the Master Interconnect, it may reach the PL by using one of the M\_GPs (the coarsely dashed line).

The UltraScale+ has two DMA controllers, one in the Full Power Domain and one in the Low Power Domain. The FPD DMA controller shares a link with the HP3. The link is driven to a second switch that gives access to either the DDR memory controller,

or it is routed to the FPD Main Switch, in turn to the OCM Switch, and finally the OCM. The LPD DMA controller is connected to the I/O Peripheral Outbound Switch that is connected to the LPD Main Switch. From there it can see the OCM Switch and the OCM memory, the DDR memory controller, or it can enter the PL via the LPD HPM.

The DMA controllers can be programmed by a PS processor. The programming of a DMA controller is certainly a more complex matter than just issuing a load / store instruction, so an increase of software complexity is for granted. However, the obvious advantage is that they come “for free”, that is, they are already present in the silicon, not consuming any PL resources. They are multi-channel and they can provide a throughput of at least an order of magnitude higher than programmed I/O, without keeping busy the processor.

However, as we saw, the flow of data crosses a lot of already busy interconnects. The sharing of bandwidth reduces all aspects of performance, including its predictability and repeatability – of all users. We shall not forget also, that even the DMA controllers are actually shared with other system components, eg the network driver will typically use it to transfer data to/from the network interface. On top of that, the master ports in both 7000 and UltraScale+ are fewer, and in the former, are also narrower.

Specifically for this work, it should be added that PS DMA is full AXI compatible and has no AXI-Stream – and neither of the ports of 7000 or UltraScale+ are AXI-Stream-compatible anyway. That means that once the traffic is in the PL, it must be converted to AXI-Stream. The PL resources needed for this conversion are comparable to implementing a DMA controller directly to the PL, making this choice appear unattractive.

All in all, the use of the PS DMAs is a viable solution, albeit certainly not the best performant. To make things worse, it does not map favorably to the goals of this work. Therefore, this solution was abandoned.

### **3.4.3 Implementing a DMA controller in the PL**

Modern FPGAs are large enough to allow us the possibility to implement our own DMA controllers at a tolerable cost in PL resources.

The sacrifice of PL resources is not trivial. However this drawback could be offset

by the number of advantages that this method gives us:

- Both Zynq series feature significantly more slave ports to the PL than master ones. Their parallel use would increase aggregate bandwidth.
- There is the opportunity to use other interconnect pathways that are not being used by the usual client peripherals, alleviating the potential issue of an interconnect bottleneck.
- Using a path that does not cross the Central Interconnect and the Master or Slave Interconnect would have the benefit of reduced and more predictable latency. In the extreme case, one could dedicate a whole pathway for a specific latency critical accelerator.
- The PS DMA would be freed for use by other services, especially the network driver.
- We gain the flexibility implement the PL DMA exactly according to our specific needs. It could itself become grounds for research, as the capability of the DMA controller plays a significant role in overall system performance.
- It opens the possibility for the design of more complicated architectures than our case of an isolated accelerator that reads from memory, processes, and writes back. For example, output could be re-routed to another accelerator or to an external device (ie chip to chip data exchange, data acquisition or data display) without the need of going back and forth to the main memory.

The choice of slave port however, is a decisive factor as it can deny us several of the aforementioned advantages. Therefore, they should be treated separately.

### **The HP ports**

In Zynq-7000 series, the HP ports have an exclusive access to two memory ports through the memory interconnect. This is an impressive feature, considering that the processor and the Central Interconnect have only one each. Furthermore, by having four of them, we gain significant flexibility on the AXI interconnect that it will need to be implemented at the PL side.

Similarly, in UltraScale+, the four HP ports have access to three memory ports, after crossing two rows of switches.

The primary drawback of these ports is that they are not cache-coherent. In this implementation, the access pattern is a continuous stream of data that flow through the accelerator. That pattern has zero temporal locality – caching the data would be useless if not harmful.

Therefore, these ports would be the prime candidates for connecting the accelerators.

### **The HPC ports (UltraScale+ only)**

The HPC ports, in contrast to HP, cross the CCI to reach the memory controller. This has two side effects: Firstly, they can optionally be cache-coherent, and secondly, the crossing of CCI will induce a latency penalty.

Eventually, since cache coherency is not important for this implementation, the HP ports would be preferable. However, despite that HPC ports incur a small latency, they offer a path to access two further memory controller ports, increasing our potential bandwidth by 66%. Therefore, they could be used in addition to the HP ports, albeit with cache coherency disabled.

### **The ACP port**

The ACP port is an interesting addition not only due to its IO coherent nature, but also due to its proximity with the processor. The ACP port would be ideal in an access pattern where the processor and the PL accelerator work together, exchanging small pieces of data or cooperatively working in the same dataset. Essentially, the “accelerator” would be more of a “co-processor”. In this access pattern, the data generated from the processor would stay in cache, from where the ACP-connected accelerator will retrieve it, without accessing the DDR memory at all. This would have a huge benefit to memory throughput and would alleviate the traffic of the memory controller. Additionally, since retrieving data from cache is an order of magnitude cheaper in energy than retrieving from memory, the power efficiency of the system would dramatically improve.

Nonetheless, this is not our case. Our streaming data pattern would cause continu-

ous cache misses and cache thrashing, increasing the latency and depriving the processor any cached memory. Furthermore, the ACP and the processor core share the connectivity of the SCU to the L2 cache, competing for access. Performance-wise it would be a disaster and therefore it will not be attempted.

### **The ACE port**

The ACE protocol differs from both HPC and ACP in that it offers full, two-way coherency. This permits maintaining cache coherency between the processor with caches and a full AXI peripheral with cache in the form of BRAM.

In comparison to ACP, the ACE (and also the HPC) do not compete with the processor for cache access. Yet, the ACE adds significant complexity to the slave and the interconnect. Its access to the memory controller is through the CCI, an access already gained through HPCs, leaving us little incentive to use this port.

### **The S\_GP in 7000 and the LPD Slave AXI in UltraScale+**

In section 3.3.1 it was described how S\_GP can reach the memory controller. It is a complex path that comprises crossing both the Slave Interconnect and the Central Interconnect. That cancels out a couple of our initial arguments for the use of the slave ports. Further discouraging is the fact that we will compete with the devc which controls the single Processor Configuration Access Port (PCAP) port used to reconfigure the FPGA.

Still, the S\_GP will open us access to one more port to the DDR controller. For this very reason, it is worth to experiment using it.

The LPD Slave AXI port of the UltraScale+ shares some characteristics with S\_GP. However, studying the figure 3.3 we will see that the port can reach the memory controller through the LPD Main Switch and the CCI, not the RPU Switch, whose only master is the RPU itself. This actually the case even for the LPD DMA.

This was an intended design decision. The RPU needs exclusive access to the memory controller in order to offer predictable and repeatable access latency, a key characteristic of its real-time nature.

Therefore, there is not much incentive to use it, since we already access the memory ports of the CCI via the HPCs.

## 3.5 Design Components

### 3.5.1 The DMA controller

By initial problem statement it was decided that the system will feature streaming accelerators. This narrows down the DMA controller selection to AXI DMA and AXI VDMA. The latter is video oriented, offering additional functionality that is desirable in video applications – and even required by some controller cores of imaging peripherals. Our test application is indeed imaging oriented and would be benefited by AXI VDMA in a few ways. Firstly, it would enable the possibility of using the Xilinx OpenCV-compatible HLS video libraries. Secondly, the out-of-band synchronization that is possible with the VDMA would enable the accelerator to detect when the image line changes and when the frame ends. Lastly, as VDMA was made for embedded use, it would be beneficial in case it was decided to bring this system in an embedded setting.

However, the test application is just for testing and not the only intended use. The loss of generality, or even the increased complexity due to specialization, was undesired. That leaves us with only the AXI DMA.

The AXI DMA IP core has a few operating modes that are worthy of mentioning.

- **Direct Register:** This is the simplest operating mode. A DMA operation is initiated by writing the control, source and destination registers. The processor can query the status register for completion, either by polling or by interrupt notification. Operation queuing is not supported, therefore there will be a time gap between completion and re-programming with next operation. This operation trades performance for resource utilization.
- **Scatter-Gather:** In this mode, the AXI DMA creates an auxiliary AXI port to a memory resource that contains a list of transfer descriptors. The AXI DMA will execute sequentially all the descriptors without any intermediate pause. Upon completion, it may pause or restart executing the same list. This mode offers higher performance as the DMA controller never stalls.
- **Multichannel:** An option to the Scatter-Gather mode, that allows multiple virtual channels on the streaming side. The AXI DMA will still have only two interfaces and the channels will be differentiated by the sideband information that

is carried over a standard AXI-Stream channel. This option is interesting in that it permits a single DMA controller to handle multiple accelerators.

The more complex operating modes show a lot of promise for our intended system. However, the Direct Register mode was chosen, on the grounds of simplicity and as a “starting point”. The potential of the other modes will be discussed at the “Future Work” section.

In table 3.4 the resource cost of each operating mode is displayed.

		LUT	FF	BRAM	Nets	
Zynq 7000	32 bit data	DR	1571	1996	2/0	404
		SG	2076	3235	2/0	593
		MC 1ch	2696	3807	2/0	637
		MC 8ch	3245	4188	2/0	637
		MC 16ch	3877	4626	2/0	637
	64 bit data	DR	1811	2394	2/2	544
		SG	2362	3636	2/2	733
		MC 1ch	2978	4208	2/2	777
		MC 8ch	3527	4589	2/2	777
		MC 16ch	4159	5027	2/2	777
Zynq UltraScale+	32 bit data	DR	1544	2002	2/0	404
		SG	2101	3244	2/0	593
		MC 1ch	2722	3816	2/0	637
		MC 8ch	3271	4197	2/0	637
		MC 16ch	3905	4636	2/0	637
	64 bit data	DR	1830	2403	2/2	544
		SG	2380	3645	2/2	733
		MC 1ch	3001	4217	2/2	777
		MC 8ch	3550	4598	2/2	777
		MC 16ch	4184	5037	2/2	777

**Figure 3.4:** Resource utilization of the AXI DMA core. Configuration: burst size 16, address width 32b. LUT: Look-up tables, FF: Flip-flops, BRAM: Block RAM (36kib/18kib), Nets: Boundary crossing nets. DR: Direct Register, SG: Scatter-Gather, MC: Multi-channel,  $n$ -ch: number of channels

As a final note, it should be mentioned that each AXI DMA core outputs two reset signals, one for the MM2S and one for the S2MM channel, that are asserted when the corresponding channel is also reset. It also outputs two interrupt signals, again, one

for each channel. The Zynq-7 has an interrupt input of 16 bits and Zynq UltraScale+ has two 8-bit inputs. The maximum number of AXI DMA that could be directly connected would be 8, which is very restrictive.

For the Zynq-7, a workaround approach was employed. We only use the S2MM interrupt output, and we “lie” at the operating system about MM2S. This is mandatory, as the Xilinx DMA driver requires both interrupts to be defined. In our kernel driver however, the DMA client to the Linux API takes care *not* to query or place any callback function on MM2S interrupt, and considers the transfer done when the S2MM interrupt is asserted. With this workaround, we raise the maximum number of AXI DMA instances to 16, which were sufficient given the Zedboard’s FPGA size.

For the Zynq UltraScale+, this does not suffice, as its size enables the instantiation of much more than 16 AXI DMAs. Therefore, in our port, we made use of the AXI Interrupt Controller IP Core, which supports up to 32 interrupts per instance. The Zynq-7 workaround was still used, in order to reduce the interrupt controller instance count and to avoid instantiating unnecessary wires.

### 3.5.2 The Interconnect

The choice of PL interconnect is a trade-off between several important metrics. The interconnect must match the required clock. Using register slices to pipeline the AXI channels is useful, but may not suffice if the critical path is within the switching logic.

A large and complex interconnect, one that connects many slaves to many masters may achieve maximum flexibility but will likely become the clock bottleneck, drastically reduce design routability and consume significant FPGA resources.

As an attempt to break down complex interconnects, both Zynq families offer multiple ports for pathways that will most probably host a large number of peripherals. The ports have either a silicon implemented switch that, from FPGA designer’s perspective, comes “for free” or they are routed to another memory port.

In order to explore the potential solutions, a test design was created for UltraScale+. It features eight accelerators that transfer data with 16 unidirectional links. We will then experiment on which interconnect will best match our needs.

Note that Xilinx does offer very detailed mathematical formulas for estimating resource utilization of each component of AXI Interconnect in 7-series FPGAs (see [23],

pp 35-52).

### A Naive Solution and Basic Configuration

Let us begin with using a single AXI Interconnect v2.1 with 16 slave interfaces (SI) and 1 master interface (MI), connected to a single HP port. The initial configuration will be a 32b data width crossbar, outer register slices in all interfaces, as well as a 32 byte FIFO.

The first thing to notice in this basic design, it that with the default memory map, the PL has access not only to the DDR, but also to PCIe, QSPI and OCM address spaces. This is made possible by appending an AXI MMU core on each slave interface. As it can be seen on table 3.5, eliminating this unused flexibility reduces lightly the resource usage. Other tweaks, like the map base or range have negligible effect.

Note that the “Nets” column refers to the core’s boundary crossing nets, and is a metric that predicts routability.

	LUT	FF	Nets
DDR, PCIe, QSPI, OCM	3002	5466	1792
DDR only	2883	5430	1792

**Figure 3.5:** The effect on memory map on resource utilization and routability.  
Configuration: AXI Interconnect v2.1, 16 SI / 1 MI, 32 bit crossbar, outer register and 32 byte FIFO per port.

The next challenge would be other two coupler components, the register slices and the FIFO buffer. The register slices are applied to all five channels of the full AXI, albeit using different structures (for details, see [23] pg 93). The optional FIFO buffer may be implemented in distributed RAM as 32-deep or in BRAM as 512-deep. The latter option will also add 32-deep FIFOs to address channels. Table 3.6 shows the effect of these options.

Apparently, these couplers take up the lion’s share of the total interconnect resources. Their importance however, differs greatly. The register slices, despite the fact they cause a rise of about 40% in LUT usage, during the development of the high core count design, they were found to permit a 10 to 20% higher clock in a congested circuit. Their cost was therefore justified. The FIFO however is mostly useful when data or clock conversion occurs in the interconnect. Therefore, the 32-deep version might be used if the overall design has sufficient free LUTs. The 512-deep version

	LUT	FF	Nets
No register slices, no FIFO	1337	526	1792
No register slices, 32-deep FIFO	2362	2800	1792
No register slices, 512-deep FIFO	4459	7967	1792
Outer register slices, no FIFO	1858	3156	1792
Outer register slices, 32-deep FIFO	2883	5430	1792

**Figure 3.6:** The effect of register slices and FIFOs on AXI Interconnect implementation size.  
Configuration: AXI Interconnect v2.1, 16 SI / 1 MI, 32 bit crossbar

however, not only dramatically increases resource usage, but also places area constraints on the design since BRAMs are found in specific areas, that at least in Zynq-7000 are scarce, since BRAM is also required by both the AXI DMA and our reconfigurable partitions in order to implement the convolution line buffer.

Presumably, the most important parameter of an interconnect would be its data channel's width. The width should match both that of the peripheral's and of the Zynq's connecting port. Table 3.7 demonstrate its effect on implementation size.

	LUT	FF	Nets
32 bit	2883	5430	1792
64 bit	3777	7942	2404
128 bit	5605	12966	3628

**Figure 3.7:** The effect of the AXI Crossbar width on AXI Interconnect implementation size.  
Configuration: AXI Interconnect v2.1, 16 SI / 1 MI, outer reg slices and 32-deep FIFO

Therefore, the resource utilization does increase, but there is little we can do. It is a parameter normally decided by the data width of the accelerator. If we manually override the crossbar width, the AXI Interconnect will place data width converters in all interfaces where the remote endpoint of the channel disagrees.

Here comes a significant pitfall in the Vivado tool. The crossbar width is inherited from the endpoint that connects to the Master Interface. In our case, the UltraScale+'s HP port. By default, these ports will be defined with the maximum possible value, i.e. 128 bits. The Slave Interfaces of AXI Interconnect are driven by the AXI DMA controllers, whose memory mapped channel's data width are inherited by the data width

of the AXI stream, which is decided by the accelerator. Note that AXI DMA will not allow a data width of less than 32 bits despite that it allows a stream data width down to 8 bits.

It is unlikely these values would match; data width converters would be instantiated by the AXI Interconnect, leading to severe increase in resource utilization. To illustrate this, the table 3.8 shows the effect of using the default settings when connecting a 32-bit accelerator to a 64-bit port (Zynq-7000 default) or 128-bit port (UltraScale+ default).

A special case that arose during the development of the UltraScale+ port, is the AXI-Lite peripheral control interconnect. If the IP core is automatically placed by Vivado Block Automation, it is instantiated with a data width converter on its single slave interface and the crossbar hierarchy is implemented in 32 bits. However, if it is manually placed, the data width converter will be placed at the slave interfaces of all tier-2 crossbars, while the tier-1 crossbar will be at the port width, i.e. 128 bits. This effect, in our case, lead to a comparative 2.5x increase in LUT and 2x increase in FF utilization.

	LUT	FF	Nets
32 bit SI, 32 bit MI	2883	5430	1792
32 bit SI, 64 bit MI	5421	5286	1826
32 bit SI, 128 bit MI	8053	8214	1962

**Figure 3.8:** The effect of master-slave interface data width mismatch on AXI Interconnect implementation size. Configuration: AXI Interconnect v2.1, 16 SI / 1 MI, outer reg slices and 32-deep FIFO

Unless it is a purposeful choice\*, it is advisable to configure the Zynq port at identical data width with the full AXI ports of AXI DMA.

### Attempting Parallel Access to Memory

Since both Zynq families offer multiple PS-PL ports that lead to multiple memory ports of the memory controller, it is an opportunity for access parallelization.

However, in section 3.3.1 we saw that, regarding Zynq-7000, the aggregate throughput of just two HP ports is sufficient to saturate the memory controller, even if we do

\* In our 16-core design, we did use an interconnect of 32b SI / 64b MI as a workaround. If the port was configured as 32b it was seen that for each 4 correct pixels the accelerator was receiving 4 zeroed ones, effectively creating black bars over the original image. The cause was not discovered but the workaround resolved the issue.

not take into account its other potential users, e.g. the processor or the I/O peripherals. Given the comparatively low throughput of the Zynq-7000 memory subsystem, one may wonder if there is sufficient motivation for even attempting parallelization. There are two arguments in favor of doing it – and virtually none against.

Firstly, the maximum theoretical throughput of 2400MB/s that Xilinx asserts ([21], pg 652) was calculated assuming maximum port frequency, maximum port width, and continuous bidirectional traffic. The latter is unlikely, but there is little we can do. As for the other two, they are subject to design trade-offs between per-accelerator performance vs. accelerator count and lower values might be chosen, effectively operating the port at a fraction of its maximum rate. At the PS side however, the memory interconnect and the memory controller operate in separate clock domains and with fixed width ports, so their maximum throughput is undifferentiated by the PL design.

Secondly, the complexity of the interconnect might limit the maximum attainable clock. Breaking down a single big interconnect to several smaller will allow a far better \* maximum clock.

Therefore, the parallelization is worth of exploring, the only question is which is the best architecture for our purpose.

Intuition would command us to increase the number of AXI Interconnect Master Interfaces. Granted, it violates our second argument, but since it is the simplest architecture and actually the first that was attempted in this work, it is worth of assessing it.

This configuration will complicate the processor memory map since a peripheral must be able to reach a processor address using exactly one possible route. This could be solved by segmenting the processor memory and offsetting the burden of access parallelization to the software, in the sense that if software chooses the memory buffers properly it can balance the transaction rates between all ports.

What cannot be solved however, is the hardware cost that arises of the increased complexity, as it is shown in table 3.9.

The resource increase is significant but not prohibitive. However one could see a source of redundancy: Even if the problem of accessing a memory address by multiple

---

\* According to [23] pg 23, for Artix-7 class FPGAs, like the smaller members of Zynq-7000 family, the maximum attainable clock for speed grade 2 devices is 130MHz for a 64b interconnect of 16 slave / 1 master interfaces, and 205MHz for a 4 slave / 1 master.

	LUT	FF	Nets
1 MI	2883	5430	1792
2 MI	3590	6195	2046
3 MI	4044	6912	2300
4 MI	4849	7641	2554

**Figure 3.9:** The effect of MI count on AXI Interconnect resource utilization and routability. Configuration: AXI Interconnect v2.1, 16 SI, 32 bit crossbar, outer register and FIFO size 32 per port.

routes can be solved by segmentation, it would be better if the redundant paths were not available at all. This would not only simplify software, but most importantly we would expect a simplification of the interconnect.

The solution is to break up the interconnect to several smaller. A 16 SI / 4 MI interconnect could be broken to four 4 SI / 1 MI interconnects. This would yield several advantages:

- Avoidance of segmentation would lead to software simplification.
- The removal of redundant paths is expected to reduce interconnect resource utilization.
- An interconnect with fewer SI or MI interfaces can work in higher clock speeds.
- It opens the way of asymmetric access to memory, including exclusive memory access for reconfigurable partitions hosting time-critical accelerators.

A secondary question would be how to split the interconnect. For example, if we decide to split the interconnect by two, would we split it by assigning half of the DMA controllers to the first part of the interconnect and half to the second, or it might be a better idea to split by channel type, given that read and write channels are significantly different? In order to answer this, a design with two 8 SI / 1 MI interconnects was created. At first the AXI DMA were split equally and later split by interface type. The results are in table 3.10.

Now, we can proceed to explore three interconnect architectures. The table 3.11 reveals the resource utilization for each one.

	LUT	FF	Nets
Split equally	3060	6290	2044
Split by interface type	2787	6006	2044

**Figure 3.10:** The effect of SI type homogeneity on AXI Interconnect resource utilization and routability. Configuration: AXI Interconnect v2.1, 2x 8 SI / 1 MI, 32 bit crossbar, outer register and FIFO size 32 per port.

		Logic LUT	FF	Nets
32 bit	1 I/C of 16 SI	2883	5430	1792
	2 I/C of 8 SI	2787	6006	2044
	4 I/C of 4 SI	3282	7420	2544
64 bit	1 I/C of 16 SI	3777	7942	2404
	2 I/C of 8 SI	3759	8790	2724
	4 I/C of 4 SI	4400	10812	3360
128 bit	1 I/C of 16 SI	5605	12966	3628
	2 I/C of 8 SI	5656	14359	4084
	4 I/C of 4 SI	6874	17604	4992

**Figure 3.11:** Effect of interconnect architecture on resource utilization and routability. Configuration: AXI Interconnect v2.1, outer register and FIFO size 32 for each port.

Apparently, splitting up the interconnect proved to only marginally increase resource utilization, essentially giving us all the aforementioned advantages at almost no cost.

### Comparing AXI Interconnect and SmartConnect

In 2016, Xilinx released a new IP core that implements the functionality of a Full AXI interconnect, claiming remarkably higher maximum clock at UltraScale+ devices.

During the hardware design development, the new IP core was put to test. The maximum clock frequency was sought in a moderately congested design, using a static workflow but with all partial reconfiguration related pblock placement constraints enabled. The end result was that using SmartConnect a clock of 143MHz was achieved, whereas with AXI Interconnect only 125MHz could be attained.

Unfortunately, this achievement came at the expense of resource utilization. As the increase was significant, it was deemed necessary to quantify the overhead. The

table 3.12 summarizes the results of implementing a test-case of a single 16 SI / 1 MI interconnect implemented with both the AXI Interconnect and the SmartConnect, in both Zynq-7000 and UltraScale+ devices.

		Zynq 7000			Zynq UltraScale+			
		LUT	FF	Nets	LUT	FF	Nets	
Single 16-MI	32 bit	AXI Interconnect v2.1	3305	5877	1756	2883	5430	1792
		SmartConnect v1.0	12910	9381	1777	14151	10835	1817
Single 16-MI	64 bit	AXI Interconnect v2.1	4231	8389	2368	3777	7942	2404
		SmartConnect v1.0	15605	12351	2429	15605	12351	2429
Quad 4-MI	32 bit	AXI Interconnect v2.1	4914	9150	2392	3282	7420	2544
		SmartConnect v1.0	13748	10470	1972	12998	9920	2052
Quad 4-MI	64 bit	AXI Interconnect v2.1	6082	12542	3208	4400	10812	3360
		SmartConnect v1.0	15046	12154	2652	14404	11608	2732

**Figure 3.12:** Comparison of AXI Interconnect and AXI SmartConnect, on 7000 and UltraScale+. Configuration: 4 instances of 4 SI / 1 MI, 32 bit crossbar, outer register and FIFO size 32 per port.

Apparently, the 15% gain in clock speed was accompanied by a threefold increase in LUT utilization. Therefore, its use may not be recommended in the following cases:

- If area is at a premium and functional completeness cannot be achieved with SmartConnect.
- If the critical path resides inside the accelerator core.
- For the AXI-Lite peripheral control interconnect. Either way, AXI-Lite peripherals cannot be clocked as high as full AXI or AXI-Stream ones \*. Given that configuration ports, the typical users of AXI-Lite, are not performance critical, it would be better that the AXI-Lite paths reside in a separate (lower) clock domain.

## 3.6 Partial Reconfiguration

The Partial Reconfiguration technology allows us to reconfigure a single accelerator without affecting the rest of the system. To illustrate why this is so important, let us

\*For the AXI DMA case, the figure is actually around 25% lower (see [24], pg 8).

imagine how our system would be implemented without this technology and we will discover the difficulties that will arise.

Our system needs to change its accelerators on-demand at run-time, and therefore the FPGA programming state has to continuously change.

Let us assume a homogeneous system of  $n$  accelerator slots, with each accelerator slot being able to host  $m$  different accelerator variants. The number of different possible system configurations would be  $m^n$ . In a realistic scenario  $m$  can be more than 10 while  $n$  can reach several dozens in a large FPGA. Apparently, it would be infeasible to create a bitstream for every possible configuration, as the computational effort could exceed our lifetime even if we used a mid-sized computer cluster. Even then, the bitstream storage medium would be impractical for an embedded system.

If we opt for the most useful configurations, the implementation would be doable but the system would be inflexible in case that the client load changes in an unpredicted way and it usually would not be a perfect fit for our load.

The next thing to consider would be the reconfiguration mechanics. If the FPGA had to be programmed all at once, we would have to stop accepting new requests, wait pending ones to finish, program the FPGA and start again. There are three points to consider here: If a pause in operation would be acceptable, if a full FPGA reset can be tolerated by the external circuitry, and finally, how much would be the overall performance degradation due to the pauses. The latter should be expected, as programming a big FPGA with 48-accelerators plus static logic would definitely incur a huge overhead, if what we needed to change was just a single accelerator instance.

At the other extreme, let us consider a hypothetical single big computation task that can be broken down to several stages, each of them able to fit in our FPGA. In order to accommodate all stages in an FPGA, we would need a high-end model of prohibitive cost. Alternatively, we could compute the stages sequentially in a small and inexpensive FPGA. In this scenario, we would have to use an external microcontroller with sufficient storage to hold the computation state while the FPGA is being re-programmed. The cycle of restoring state, executing, saving state, and re-programming, is costly not only in performance but also in power consumption.

The Partial Reconfiguration technology attempts to mitigate these issues by allowing re-programming only a part of the FPGA while the rest of the circuit continues operation unaffected.

In summary, partial reconfiguration:

- Allows continuous operation while part of the FPGA is being updated, eliminating system downtime.
- Reduces silicon requirements by making it easier to time-share the FPGA resources.
- Reduces bitstream storage demand as partial bitstreams are a fraction of the full bitstream in size and there are no redundant copies of the same accelerator in multiple configurations.
- Reduces configuration time as a result of smaller bitstreams.
- Reduces static power consumption by enabling the use of smaller FPGAs and smaller bitstream memory.

These effects combined allow the implementation of architectures that further improve cost or power consumption. For example, a machine vision system may add a low-power core version for object recognition and enable it when in low alertness level. This could not have been done with full bitstream programming as the other sensors need to remain online, or the frequency of swapping could harm responsiveness or performance degradation.

However, partial reconfiguration has its own disadvantages:

- Design time and effort are significantly increased, and higher skilled engineers are required. Relevant tools are less developed and not user-friendly.
- It may require additional license, further increasing the development cost. \*
- It will reduce FPGA area utilization efficiency and/or maximum attainable clock due to the physical constraints the technology imposes.
- The partially reconfigurable region will unlikely be a perfect fit for all reconfigurable modules. If the designer cannot divide their tasks evenly, additional FPGA area will be lost.

---

\*As of Vivado 2017.1, PR license is now included in System and Design editions.

### 3.6.1 The Partial Reconfiguration Workflow

The Partial Reconfiguration workflow is a bottom-up hierarchical approach. All logic that consists a functional unit we wish to be dynamically reconfigurable, the so called reconfigurable module, has to be synthesized independently of the design. Xilinx calls this technique as “out-of-context synthesis (OOC synthesis)” and is required to prevent cross-boundary optimizations between the dynamic and static parts of the design. The OOC synthesis will produce a netlist, saved as a “Device Check-Point (DCP)”, for each reconfigurable module variant that will later be applied to the static design to form a valid configuration, from which the partial bitstreams will be generated.

All reconfigurable module variants will be implemented in a certain physical region of the FPGA (a pblock) that we will need to define. A dynamically reconfigurable pblock is called “reconfigurable partition”.

There can be as many reconfigurable module variants implemented for a specific reconfigurable partition, provided we possess the means to store and deliver the bitstreams in the embedded system. Each reconfigurable module variant may implement any logic but the physical and logical interface to the static design must remain consistent – in case that the reconfigurable logic of any reconfigurable module variant requires different interconnect, a superset of all interface signals must be defined as the common reconfigurable module interface.

Likewise, there is no maximum number of reconfigurable partitions defined. However, as the implementation of a reconfigurable partition imposes physical constraints that reduce resource utilization efficiency and decrease routability, there would be practical limits that vary depending the FPGA architecture.

If a module needs to be instantiated more than once in different reconfigurable partitions, it has to be implemented separately and different bitstreams will be generated. This is an important detail considering the limited storage capability of embedded systems. Several approaches can be attempted, from bitstream compression to hierarchical storage systems.

The P.R. workflow defines these steps:

- Synthesis
  1. Initially, a static design that incorporates any set of the possible reconfigurable module variant is synthesized.

2. The synthesized design must be floorplanned to define the pblocks that will constitute the reconfigurable partitions. A reconfigurable partition must encompass all the logic that is required to implement any reconfigurable module variant, usually with an added overhead of the order of 20% depending the regularity of the reconfigurable module variants.
3. After floorplanning, all reconfigurable modules must be converted to black-boxes, rendering all the reconfigurable partitions empty. The produced (incomplete) synthesized design, the so-called “static design”, is saved as a DCP.
4. Each reconfigurable module must be synthesized using OOC synthesis. The synthesized result will be saved as a DCP.

- Implementation

1. The designer must create a set of “configurations”. Each configuration consists of the synthesized static design with all of its black-boxes replaced by a synthesized reconfigurable modules. Not all possible reconfigurable module combinations have to be represented by the configurations \*. It is sufficient to define as many configurations are needed so that every reconfigurable module variant is present in every desired reconfigurable partition.
2. One configuration will be chosen as the initial one. This will lead to the implementation of the top-level, static part of the design that will be saved and later reused. It will therefore decide the partition pins, i.e. the physical sites that will serve as the connection points between the static and the reconfigurable domains. These sites will be locked and all subsequent reconfigurable module implementations will be constrained to respect them. Therefore the initial configuration must be chosen carefully, as a bad choice may render some reconfigurable module variants unroutable for some reconfigurable partitions. It is recommended to begin with the configura-

---

\* There is an exception in the case that a signal from one reconfigurable module is connected directly to another reconfigurable module without any intermediate register in the static part. This technique is not recommended but it can be done. However, static timing analysis must be performed for every possible source and sink reconfigurable module variants. Therefore, these combinations must be represented in the defined configurations.

tion that contains the most difficult to route reconfigurable module variant of each reconfigurable partition. In order to find a partition pin set that permits the routing of all module variants, some trial and error is unavoidable.

3. All subsequent configurations import the static part of previous step and implement their reconfigurable module variants within its context.

- Bitstream Generation

1. After all configurations are implemented, they will be verified for compatibility to ensure the consistency of partition pins.
2. For each implemented configuration a full bitstream will be generated. From this bitstream, a partial bitstream will be extracted for each reconfigurable module that was implemented with that configuration. It is possible that a specific module variant for a specific reconfigurable partition to be extracted from more than one configuration, but should be functionally equivalent.

### 3.6.2 Floorplanning

Floorplanning for Partial Reconfiguration involves the placement and sizing of the pblocks that will implement the reconfigurable partitions.

The physical constraints of the Partial Reconfiguration technology in the respective FPGA architecture have to be respected. Furthermore, certain restrictions are derived from the implementation tools – most notably, reconfigurable partition nesting, overlapping or vertical stacking is not supported.

Apart from these, floorplanning in partial reconfiguration workflow is no different than in static design; only that problems are more pronounced and a sub-optimal decision will be more punishing. The difficulty heightens as the number of reconfigurable partitions increases.

As a starting point, a good practice proved to be to place the reconfigurable partitions as far as possible between each other as this eases local congestion.

Timing closure is first achieved in a static workflow that is gradually constrained to match the actual partial reconfiguration constraints. A reasonable workflow could

involve these steps:

1. Initial clock estimation and functional validation should be done in a static design instantiating the most difficult to route accelerator. No pblocks are defined. This implementation will set our expectations on clock speed and accelerator capacity.

Since reconfigurable partition may not be stacked within a clock region, if a special resource (BRAM or DSP) has to be included, the maximum number of reconfigurable partitions is bounded by the number of clock-region tall resource columns in the device. In 7-series, if the resource is the BRAM, this upper bound is further reduced because of the AXI DMA instances; ideally, five 32b or three 64b AXI DMA instances may be accommodated by a single clock region BRAM (see table 3.4), but this packing may not be possible in actual design.

2. Pblocks are defined and accelerator instances are assigned to them. Pblocks are placed as sparsely as possible and their size should target an 80% utilization. While a utilization of 85% to 90% may be possible to implement, it might be better left as an optimization after getting a first working P.R. implementation. If at this step the design is unroutable or it misses the clock target by far, the initial floorplanning may be fundamentally flawed.

3. The constraints of partial reconfiguration should be applied. Setting the HD.RECONFIGURABLE property will enable all the related constraints. It is possible to gradually import the P.R. constraints by initially using EXCLUSIVE\_PLACEMENT, that pushes all static logic outside of the pblocks, and CONTAIN\_ROUTING, which will force all reconfigurable module routing to be contained within the pblock \*

DRC for partial reconfiguration should be done now. Additionally, a good set of implementation settings must be determined – for our pblock arrangement it was usually a choice between the ExtraTimingOpt and RefinePlacement strategies.

---

\*In UltraScale+ this definition is relaxed to allow the reconfigurable module routing and partition pin to expand outside the reconfigurable partition's pblock borders, but no further than clock region boundaries. This is to facilitate routing but as more reconfigurable frames are involved, it incurs a cost at the size of the partial bitstreams.

4. If implementation has failed despite DRC was successful, the first step to try would be the placer parameters. Also, it must be ensured that pblock utilization is below 90% and all reconfigurable partitions contain the required special resources of their module variants.
5. In UltraScale+, if the initial reconfigurable module configuration succeeds placement and routing but the subsequent fails at placement due to unavailable BRAM or DSP tiles, a change of placer directive to `ExtraPlacementOpt` and pblock reshaping will resolve the issue.
6. If the initial configuration achieves timing closure while some of the others do not, the choice of initial configuration was incorrect and must be changed. If a configuration whose partition pins would satisfy all other configurations cannot be found, re-floorplanning may be required.
7. If all but one or two configurations succeed, the designer may try to re-synthesize the offending variant using an HLS output of lower target clock period. If this increases utilization too much or if timing closure is very near, the designer may attempt to tweak implementation settings of the specific variant only.
8. If the target clock was missed by a wide margin, e.g. a Total Negative Slack (TNS) of more than 10ns, a different floorplan may be required or the clock may be needed to be stepped down.
9. If the target clock was missed by a moderate amount, e.g. a TNS of 1ns, the implemented design must be examined in order to identify congestions that were caused by too dense pblock placement. A pblock may be moved to the next column pair (7-series) or have its aspect ratio changed (UltraScale+). It is useful to group the pblocks by the interconnect IP their reconfigurable modules use.
10. If timing closure is near, the specific settings of an implementation strategy can be tweaked. The most sensitive tool appears to be the placer. A P.R. workflow may be attempted – oddly enough, a case was met where a specific floorplan that failed timing closure in static workflow, succeeded in P.R. workflow at the same clock target (see figure 4.6).

# Chapter 4

## Hardware Architecture

This chapter is devoted to the system's hardware architecture. We will present the final hardware designs that were implemented, describing the reasoning behind the design choices.

The discussion will conclude with the application of partial reconfiguration technology on this work. We will look into its physical aspects that affected our design, the challenges of the floorplanning, and finally, the details of loading and configuring a new accelerator.

### 4.1 The Implemented Designs

Taking into account all these observations, an architecture had to be designed. Firstly, the design objectives must be stated:

- **Correctness:** The design will be the means to prove that the proposed system does work. It shall function properly for any user input data within its specifications and process it in a timely manner.
- **Flexibility:** The software (i.e. the kernel driver) shall support, without recompilation or additional user intervention, any possible interconnect architecture and memory topology as long as it is properly described in the FDT. An accelerator may have a restricted view of the addressable memory space or might be given exclusive access to a certain region. The memory space itself might be a collection of

different memory resources with unequal proximity to the programmable logic. The designer must be able to explicitly express a relative preference to one resource against another.

- **Performance:** The design will implement an efficient interconnect that enables parallel memory access for the accelerators. Its architecture must permit sufficiently high clock speeds. Finally the final system will show the performance of partial reconfiguration as well as the flexibility of the accelerator scheduler. However, we will not explore the optimal architecture for solving any specific problem.
- **Portability:** A constant effort throughout both hardware and software design was that they should be as decoupled as possible. The software shall use standardized operating system interfaces to ensure that porting the system to a different would be done with least effort possible.

In order to pursue these objectives, three designs were developed.

#### 4.1.1 An Accelerator Performance Oriented Approach

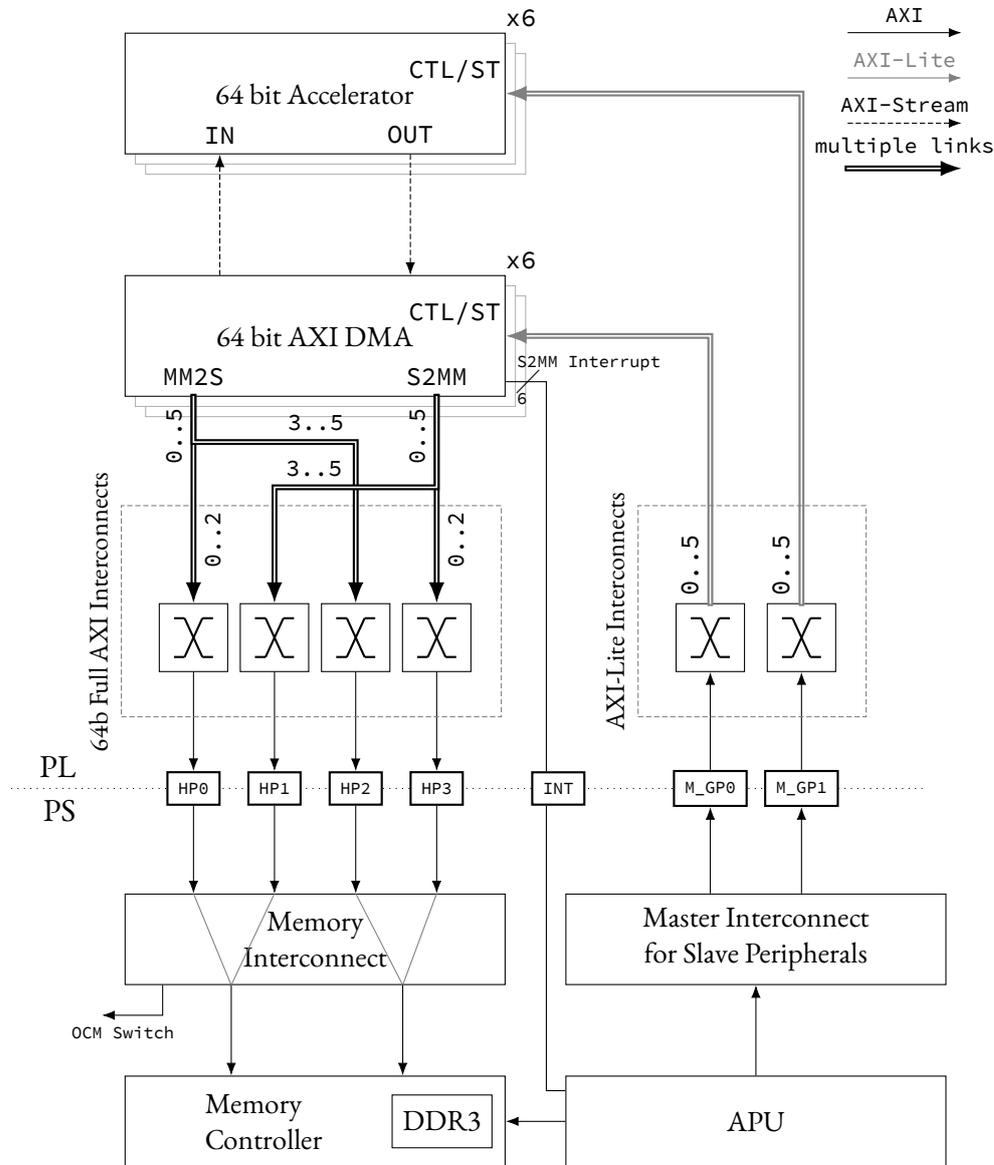
The first design is geared to accelerator performance, featuring accelerators with wider interconnect that allow greater data processing parallelization. It has homogeneous reconfigurable partitions with full memory view of uniform access on the accelerator side, allowing total freedom of accelerator placement that maximizes the scheduler's decision options.

The design was floorplanned and implemented with moderate difficulty. It was made possible to include 6 reconfigurable partitions of 64b data width, achieving a clock speed of 143MHz.

#### 4.1.2 An Accelerator Count Oriented Approach

The second design is geared to high accelerator core count that sacrifices per-core performance and flexibility.

More specifically, the following compromises have been made:



**Figure 4.1:** The accelerator performance oriented design's block diagram. Arrow direction is from master to slave.

- The accelerator data width has been reduced from 64 down to 16 bits. This cuts down accelerator size, in our application by 50 to 70 %. Furthermore, it helps routability by reducing the wire count of the incoming and outgoing AXI streams.

The AXI DMA was accordingly reduced to 32 bit data width at its memory

mapped channels, which in turn allows the reduction of AXI Interconnect's crossbar to 32 bits.

- The alternative pathway of S\_GP ports was also used, forming a secondary group of accelerators with inferior connectivity to memory of higher latency and less predictability. It still offers a throughput increase as long as the central interconnect is not saturated by other peripheral traffic.
- Two accelerator sizes are defined. Both sizes consist of the same number of LUTs, however the “big” one contains BRAM and DSP slices. This was decided as the Zedboard's FPGA is rather poor in BRAM slices, a resource that AXI DMA also needs, and the reconfiguration technology of 7-series FPGAs make it almost impossible for a single column of BRAM slices to be shared by static and reconfigurable design parts (see section 4.2.1). To ease the pressure on BRAM, a “small” accelerator size with no memory capability was introduced. In our application, it translates to an accelerator that can do pixel stream transformations but cannot perform a 2D convolution.
- Memory view is segmented. Each Zynq port is assigned an exclusive memory region where the accelerator can read and another that can write. Initially this restriction was implemented as it was found that it improved routability, however its real usefulness is to fulfill the stated flexibility goal of supporting a scenario of heterogeneous memory resources as well as the possibility of exclusive region access. Additionally, it allowed the implementation of allocator bias towards a memory resource over another. This feature was indeed activated to discourage the use of high-latency and congestion-prone S\_GP ports over the lower latency HPs that have a more direct access to the memory controller.

The floorplanning of this design proved to be very challenging in order to achieve a clock of 133MHz. However it could be implemented at 125MHz with relative ease. A total of 10 “big” reconfigurable partition and 6 “small” ones were included.

More importantly however, this design led to a complete rewrite of the kernel driver's special memory allocator to enable memory segmentation awareness. Furthermore, the scheduler needed to be revised in order to understand the partition schedulability constraints due to the corresponding interconnect's restricted view of memory.

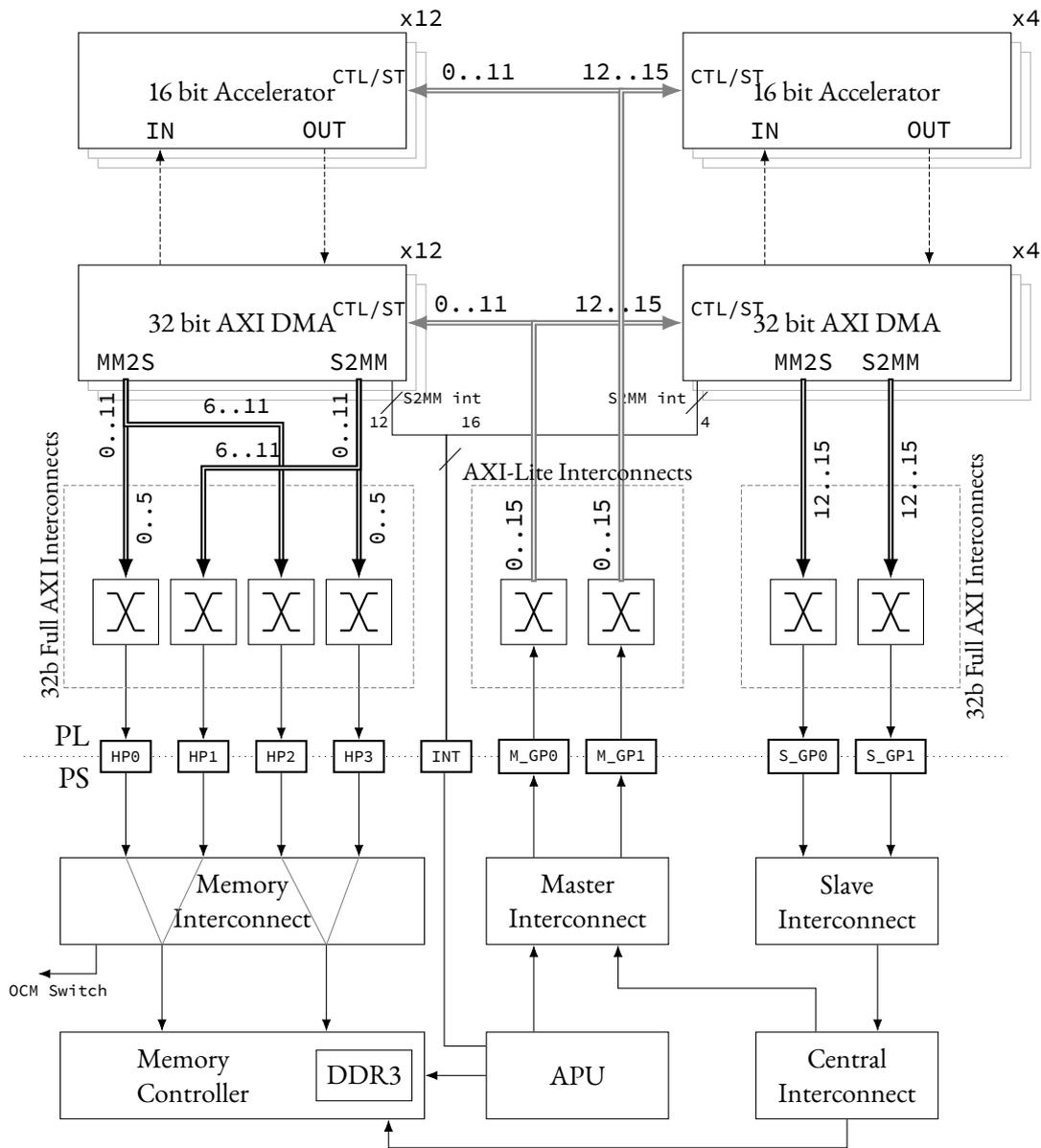


Figure 4.2: The accelerator core count oriented design's block diagram.

### 4.1.3 The Zynq UltraScale+ Port

The final design is the UltraScale+ port. The accelerator complexity is midway between the two other designs.

The newer platform's increased capacity, routing capabilities and more relaxed partial reconfiguration restrictions allowed the packing of several times more accelerator

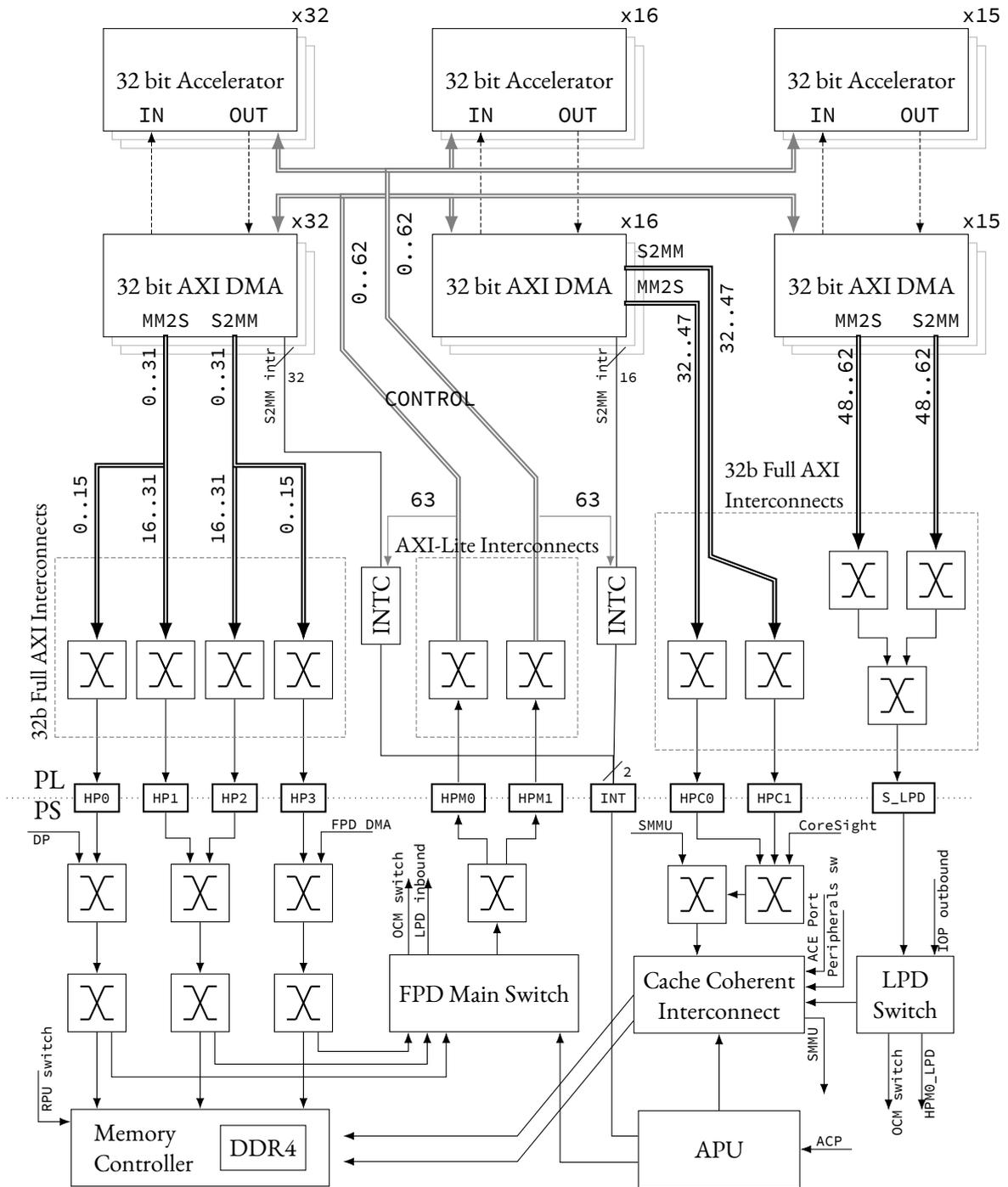


Figure 4.3: The UltraScale+ port block diagram.

instances operating at significantly higher frequency.

In total, 63 homogeneous reconfigurable partitions of 32b data width were implemented, running at 270MHz.

## 4.2 Enabling Partial Reconfiguration

### 4.2.1 Challenges

The Partial Reconfiguration workflow is subject to several physical constraints that arise from the FPGA architecture. These affect which hardware resources are reconfigurable, the granularity of the reconfigurable elements, the reconfigurable partition shape and placement, etc. These rules may range from absolute restrictions to good design practice advisories and are usually architecture dependent.

Between the Xilinx 7-series and UltraScale+ there has been a significant improvement in partial reconfiguration capability. In 7-series, only the CLB, the BRAM and the DSP tiles were partially reconfigurable. In UltraScale+, the clocking resources, the I/O and the Multi Gigabit Transceivers (MGTs) are also reconfigurable, albeit with coarser granularity. It is worth to note that in UltraScale+ the need of “clearing bitstreams” that was imposed in UltraScale devices is now removed.

However, what really matters to our work is the flexibility in the physical layout. This is because in this work we have several rather small accelerators. The shape and placement of their container pblock is much more critical to the system performance compared to, for example, a design with a single huge reconfigurable partition that spawns across multiple clock regions.

It is in this respect that the architectural differences are more important. In 7-series there are several physical constraints that lead to a rather coarse reconfigurable partition placement. A reconfigurable frame is a clock region tall. A reconfigurable partition is allowed to be shorter than this, however it will lose its automatic post-reconfiguration initialization to a known state. Even worse, as we will see in the following example, the vertical space between the reconfigurable partition and the clock region border would be of less usability, as no memory resource can be placed there. On the horizontal axis the reconfigurable partition is allowed to end at any column. In this architecture, each resource column has an interconnect column placed next to it. The pairs of resource

plus interconnect columns are placed sequentially but with mirrored orientation, that is, two resource columns facing each other, two interconnect columns connected back to back, and so on. Though it is legal that a reconfigurable partition ends between two interconnect columns, this would cause the splitting of a clock distribution resource. In order to avoid any disturbance, the tools will not place logic in any affected resource, causing a significant utilization efficiency drop. Therefore, a reconfigurable partition will almost always end at a resource column border. Effectively, the minimum reconfigurable partition size would be a clock region tall by two resource and their interconnect columns wide.

In UltraScale and UltraScale+ however, this has changed radically. A reconfigurable partition can be as small as a pair of CLBs or a single BRAM/DSP with its five neighboring CLBs. Xilinx actually recommends *against* having clock region tall partitions, as this is restrictive to the router. This fine-grained reconfigurable partition definition allows a far better FPGA utilization in the case of many small reconfigurable partitions. It also makes easier to define non-rectangular reconfigurable partitions, though routing difficulty at the pblock corners will still be a limiting factor. It must be noted that the reconfigurable frame is still one clock region tall and all logic in that column will be reprogrammed. However, in contrast to 7-series, in UltraScale+ consistency is guaranteed for any resource that is placed between the reconfigurable partition and the horizontal clock region border, including memory resources. Still, a reconfigurable frame may be controlled by one reconfigurable partition at most. Therefore, it is still prohibited for two reconfigurable partitions to be vertically stacked within a clock region.

To illustrate this, let us discuss the development of the two Zynq-7000 designs. Initially, the intent was that a single design of 12 accelerators would be created. Each accelerator, in order to convolve an 1080p image with a 5x5 kernel would need a line buffer of 1920 pixel by 5 lines by 1 byte/pixel for grayscale, which is 9600 bytes or 76.8 Kibits and therefore the implementation would require two BRAM36 or three BRAM18 resources. In figure 3.4 we saw that a single instance of a 32-bit AXI DMA requires two BRAM36 tiles. In total, for 12 cores, we need 36 BRAM18 and 24 BRAM36 tiles. The Z-7020 FPGA that Zedboard employs, has 14 clock region tall BRAM columns, each column containing 10 BRAM36 or equivalently 20 BRAM18 tiles. A static design would be easily implemented with a mere 30% BRAM utilization.

As a reconfigurable partition extends vertically to clock region border, each recon-

figurable partition would be assigned an entire BRAM column, underutilized at 3 out of 20 BRAM18 tiles. Out of the 14 columns the Z-7020 has, only two will be left for static logic offering 20 BRAM36 tiles, falling short of the 24 required. The designed was of course unplaceable.

A first thought was to make the reconfigurable partition shorter than a clock region, sacrificing the post-reconfiguration initialization. The accelerator reset signal was instead driven by the reset-out signal of the AXI DMA core, which is asserted when the AXI DMA itself is being reset. Since we anyways reset the corresponding AXI DMA after the accelerator reconfiguration, it seemed a viable solution.

However, the placer would fail, declaring such a placement illegal. Eventually, it was made clear that all logic within a reconfigurable frame is reconfigured, even if it is *not* included in the reconfigurable partition. The hardware guarantees consistency for any combinatorial logic, but any sequential element (flip-flop, LUTRAM, BRAM) would be initialized, regardless of the post-reconfiguration reset enablement.

Despite that this could be tolerated if by location constraints we assigned this BRAM only to the soon to be reset AXI DMA instance, the placer refused to be “forced” to accept this placement. The only solution was to place both accelerator and its DMA controller in the same reconfigurable partition but this was not materialized as it would double bitstream size and reconfiguration time.

At this point, it was decided to split the design to two targets, one of high accelerator count where some of the instances would contain no memory and one of high performance with no compromise in resource utilization.

As this physical constraint does not exist in UltraScale+, our design was implemented using rows of shorter than a clock region reconfigurable partitions, which additionally could be placed more tightly in the horizontal axis as routing could pass from the bottom part of the clock region.

### 4.2.2 Implementation

The final implementation of the three example designs is presented in table 4.4 and the floorplanning of a static workflow implementation is shown in figure 4.5 for performance oriented design and figure 4.6 for the accelerator count oriented. Finally, figure 4.7 displays the UltraScale+ port implementation.

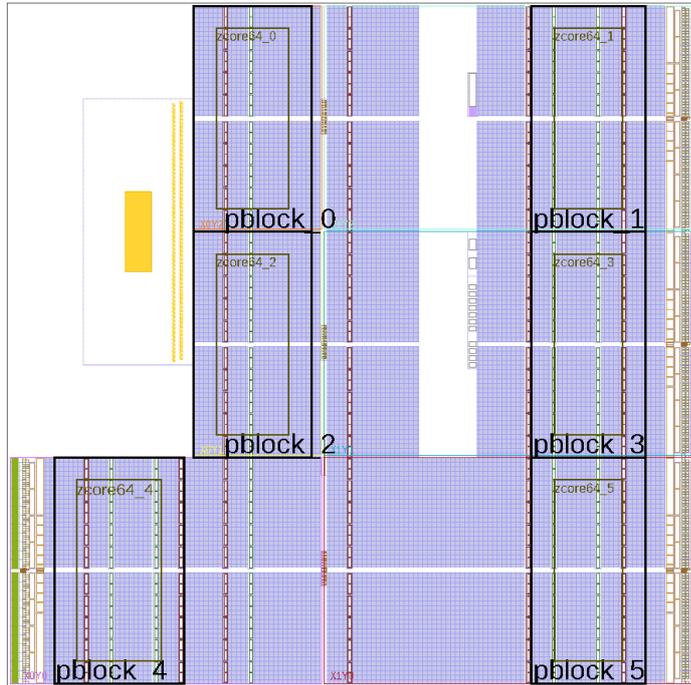
	Accelerator Performance	Accelerator Count	UltraScale+ Port
Accelerator Count	6	10 + 6 small	63
Accelerator Data Width	64	16	32
Clock Frequency (MHz)	143	133	250
LUT6 capacity	3600 - 4000	800	1680 - 1800
LUT6 utilization* (%)	87.61	46.25 - 89.62 <sup>†</sup>	76.67 - 89.05 <sup>†</sup>
BRAM36 capacity	10	10 / 0	6 - 10
BRAM36 utilization (%)	25	25 / -	41.7
DSP48 capacity	20 - 40	20 / 0	16 - 24
DSP48 utilization (%)	100	80 / -	75
Bitstream size (kiB)	516 - 666	294 / 148	341 - 637
AXI DMA Data Width	64	32	32
AXI Crossbar Width	64	32	32
Interconnect buffering	yes (32)	no	no
Interconnect IP	SmartConnect	AXI Interconnect	AXI Interconnect
Initial Config	gauss	contrast	gauss
HLS Target Clock (ns)	6.67	6.67, 7.5 <sup>c</sup>	3.33, 2.0 <sup>c</sup>
Logic Optimizer	Explore	Explore	Explore
Placer	ExtraTimingOpt	ExtraPostPlacementOpt <sup>§</sup> ExtraTimingOpt <sup>§,s</sup>	ExtraPostPlacementOpt
Physical Optimizer	Explore	Explore AlternateFlowWithRetiming <sup>g,s</sup>	Explore
Router	Explore	Explore	Explore MoreGlobalIterations <sup>c,h</sup>

**Figure 4.4:** Design parameters and implementation settings of each design.

<sup>s</sup>: sobel, <sup>g</sup>: gauss, <sup>c</sup>: contrast, <sup>h</sup>: sharpen. <sup>§</sup>: gauss is only placeable by forcing use of DSP48.

Despite the apparently high numbers of resource utilization in first design, it was the second one that proved to be the most challenging. The two reconfigurable partition types, “big” and “small” refer to the specialized resource capacity. The “big” contains BRAM and DSP tiles while the small does not. With respect to LUT6 capacity, they are equivalent.

Both other designs were homogeneous. The variation of pblock size in first design was used to help timing and there is no functionality difference. The UltraScale+ port required some tweaking in the pblock shape, size and placement in order to avoid an unexplained placer behavior of locking the specialized resources of initial configuration and causing subsequent ones to fail. This caused a variation in resource capacity, utilization and partial bitstream size, none of which has no further effect and will not be



**Figure 4.5:** Floorplan of accelerator performance oriented design.

visible at the system level.

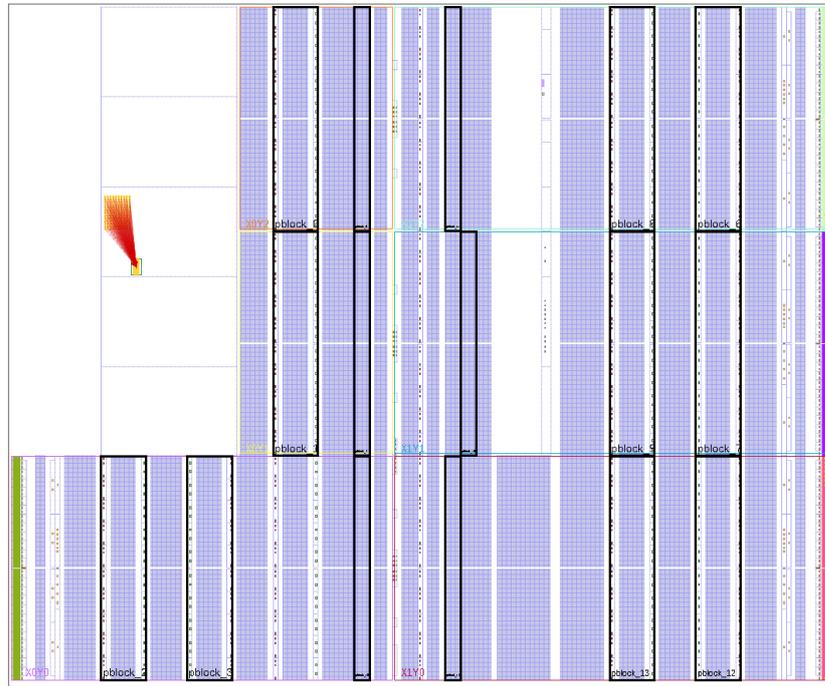
### 4.2.3 Partition Sizing

The size of a partition should be as small as possible, which roughly means a 10% bigger than the biggest module variant. But how big should a module need be? A fine-grained approach is problematic in many ways. It decreases routability by having more wires, it complicates the interconnect by requiring bigger crossbar switches, it makes placement more difficult and it wastes resources as practically a 10% of partition resources are left unused so to allow efficient intra-partition routing. On top of that, in our architecture there is a fixed overhead for each reconfigurable partition due to the necessary AXI DMA IP block, which, as we see in table 3.4, is roughly the size of a 32-bit accelerator slot in our system.

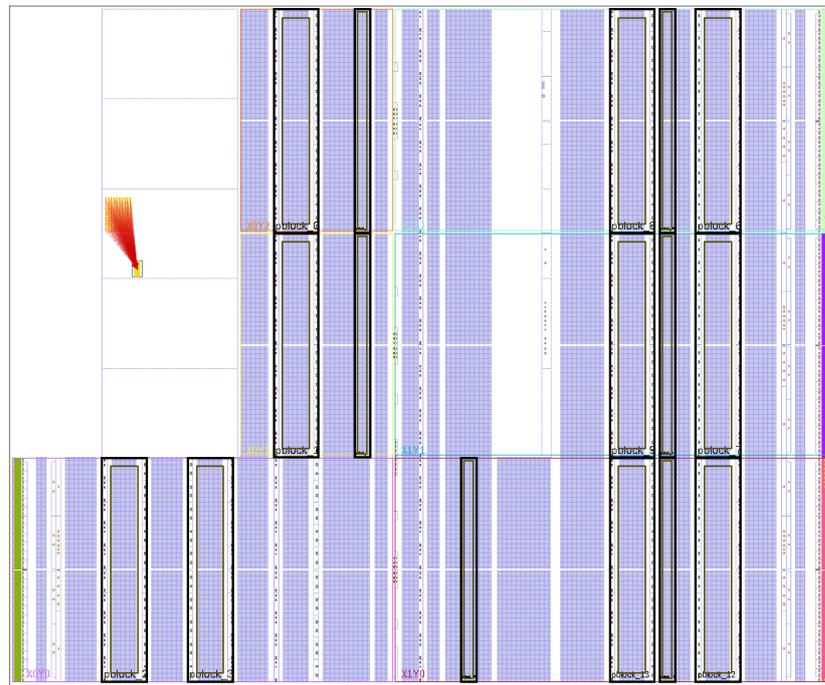
On the other hand, increase of partition size does not necessarily provide proportional latency decrease, as other factors (e.g. data I/O rates, specialized logic constraints,

\*Utilization of the biggest module variant in the smallest reconfigurable partition.

†The former is for the most difficult to route variant, the latter is for the most resource demanding.



(a) Successful timing closure in PR workflow but not in static.



(b) Successful timing closure in static workflow but not in PR.

**Figure 4.6:** Floorplan of accelerator count oriented design.

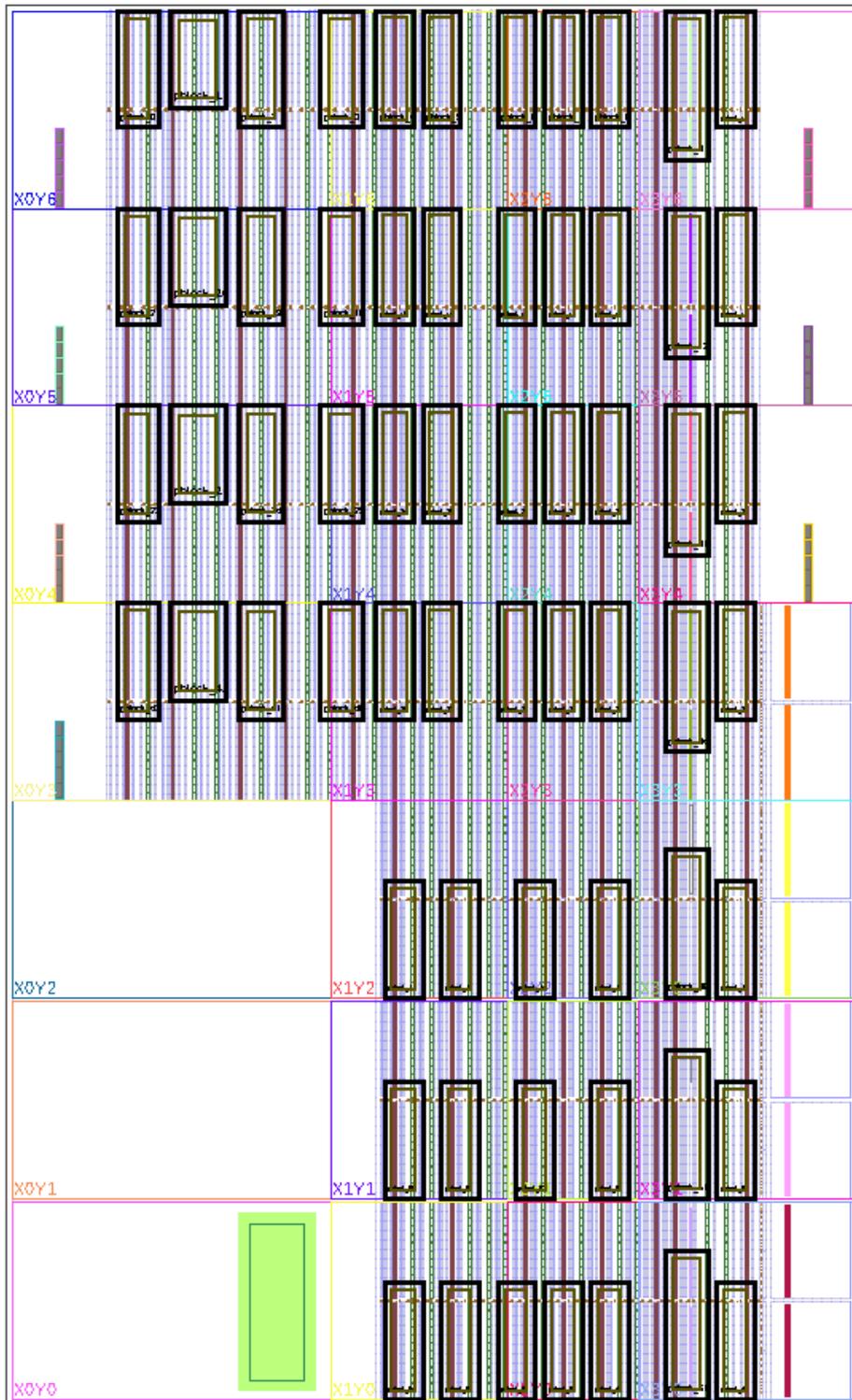


Figure 4.7: Floorplan of UltraScale+ port.

etc) may limit throughput. Often, we could replace one bigger partition with two smaller ones, where each could give an 80% of the bigger's capability.

In section 6.3 we see that while accelerators that do pixel transformations scale linearly with data width increase, the ones that perform 2D convolution see very little improvement as the BRAM cannot keep up with the increased bandwidth demands. The HLS estimates (see table 6.7) were misleading but they did predict a sublinear performance increase in these accelerators. In any case, it became evident that the advantage of scaling the pipeline must be verified by experiment as a resource contention may bottleneck our application.

At the end of the day, the advantage of a bigger and more complex accelerator strongly depends on the computation nature. However, it does also depend on FPGA architecture and the quality of the generated HDL code. A hypothetical FPGA with wider BRAM, with more read ports or with more BRAM columns, would alleviate the contention and turn the tide in favor of larger accelerators. Finally, a better implementation of the linebuffer that could reduce the BRAM traffic by buffering the data, could also achieve the same gain without changing the FPGA.

#### 4.2.4 Partition Heterogeneity

Despite that having two types of reconfigurable partitions in the 16-core design was initially driven by necessity, it still is a good design decision.

A system may require accelerators that vary significantly in complexity or demand very specialized resources, like the MGTs that were made reconfigurable in UltraScale+. In a different scenario, the same computation may be required in different degrees of intensity and power consumption. Or, as we saw at the previous section, a computation may benefit more or less from a wider, more pipelined accelerator.

At some point, taking the “one size fits all” path, it will essentially sacrifice area efficiency, power efficiency, and functionality offered.

In our heterogeneous design, a simple two-level approach was used. If a module variant cannot fit a certain reconfigurable partition, a loopback variant is used as a placeholder, which is later discarded. In this way, no partial bitstream will be produced for this combination of module variant and reconfigurable partition. The kernel driver will detect its unavailability and will take this into account in scheduling decisions.

### 4.2.5 Decoupling the Reconfigurable Logic

During the reconfiguration of a partition, the module's output is undefined and should be ignored. Depending on the design, it is a common practice to register the outputs or use multiplexers. More complex buses may require specialized IP, and Xilinx provides a PR Decoupler IP that supports the AXI bus. Additionally, the other parts of the system must be aware of the downtime. It is important not to generate any AXI traffic to any of the interfaces of the module that is being reconfigured or the bus may hang.

The Global Set/Reset (GSR) signal keeps the logic in quiescence during the programming procedure. However, in Series-7 devices, this can be disabled by not using the "RESET\_AFTER\_RECONFIG" pblock property and is always disabled if the reconfigurable partition does not extend vertically to clock region boundaries. In this case, even inputs and clocks have to be decoupled as spurious interrupts or memory writes may be generated.

In our case, all accelerator interfaces are slaves. Since no traffic is generated during reconfiguration – and this was indeed enforced in software – it was thought that no further decoupling was necessary.

However, it was noticed that frequently the AXI DMA that was responsible for a reconfigurable module, was behaving unexpectedly after the specific module's reconfiguration. More specifically, during first DMA transaction after reconfiguration, four identical groups of 8 bytes were prepended to data received. After that, the channel hung and would return to normal operation only by removing and re-inserting the kernel module, which in turn causes the Xilinx DMA driver to re-initialize all AXI DMA instances.

This phenomenon was never understood. However it was discovered by chance that if we soft-reset the AXI DMA and re-enable its interrupts, the issue went away. Since this solution costs no FPGA fabric and it does not disrupt normal operation, no other decoupling measure was used.

## 4.3 Accelerator Configuration

Most accelerators that may need to be implemented, apart from the input and output data stream, would need some kind of configuration input and/or status output.

This configuration input will affect the accelerator processing and must remain constant throughout the execution phase. After that, it is desirable that a status output will be offered.

A first thought would be the use of a GPIO port. However since the implemented functionality is not known in advance, the number of required signals is not known. Since the interface between the static part of a design and the partially reconfigurable must remain consistent throughout all reconfigurable module implementations, whatever choice is made, it has to apply for all reconfigurable modules that will be implemented. This has many consequences. If the choice is too conservative, a new accelerator type might not be able to be supported. If the choice is too lenient, many superfluous wires will be created, complicating routing. The choice may not be able to be revised afterwards, if the final product is released.

A sophisticated approach was used in [25], where the researchers used micro-reconfiguration, a technology of dynamically altering the configuration of an FPGA element without creating any additional bitstream. The principal use of micro-reconfiguration is for Single Event Upset (SEU) mitigation in space applications, for correcting bit flips in the configuration SRAM caused by high energy particles penetrating the silicon. When used in our context, it has the advantage of being able to configure an accelerator in any way possible without using any additional routing resources.

In this work, a simpler approach was used. Taking into advantage the low footprint of AXI-Lite (it was tested to cost about 100 LUTs for a slave port), a configuration port was implemented in all accelerators. This defines an address space within the accelerator, where the host may read or write any value. The wire count can be made constant by defining the address bus width – in this work it was chosen to be 6, leading to a 64 bytes of addressable space. Since this approach uses a general purpose bus, any AXI master can read or write this port, not only an owner of a FPGA programming port. The choice was primarily driven by the ease of implementation – all interconnect logic is available and the Vivado HLS is able to generate AXI-Lite compliant code through the use of simple compiler directives. The accelerator memory is mapped to the processor address space so communication is made through simple load/store instructions.

Nonetheless, long after its implementation, despite that it was proven to work reliably, there were doubts cast regarding the efficiency of this solution. More on this will

be discussed in section 7.2.

## 4.4 System Debugging

There can be several approaches at debugging the system. In its most basic form, the Linux kernel ring buffer can be used to post messages that can be later retrieved or be printed directly to the console. This method can offer extensive information about the system state and therefore is useful to analyze how the code entered an erroneous state of execution. However it is rather slow, prohibiting its use as real-time performance indicator.

An alternative path was to use the Zedboard's OLED screen. An IP was taken from [26]. As the author offered only bare-metal software, a Linux kernel driver was developed, making the OLED accessible from both within the kernel and from userspace as a character device. This solution proved to be much faster, but it still interferes with system performance. As it takes up significant space on the FPGA, it was used only during the first stages of development.

An FPGA-only solution had to be found. Thus, a simple IP core written in Verilog was implemented. This core accepts a configurable number of input events and blinks the corresponding outputs at an also selectable rate, appropriate for the human eye. The "events" were usually the DMA interrupts and the outputs were driven to the eight Zedboard user LEDs. When the input events are more than 8, the Zedboard toggle switches are used to select a group. A GUI for the IP Integrator was created.

Despite the minimal information than can be displayed, this is done at real time and without any performance impact. The IP core is quite small and was used even at the rather congested 16-core design. Using this core, one may discover possible scheduler inefficiencies in a heterogeneous design, revealing which reconfigurable partitions were under-utilized.

## 4.5 Describing the Hardware with a Device Tree

A general purpose operating system must gain knowledge of the platform it executes on. There exist computer buses that allow the automatic discovery of their peripherals, but for the majority of hardware, this is not a possibility.

Traditionally, supporting Linux on an embedded system required forking or patching the kernel with board specific drivers and parameters. As the embedded systems spread and the number of embedded platforms increased drastically, this technique proved incapable to scale. The kernel development had to be decoupled from any vendor implementation and a generic means of supporting any hardware configuration had to be invented.

The Device Tree (DT) was proposed as a solution to this problem. It is an hierarchical data structure describing hardware topology, no imperative programming functionality exists.

DEVICE TREE (DT) is not a new concept. It was designed by Sun for the Open Firmware system, as a means for handing over the hardware topology information to the operating system. Open Firmware is implemented commonly in PowerPC and SPARC platforms and as a consequence Linux already had support for it. In 2005, in order to ease platform maintainability, Linux mandated the DT support for all PowerPC systems, even if they did not use Open Firmware. In order to achieve this, the FDT representation was created. FDT is compiled to a binary blob that is passed to the kernel during boot by the bootloader – not the Open Firmware. More recently, in 2011, the use of FDT was expanded to the ARM target when insurmountable difficulties maintaining the BeagleBoard prompted Torvalds to stop supporting board files for any ARM platform. Today, the ARM implementation of FDT is done and most hardware companies support it, and Xilinx is not an exception.

For Zynq SoC FPGAs, the FDT is comprised of four parts:

- The description of the Zynq SoC with its on-chip peripherals.
- The board-specific information, e.g. the memory configuration and I/O peripherals.
- The hardware implemented in the FPGA fabric.
- The configuration of the implemented hardware.

The first two parts are created by Xilinx and are already in the mainline kernel, therefore we do not need to care about them.

The part of FDT that describes the implemented hardware can be generated by the Xilinx toolchain. Vivado can export the “Hardware Description File” of a design, that in turn is input to the SDK or the HSM/HSI command line tools, that are able to generate the FDT.

The last part, the one that describes the configuration and parameters of the implemented hardware, has to be written manually, as it describes the intended functional role and not a hardware instantiation. For example, an instance of the AXI DMA IP core will be included automatically at the third part. However, our intention to make it available to the Linux DMA Engine API has to be explicitly declared. The physical memory geometry description can be generated; the definition of memory segments and the description of which reconfigurable partition can read or write at each memory segment, cannot.

Note that the Device Tree may be in its source form, a .dts file, or its compiled form (a “blob”), a .dtb file. Despite that decompilation is perfectly possible (and indeed, very easy) one will lose label names and file hierarchy.

### 4.5.1 Writing a Device Tree for the System

As a guideline on how to write this part of FDT, we will use as an example the second, 16-core design.

To begin, we will declare the memory segments at the top level block “reserved-memory”.

```
reserved-memory {
    #address-cells = <1>;
    #size-cells = <1>;
    ranges;
    /* ... */
    hp0: hp0@10000000 {
        reg = <0x10000000 0x02000000>;
        compatible = "shared-dma-pool";
        no-map;
    };
    /* ... */
}
```

**Figure 4.8:** Declaring reserved memory

In listing 4.8, the identifier “hp0” before the colon is a “phandle”, i.e. a label that

will be referenced by the memory region user. The following `unit@address` syntax is not of much use; it is the “`reg`” property that actually defines the memory region at physical base address `0x10000000` (256MiB) and of length `0x02000000` bytes (i.e. 32MiB). The “`compatible`” property describes the entry for proper driver matching – in this case it indicates it will be added to a shared pool of DMA buffers. The “`no-map`” property instructs the operating system to not create a virtual mapping.

The “`reserved-memory`” block states the existence of reserved memory, not the usage. Inside our system’s sub-tree (the “`zdma@0`” node) we will define our system’s memory zones:

```

amba_pl {
    zdma@0 {
        compatible = "tuc,zdma";
        #address-cells = <1>;
        #size-cells = <1>;
        ranges;
        /* ... */
        zone@0 {
            compatible = "tuc,zone";
            memory-region = <&hp0>;
            readers = <&pb0 &pb1 &pb2 &pb3 &pb4 &pb5>;
            writers = <>;
            bandwidth = <100>;
        };
        /* ... */
    }
}

```

**Figure 4.9:** Memory zone definition

Here, we performed the following steps:

- We describe the node as a memory zone; our kernel driver will search the tree by this property.
- We declare our intent to use the memory region that we previously declared as reserved.
- We define a list of readers and writers permitted in this memory region. These must be a subset of the accelerator physical container blocks that the interconnect allows. In case we refer a physical block that instantiates an accelerator that

has no interconnect path to this memory region, an address decode error will be generated by the AXI DMA.

- We declare the bandwidth of the path to this memory region. This is an arbitrary magnitude. The higher the number we declare, the more eager our driver's memory allocator will be to use this zone.

A physical block (pblock) is a physical area of the FPGA, be it static or dynamically reconfigurable. In our system, it is a reconfigurable partition where an accelerator is instantiated.

```
pb0: pblock@0 {
    compatible = "tuc,pblock";
    core = <&zcore16_0>;
    transport = <&dma0>;
};
```

**Figure 4.10:** Reconfigurable partition definition

We define three parameters:

- The description of the entry as a “pblock”, so our driver will find it.
- A phandle to the accelerator instance. This is a reference to the Xilinx generated files that describe the PL design. This reference is useful to discover the configuration port of the accelerator.
- A phandle to the data mover that will serve this accelerator instance. It does not refer to the PL implemented AXI DMA but rather a virtual client to it, which we will define immediately after.

Finally, the declaration of the DMA clients. The “client” (in reference to the Linux DMA Engine API, not the DMA controller) defines a set of DMA controller channels that will be used by the API. The format of the node as seen in listing 4.11 is defined by this API.

The “dmas” property is used to reference the hardware. It is a list of pair values, with each pair describing a DMA channel. The first value is a phandle to the DMA controller instance and the second is the channel index within that DMA controller.

Here, “axi\_dma\_0” is the first instance of AXI DMA and it is a reference to the Xilinx generated description of PL design. In this description, the channel with index 0 will always be the transmit (MM2S) channel and index 1 will always be the receive (S2MM). The “dma-names” property is just a textual description of the DMA channels.

```

dma0: dma-client@0 {
    compatible = "tuc,dma-client";
    dmas = <&axi_dma_0 0
          &axi_dma_0 1>;
    dma-names = "tx", "rx";
};

```

**Figure 4.11:** DMA client definition

These entries, repeated for every instance of the resource they describe, is all the kernel driver needs to know about the hardware implemented in the PL. Should the hardware design change, the FDT must be revised to reflect the new design. The kernel driver should not need any modification.

### 4.5.2 Lying about the AXI DMA Interrupt Lines

As it was mentioned in section 3.5.1, we only use the S2MM interrupt output of AXI DMA. The MM2S is not connected and the DTS generation naturally will not produce any description for it. However, the Xilinx DMA driver requires it. As a workaround for this issue, we “lie” about its presence, assigning to it the interrupt line of the S2MM channel. This is done by modifying the generated .dts files that describe the PL part.

In listing 4.12 we see an example of AXI DMA IP core instance declaration along with its two channels. The first channel will always be the MM2S and the second one the S2MM. The “interrupts” property of S2MM (keyword in bold) was copied over the MM2S side (keyword in bold-italics).

```

amba_pl: amba_pl {
    #address-cells = <1>;
    #size-cells = <1>;
    compatible = "simple-bus";
    ranges ;
    axi_dma_0: dma@40400000 {
        #dma-cells = <1>;
        clock-names = "s_axi_lite_aclk", "m_axi_sg_aclk", "
            m_axi_mm2s_aclk", "m_axi_s2mm_aclk";
        clocks = <&clkc 15>, <&clkc 15>, <&clkc 15>, <&clkc 15>;
        compatible = "xlnx,axi-dma-1.00.a";
        interrupt-parent = <&intc>;
        interrupts = <0 29 4>;
        reg = <0x40400000 0x10000>;
        xlnx,addrwidth = <0x20>;
        dma-channel@40400000 {
            compatible = "xlnx,axi-dma-mm2s-channel";
            dma-channels = <0x1>;
            interrupts = <0 29 4>;
            xlnx,datawidth = <0x10>;
            xlnx,device-id = <0x0>;
        };
        dma-channel@40400030 {
            compatible = "xlnx,axi-dma-s2mm-channel";
            dma-channels = <0x1>;
            interrupts = <0 29 4>;
            xlnx,datawidth = <0x10>;
            xlnx,device-id = <0x0>;
        };
    };
};
/* ... */
}

```

**Figure 4.12:** Declaring an interrupt line for the MM2S



# Chapter 5

## Software Framework

One of the primary goals of this work is to offer an abstraction layer that hides all the hardware details from the end user. The software framework acts as an intermediary which on the one end orchestrates all control and communication of the FPGA part, while on the other, offers a simple software API ready to be used by a programmer who needs not to have any understanding of the underlying hardware.

The software framework consists of two parts. The major work is done by a kernel driver. It is responsible of the coordination of all system elements, from receiving the command to process data, to the delivery of the results. However, the kernel driver communicates to the userspace with the means of system calls and I/O control commands, which is not an appropriate interface for the end user. A separate system library was implemented, which is mostly used either as a wrapper, or to perform tasks that cannot (or should not) be done in kernel mode, namely the file operations to read the partial bitstreams.

The development was initially done in Xilinx PetaLinux, but later the work was ported to the Yocto Project framework, which is officially supported by Xilinx.

### 5.1 System Initialization

The kernel driver has two requirements: A working bitstream loaded on the FPGA, and a correct FDT that describes faithfully the hardware implemented in that bitstream. After system has initialized, further input comes from user requests via the system library.

These requests consist of a control command and, optionally, a partial bitstream.

The timing that this input is provided is important for system initialization. The FDT is supposed to describe existing hardware, and when the driver detects the hardware via FDT it will attempt to access it. Therefore, initial FPGA programming must precede the hardware declaration in the FDT if the kernel is running. This also stands true for the other drivers – for example, when the AXI DMA IP FDT entry is visible, the system will attempt to load the `xilinx_dma` driver which will attempt to initialize the IP core, assuming it is there.

Currently, there is ongoing work at the Linux implementation of the DT to make it more flexible for the reconfigurable hardware, FPGAs and modular single board computers. We will briefly discuss this in section 5.10 but for now let us assume that FDT is static. Therefore, the ideal solution would be to deliver the bitstream to the FPGA and the FDT to the kernel before the kernel boots, when the bootloader is running. Xilinx has patched U-Boot to allow it to program their FPGAs, and this is the recommended way.

When system has booted and the driver is loaded, it assumes it is the sole user of the FPGA. The end-user can interact with the driver either through the wrapper library or directly with `ioctl` commands. Partial reconfiguration should be done only via this interface, or it will go unnoticed and the system will behave as if the old accelerator was still present.

A system shutdown is achieved by removing the driver. The driver will release all the resources it claimed; memory, I/O space, DMA channels, etc. The user may reload the driver, but if the intent was to perform a new full reconfiguration, the `xilinx_dma` should also be removed.

## 5.2 The System Library

The system library, which functions as the interface to the end user, is implemented as a dynamically linked shared library. The library offers a user-friendly API to the end user. The API is comprised of two parts: the functions that affect the system as a whole and the functions that affect the work of the caller. The kernel may be configured to restrict the access to the first part at the root user only.

### 5.2.1 The System-Wide API

The system-wide API is comprised of the following functions:

```
int zdma_core_register(const char *name, signed char priority, unsigned long
    affinity);
int zdma_core_unregister(const char *name, unsigned long affinity);
int zdma_barrier();
int zdma_config(enum config arg);
int zdma_debug();
```

**Figure 5.1:** System-wide API functions.

The first function is called to load an accelerator core to the system, while the second un-loads it. All bitstreams are expected to be at `/lib/firmware/zdma/` directory, and the naming convention would be `<CORE_NAME>.<RP_IDX>.bin.xz`, where `<CORE_NAME>` is the accelerator core name (i.e. “sobel”) and `<RP_IDX>` is the reconfigurable partition index, an integer that matches the `zcore{16,32,64}_i` naming in the DT (see section 4.5.1). All bitstream naming and compression is done by the P.R. scripts (see appendix section A.4), while the decompression is done inside the kernel. There is no user intervention.

In both functions, the `name` parameter is the accelerator core name and `affinity` is an OR-mask of the reconfigurable partition indices that the core is permitted to execute on. The library will search for all bitstreams that match the aforementioned pattern for reconfigurable partition index range of 0 to  $8 * \text{sizeof}(\text{affinity}) - 1$  (that is 31 for Zedboard and 63 for zcu102), omitting the ones that are i) excluded by `affinity` ii) not physically present, either by purpose (e.g. to reduce storage requirements) or due to core variant unavailability for the specific reconfigurable partition in a heterogeneous design. The `priority` parameter affects the scheduler decisions and will be discussed in section 5.9. These functions are not just initialization and termination of a core variant; they may actually be called several times to dynamically adjust the bitstream availability to the kernel driver.

The `zdma_barrier()` flushes the work queue of all tasks of all users. It blocks until all tasks have finished. However, it does not block nor wait for any tasks that may have been queued after its call, potentially by another user/thread.

The `zdma_config()` is used to configure the system-wide operational parameters.

It may be used for experimentation or in case a specific access pattern is expected. In figure 5.2 a list of possible configuration commands is presented. As most of the commands affect the behavior of the memory allocator and the scheduler, in order to understand their function, one should first consult the respective sections.

Finally, the `zdma_debug()` function is used to output the state of the system. It is useful only during development.

### 5.2.2 The Task-Specific API

Here is the task-specific API. Note that despite the system registers the user that creates a task, in its present form it does not affect its decisions. Therefore, there are no user-related actions, all functions affect the task.

- `CONFIG_ALLOC_ZONE_DEFAULT`  
Set the memory resource selection algorithm. Currently only the default algorithm is available.
- `CONFIG_ALLOC_BITMAP_*`  
Set the allocation algorithm from within a specific memory resource. Possible choices are `FIRST_FIT` and `BEST_FIT`.
- `CONFIG_SECURITY_IOCTL_{ALLOW,BLOCK}_USER`  
Allow / Block unprivileged users to issue system-wide commands.
- `CONFIG_SECURITY_BUFFER_{KEEP,CLEAR}`  
Keep / Clear the contents of the task buffers after allocation and before deallocation.
- `CONFIG_SCHED_ALGO_*`  
Set the reconfigurable partition selection algorithm for the case eviction is not needed. Possible choices: `FIRST_FIT` and `BEST_FIT`.
- `CONFIG_SCHED_VICTIM_ALGO_*`  
Set the victim reconfigurable partition selection algorithm. Possible choices: `FIRST` (First-Available), `LP` (Least Popular), `First-Programmed FIFO`, `LRU` (Least Recently Used) and `LFU` (Least Frequently Used).

**Figure 5.2:** Valid configuration commands.

```
int zdma_task_init(struct zdma_task *task);
int zdma_task_configure(struct zdma_task *task, const char *core_name,
    unsigned long affinity, int tx_size, int rx_size, int argc, ...);
int zdma_task_enqueue(struct zdma_task *task);
int zdma_task_enqueue_nb(struct zdma_task *task);
int zdma_task_waitfor(struct zdma_task *task);
void zdma_task_destroy(struct zdma_task *task);
```

**Figure 5.3:** User task API functions.

A task's lifetime begins with the `init()`. The library opens a file descriptor with the driver's `/dev` entry, which simply registers the task. No resource allocation happens at this time except for the task control structures.

The `configure()` is responsible for all reservations that take place and therefore it can easily fail, so return value must always be checked. The `core_name` parameter is self-explanatory, but it should be noted that the core must be already registered using `zdma_core_register()`. The `affinity` parameter is similar to the one of core's registration, the only difference is that now it is enforced in per-task basis. A user may define a task affinity that allows execution on an accelerator slot where the core is disallowed to be placed. However, if the combined affinities, further restricted by bitstream availability, lead to a void slot set, the request will fail. The `tx_size` and `rx_size`, the transmit and receive buffer respectively, must be fulfillable by at least one memory zone. Finally, `argc` is the number of accelerator variant configuration parameters. If non-zero, the parameter arguments must follow `argc`. If the number of supplied arguments does not match with `argc`, a software undefined behavior will occur. If the list is valid but not legal for the specific core, a hardware undefined behavior will occur. Specifically for our application, if the user does not specify a legal value for line width, it will default to 1080.

The `enqueue()` and `enqueue_nb()` will place the configured task to the execution queue. Their difference is that, in case the previous execution of the same task is not yet complete, the former will block whereas the latter will return an error. Neither will block for the current task execution – one should use `waitfor()` for this.

Finally, the `destroy()` will release the task resources. It implicitly calls `waitfor()` to terminate the task gracefully.

### 5.3 Communicating with the Hardware

One of the most frequent reasons for implementing something in kernel space, is the need to communicate directly to the hardware. In processor architectures that support hierarchical protection domains, the hardware is exposed only to code running in a privileged mode, typically the operating system. If the user application requires access to a device, it has to call the operating system to execute on its behalf. This way, the system enforces security policies and offers hardware abstraction.

Since for our work we do need to manipulate hardware directly, we have to do it from within the kernel. Programming the kernel however, is a challenging feat. It is a completely different environment, lacking all the tools and libraries any user is familiar with – including the standard C library. Many everyday operations that are taken for granted, like file I/O or floating point arithmetic, are not allowed or at least greatly discouraged. Debugging is much more restricted and complicated while many of the system safety nets are not present – a programming error is not isolated at the running process and its resources, but may affect the whole system. Last but not least, modifying kernel code is not as direct as it is in the userspace.

A naive workaround at these apparent difficulties is offered through the `/dev/mem` Linux device. It consists a direct interface to the machine's physical address space and it can be mapped to a user virtual space. Its primary use is for debugging the system and is always disabled in production systems, but many hardware engineers find it convenient, since it allows them to program a Linux based FPGA SoC as if it was a bare-metal environment, with the only restriction that interrupts cannot be detected. Nonetheless, this solution was rejected on principle, as it negates the operating system's *raison d'être* – security and hardware abstraction.

A much more “clean” solution is given by the Linux Userspace I/O system. Written with industrial I/O cards in mind, this system allows the control of an interrupt and memory capable device using only a minimal device driver that just declares the device's hardware resources. All control and data processing can be done at the userspace using the tools and libraries the programmer is familiar with. The device memory resources can be mapped to the virtual address space and its interrupts may be detected with the use of blocking `read` or the `select` system call. This solution can be made even more attractive from the fact that Vivado HLS is able to generate the aforementioned mini-

mal device driver automatically during IP packaging. However, UIO may be ideal for simple I/O devices but it is too restrictive and inflexible for anything more complicated. Since the control logic is written in userspace, the programmer loses all access to kernel facilities and data structures that are usually necessary for more complex tasks involving other kernel subsystems. A degree of uncertainty on whether or not UIO is a feasible and efficient solution led to the decision of implementing the core system entirely in-kernel.

## 5.4 Performing DMA from the kernel

In a bare-metal environment one would program a PL peripheral by manipulating its control / status register. This is also entirely possible in an operating system, and some times it might be the only way. However it has two major downsides. The first is that it sacrifices portability, as the programming sequence of an IP core is fundamentally dependent on the specific hardware. A newer core revision, a different core configuration, a different host platform, or even worse, an IP core from a different provider, may have significantly different programming interface. For the designer, this means that they have to learn the low-level programming details of the IP core, and in case it needs to be updated or replaced, they must put a significant effort to port the software to the new version. The other downside is that direct manipulation of hardware control registers bypasses all OS subsystems that may offer useful support functionality and integration with other kernel services.

The Linux kernel supports myriads of DMA controllers on the several computing platforms it is ported. Likewise, there are several kernel subsystems that wish to use these DMA controllers. In order to offer abstraction, the “DMA Engine API” is offered. The API has a “memory-to-memory” part as well as a “Slave DMA” one. The Slave API provides hooks where the DMA controller vendor may register their backend driver specific to their hardware. Following this approach, Xilinx has written a backend driver for AXI DMA, CDMA and VDMA drivers, called “`xilinx_dma`”. When an API client attempts to reserve a DMA controller channel, it specifies explicitly which hardware resource it needs, through the use of DT, so the kernel knows how to pair the client with the correct backend driver.

Essentially, the steps to program the AXI DMA core are the following:

1. Allocate DMA'able memory.
2. Request both the MM2S and the S2MM channel of an AXI DMA instance using `dma_request_channel()`.
3. Create a transaction and get a descriptor from both channels using `dmaengine_prep_slave_single()`.
4. Submit the descriptor to the driver queue with `dmaengine_submit()`.
5. Force queue processing with `dmaengine_issue_pending()`.
6. Wait for transaction completion.

Now, some questions may come naturally: What is DMA'able memory and how is it allocated? How can one choose which DMA controller and channel to request? How can one know when a transaction is complete? We will discuss these questions one by one.

### 5.4.1 Allocating DMA'able Memory

A “DMA'able memory” is a memory region where a slave DMA controller may perform DMA transfers. There are two criteria that must be fulfilled:

Firstly, the DMA controller and the system bus that connects it, must be able to generate addresses for the buffers. The AXI DMA IP by default generates 32 bit addresses and the AMBA bus is able to support them. So, for Zynq, which features 32-bit ARM cores, we do not need to do anything. For ZynqMP, which is capable of 40-bit addressing, if one desires to take advantage of it they must configure AXI DMA accordingly and also use the `dma_set_mask*()` family to set up the proper DMA address mask, as it defaults to 32 bits.

Secondly, the region must be physically contiguous. A user-space allocation with `malloc()` is not, as is the case with its kernel counterpart, `vmalloc()`. A special case is `kmalloc()`. This function returns physically contiguous memory mapped at a linear

address\*. However, the maximum allocation is currently limited to 4MiB<sup>†</sup> for ARM. This is a significant limiting factor, as even for a single DMA transaction, the AXI DMA is capable of moving 8MiB of bytes.

As this was a long-standing issue in all architectures, there have been many workarounds, especially in the past, where the maximum allocation was only 128kiB. On systems equipped with an IOMMU, one may setup a contiguous bus address space that can be discontinuous in the physical space. This could be an option for UltraScale+ but not for Z-7000. Another approach is to use scatterlists, a software construct in the Linux kernel that abstracts the Scatter-Gather functionality. Despite that this does work, it adds an unnecessary and significant overhead in computation and implementation effort. A simpler solution would be to keep some memory out of kernel's reach and manipulate it manually. This can be done by either a kernel command line parameter or by reserving the memory in early boot, before the buddy allocator is started. Apart from consisting not-so-nice trickeries, they also inhibit the system from ever using this memory even if we do not actually use it.

To overcome all these issues, a new kernel facility was introduced in 2012. The Contiguous Memory Allocator (CMA) allows the allocation of indefinitely large physically contiguous memory. The memory assigned to the CMA will be lent to the buddy allocator under the precondition that it may not be pinned. Should the CMA need these pages back, the buddy allocator will migrate away the offending pages and will return them to the CMA. This way, the memory assigned to the CMA is still available for general usage until it is requested. This also permits changing the size of the reserved memory without the need of a reboot.

The CMA is integrated at the DMA Engine API and is automatically called when using the `dma_alloc_coherent()`. This function performs two actions – it allocates the requested DMA-able memory amount, and also creates a coherent mapping. Coherent (or consistent) memory is a memory that can be accessed by both the CPU and the device without caching side-effects. In an architecture that offers two-way cache coherency, this is a normal mapping, but in our case, this is ensured by marking all the

---

\*The linear address is the kernel's mapping of the physical address space to a virtual address space whose address is a constant offset from the physical.

<sup>†</sup>The maximum allocation order, `KMALLOC_SHIFT_HIGH`, is defined at `linux/slab.h` as `min(25, MAX_ORDER+PAGE_SHIFT-1)`, where `1u<<PAGE_SHIFT` is the page size, defined at `asm/page.h`, and `MAX_ORDER` is the maximum allocation order of the buddy allocator, defined at `linux/mmzone.h`.

allocated pages as uncacheable.

Still, we are not done yet. By using the CMA alone, we can receive an arbitrarily sized DMA capable buffer that is aligned at a boundary set at kernel configuration time. However, we cannot force the exact physical placement of the buffer. This is a critical requirement, as the physical address defines which PS-to-PL port will be used, which in turn leads to a certain AXI interconnect. Effectively, each accelerator may be reachable from different physical address regions and by controlling the physical address we balance the traffic between the interconnects. Traditionally this effect was achieved the same way contiguous memory was reserved – by excluding it from any kernel access, which comes with the disadvantages already discussed.

A newer, cleaner solution is now available with the combined usage of the CMA and the DT. In section 4.5.1 it was shown how a memory bank is described in the DT file. This file is loaded by the first-stage bootloader and passed to the Linux kernel and can be manipulated by the OpenFirmware support functions. So, what has to be done is that the DT be traversed to reach the node that describes the memory bank, and from there one must use the `of_reserved_mem_device_init_by_idx()` to instruct the CMA that the subsequent `dma_alloc_coherent()` will use the specific memory pool. The CMA already has this memory under its authority as we have already described it as reserved (see listing 4.8 at section 4.5.1). A final detail would be that in fact, the assignment of reserved memory regions is done by Linux on device basis. A single device shall not have multiple active reserved memory regions. To overcome this, the driver declares a pseudo-devices for every memory bank (or “zone”) defined, and the reservation is instantiated on its behalf\*.

## 5.4.2 Controller and Channel Selection

The selection of the hardware resource that will perform a DMA transaction is done with the help of the DT. Let us recall the accelerator slot declaration, listing 4.10 at section 4.5.1:

---

\*The technique was exemplified by the Samsung Exynos Multi-Format Codec to support parallel memory access for left and right audio channel. See `s5p_mfc_alloc_memdev()` at `drivers/media/platform/s5p-mfc/s5p_mfc.c` on recent kernels (4.8+).

```

pb0: pblock@0 {
    compatible = "tuc,pblock";
    core = <&zcore16_0>;
    transport = <&dma0>;
};

```

**Figure 5.4:** Reconfigurable partition definition

We see that an accelerator slot may be served by only a specific DMA controller, defined with the `transport` property. This is logical, as we chose to dedicate one DMA controller for each accelerator. The `dma0` is a phandle, that is, a reference to another DT leaf. If we dereference it, we will obtain the DMA client definition, which is what `dma_request_channel()` needs to know. Let us recall however, that the DMA client definition is not the actual hardware definition. It just states our intent to use the hardware, which is pointed by the DMA client description.

### 5.4.3 Termination of a DMA Transaction

The AXI DMA offers two ways to signal the completion of a DMA transaction. One may poll its `S2MM_DMASR`, the receive channel status register, to find out if the channel has reached an idle state. Additionally, the DMA controller has two interrupt outputs, one for each channel, which we have driven to the interrupt inputs of the Zynq APU. The standard procedure would be to wait for an interrupt on the `S2MM` channel, and when received, check the status register for a possible error condition.

Fortunately, these are handled by the Xilinx back-end driver which is hooked at the DMA Engine API. The completion notification functionality is provided by the Linux Asynchronous Transfer API, or simply “`async_tx` API”. Technically, this API is a client to the DMA Engine API, however it is now integrated with it.

The `async_tx` API is frequently used with the “completions” synchronization mechanism. As soon as a DMA transaction is issued using `dma_async_issue_pending()`, the programmer calls `wait_for_completion_timeout()` which will block until either a `complete()` is called or the timer has expired. Completions are implemented using work queues, which in turn are implemented with semaphores, but the indirect use is more readable and less error prone.

But who is going to call `complete()`? The `async_tx` API has a transaction descriptor, which we get when we call `dmaengine_prep_slave_single()`. This descriptor has a field named `callback`, a pointer to a function that is executed when the transaction is complete. This is an ideal place to call the `complete()`, waking up the blocked kernel thread.

When code flow is resumed, the programmer may check two conditions: If the timer has expired and if the DMA controller has reported an error. The latter is retrieved by `dmaengine_tx_status()`.

## 5.5 Zero-Copy Transfers

Traditionally, a DMA transfer between a device and a user application was done through bounce buffers. A bounce buffer is an intermediate memory buffer that resides in kernel space where all data are temporarily stored before being sent to the device or the userland. This technique arose for several reasons. Some older buses or devices might not offer a sufficiently large DMA'able address space. Some 32-bit architectures offer the notion of high memory, where not all physical address space may be mapped at the same time, a property that is a requirement for DMA. The difficulty of allocating large physically contiguous memory contributed to the problem. And finally, it is also the simplest way of dealing with protected memory.

The use of intermediate buffers has three strong drawbacks: The increased latency, the decreased bandwidth and the doubling of memory footprint. The first issue may be alleviated by using multiple buffering, but it does not help the other two. The advent of IO-intensive peripherals like the GPUs necessitated a solution to this bottleneck.

In our target systems we have all the hardware support we need to avoid any buffering. The AXI DMA IP core can be configured to support the full address space of both Zynq-7000 (32 bit) and UltraScale+ (40 bits). Neither the ARM A9 cores in the former nor the A53 ones in the latter support Large Physical Address Extensions (LPAE), ARM's implementation of high memory. At the software front, the support of the Open Firmware's Device Tree and the creation of Contiguous Memory Allocator, eliminated any remaining software obstacles.

The support for zero-copy DMA transfers could be incarnated in two forms. Either by mapping the user memory to the kernel address space or by mapping a kernel buffer

to the calling process' virtual address space. The former is already there, as the installed memory in both platforms is small enough compared to the address space so it is fully and permanently mapped as the kernel's linear address space. However, userspace allocations are not physically contiguous and the overhead of working around this through the use of scatterlists is not worth the effort. Conversely, mapping a kernel buffer to the userspace is a straightforward process.

The allocation is done in two steps. First, the user application will inform the kernel about its desire to acquire memory buffers. The kernel will attempt to make a reservation from the memory allocator and report the result. If successful, the application will issue a `mmap()` system call for each buffer. The kernel will create a coherent mapping of the buffer it previously created to the virtual space of the calling process and hand over the address.

The mapping is done with `dma_mmap_coherent()` and not directly by `remap_pfn_range()`. This is because before performing the actual remapping, the kernel must set the protection bits of all pages to non-cacheable, so the new mapping will also be coherent. Recall that we initially reserved the memory from the system by `dma_alloc_coherent()` for the exact same reason.

## 5.6 Security and Error Handling

A hardware fault may reduce the system's functionality, i.e. a certain accelerator slot may become inoperable, or even make it fail completely. A proper failure path was established in order to roll back any half-done operations. The Managed Device Resource (`devres`) was used, a framework that guarantees that any reserved resource can be later reclaimed. Overall, it is very likely that normal operation can be restored, in the worst case by unloading and re-inserting the kernel module.

Nonetheless, there is at least one known weakness: If the hardware designer has not implemented the AXI-Lite control interface of an accelerator, a data abort will occur at the first configuration attempt and will cause the kernel to hang. It must be added that, in case the hardware design does not meet timing, the AMBA bus may hang, causing the whole system (PL and PS) to become inoperable. There is nothing that can be done to counter it.

A lot of effort is put to make the system foolproof, in the sense that it will not

fail by a user mistake. Furthermore, the system offers basic security and isolation to prevent leakage of data between user tasks. However, the system does not implement user quotas and therefore it is vulnerable to a DoS attack. A malevolent non-root user can create a large number of small tasks that will eventually deplete all reserved memory, denying access to any new user until the system is restarted by removing and re-inserting the kernel module. In a more simple case, a greedy user is able to abuse the task priority mechanism to dominate processing power.

## 5.7 Configuring the Accelerators

Configuring the accelerators is quite simple. By using the AXI-Lite, the accelerator registers are all mapped to the APU address space. Once we create a virtual mapping, we can use load/store instructions to query/modify them.

At the listing 5.4 of the previous section, we saw how an accelerator slot is defined. There, the property “core” was defined. This is a reference to the accelerator instance. The record for the accelerator is contained in the automatically generated part of DT, using the hardware information file (.hdf) extracted from the static workflow. However, since the partial reconfiguration workflow requires the interface to remain consistent, the record would be valid for all accelerator variants. The record would be like this:

```
zcore16_0: zcore64@43c00000 {
    compatible = "xlnx,zcore16-3.7";
    reg = <0x43c00000 0x10000>;
    xlnx,s-axi-control-addr-width = <0x6>;
    xlnx,s-axi-control-data-width = <0x20>;
};
```

**Figure 5.5:** Accelerator instance definition.

The value of interest is the `reg` property. The first part is the base of the address space and the second is its length. The internal organization of this address space is described in section 6.2. We see that the control and status signals are at stable positions of the control register at the offset 0x00. The accelerator-specific registers start at 0x10 with a step of 8 bytes. However, the argument count and the data represented are

strictly accelerator variant specific and cannot be retrieved by neither the hardware nor the kernel module as they have no prior knowledge of the logic they will execute. This information must be passed at runtime, when the end user registers the accelerator core to the system.

To sum up, the communication to the accelerator follows these steps:

1. Initialize the control/status register (CSR, 0x00) to zero. This de-asserts `ap_start` and `auto_restart`.
2. Read back the CSR. The `ap_idle` should be asserted. Abort if not.
3. Write all the user arguments at the corresponding offsets.
4. Assert the `ap_start` of the CSR.
5. Issue the DMA transaction and wait for completion.
6. Read the CSR, `ap_done` should be asserted. Abort if not.
7. Read the accelerator return code at register with offset 0x38.

A final note would be that the programmer shall not use the pointer dereference operator to access the registers. Instead, they must use the Linux kernel provided functions `ioread32()` and `iowrite32()`. The functions are for I/O memory and they will disable write-combine and enforce the proper memory barriers.

## 5.8 The Memory Allocator

In section 5.4.1 we saw how a set of pre-defined memory regions can be reserved and allocated to the kernel module. The allocation takes place only once during module initialization and its management is passed to a custom memory allocator which is charged with fulfilling accelerator requests.

Although this functional segregation reduces memory efficiency as it prevents CMA from lending unused memory to the buddy allocator, it offers multiple advantages:

1. Reduction of run-time latency, as the page frames are guaranteed to be unused, no page migration is going to happen.
2. Elimination of uncertainty. Although the buddy allocator will not make any borrowed page frame unmoveable, kernel code may freely do so.
3. Control of allocation range. In our system our memory resources are not required to be equal. Furthermore, they are not accessible by all accelerator slots and if they do, they may not use a path of similar latency. To handle all this heterogeneity, we need an allocator that is aware of the implemented hardware details.
4. Flexibility. Although currently not implemented, it would be useful if the system could re-balance the memory bank utilization migrating pages between memory resources. This can be done much easier if we have full control of all the available space.

The allocator executes in two stages. At first, it will attempt to find the most suitable zones that fulfill all requirements for the transmit and receive buffer. After a selection was made, it will attempt to reserve the requested amount of memory. Initially, the allocator was implemented using the `genalloc` subsystem. This subsystem is intended for allocation of special-purpose memory resources, typically from on-board SRAM or OCM of embedded devices. It was a quick and easy implementation but the subsystem was too restrictive for our purposes as it does not differentiate the memory zones and their selection was only on a first-fit basis. The subsystem was patched to allow custom algorithms for zone selection and zone metadata were held at the kernel driver. However, it was an unclean solution that became difficult to maintain and extend. It was decided that the first stage should be re-written to our needs while keeping its original concept. The second stage that involves the per-zone bitmap manipulation continues to use the `genalloc` library.

In figure 5.6 we can see an overview of the memory allocator function. A reservation always takes place in pairs: a transmit and a receive buffer, one of them can be of zero size for a transmit-only or a receive-only function\*. If the calling process already holds

---

\*Note that this operation mode is not tested.

buffers, they will be deallocated. When allocating or de-allocating a buffer, the “popularity” of the memory zone, a metric needed by the scheduler, is updated accordingly. For brevity, both functions are not represented in the diagram. Initially, the allocator will attempt to reserve the transmit buffer, and afterwards, the receive. If the latter fails, it will try the allocation in the reverse order. This is because the scoring algorithm, in critically low memory situations, may suggest a memory zone for one buffer that renders the other unplaceable due to external fragmentation. Reversing the order of allocation may resolve this issue.

Figure 5.7 depicts the algorithm of the actual allocation. The inner loop scans the list of memory zones and calculates a score metric. The default scoring algorithm is given by:

$$score = \frac{ZoneBandwidth \cdot (MemoryAvailability/8 + Schedulability)}{1 + ZoneOccupancy}$$

where *ZoneBandwidth* is a designer-supplied memory zone parameter, defined in the zone description in the DT. Although it was implemented to compensate for the actual interconnect bandwidth, it actually needs not to match any literal hardware specification – it is rather a means for the designer to affect the desirability of one memory zone versus the others. The *MemoryAvailability* is the integral ratio of the memory zone size to the requested memory amount while *ZoneOccupancy* represents the number of clients that have a reservation on this memory zone, irrespectively of the amount of memory allocated. These parameters aim to spread the memory utilization among the zones, under the assumption that they are served by different interconnects, therefore balancing the data traffic at the underlying hardware. The *Schedulability* represents the scheduler freedom to place the task if this zone is chosen. It is quantified as the population count (i.e. the number of ones in a word) of the *SchedulabilityMask*, which we will define as the AND product of the following three masks:

- **Core Availability**, that represents the bitstreams that are available to the kernel driver for the specific core variant. In turn, this is a product of two factors:
  - **Bitstream Availability**, which refers to the physical availability of the bitstreams on the storage medium.

- **Core Affinity**, which refers to the system administrator’s decision to restrict a core variant placement to a set of accelerator slots. See the `affinity` parameter of `zdma_core_register()`, section 5.2.1.
- **Task Affinity**, which is the client’s request to constrain the slots where their task may run. See the `affinity` parameter of `zdma_task_configure()`, section 5.2.2.
- **Partition Affinity Mask**, which refers to the physical interconnect constraints that restrict core placement. It is determined by two factors:
  - **First Interconnect Path**. Each memory zone has defined a reader and a writer mask at their entry in the DT, defined by the interconnect architecture. If we are allocating a transmit buffer we may select a slot from the reader mask and, conversely, in receiver buffer we are constrained by the writer mask.
  - **Second Interconnect Path**. After we choose a memory zone for one direction, our choice for the other will not only be constrained by the specific direction’s mask, but by its intersection with the other mask. A core can be scheduled only in a slot that can both read from the transmit buffer and write to the receive buffer.

The score function will always return zero if either *MemoryAvailability* or *Schedulability* are zero.

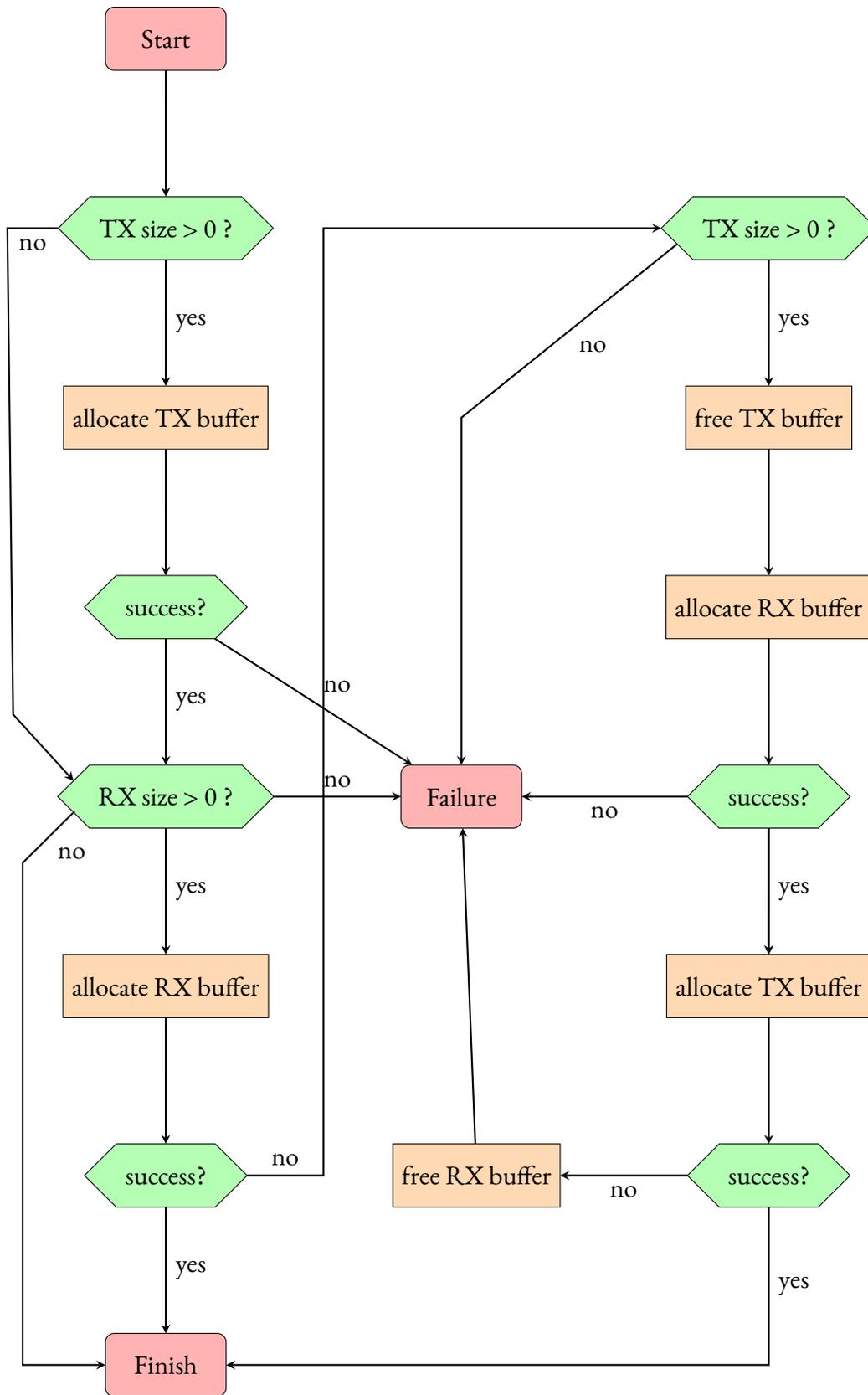
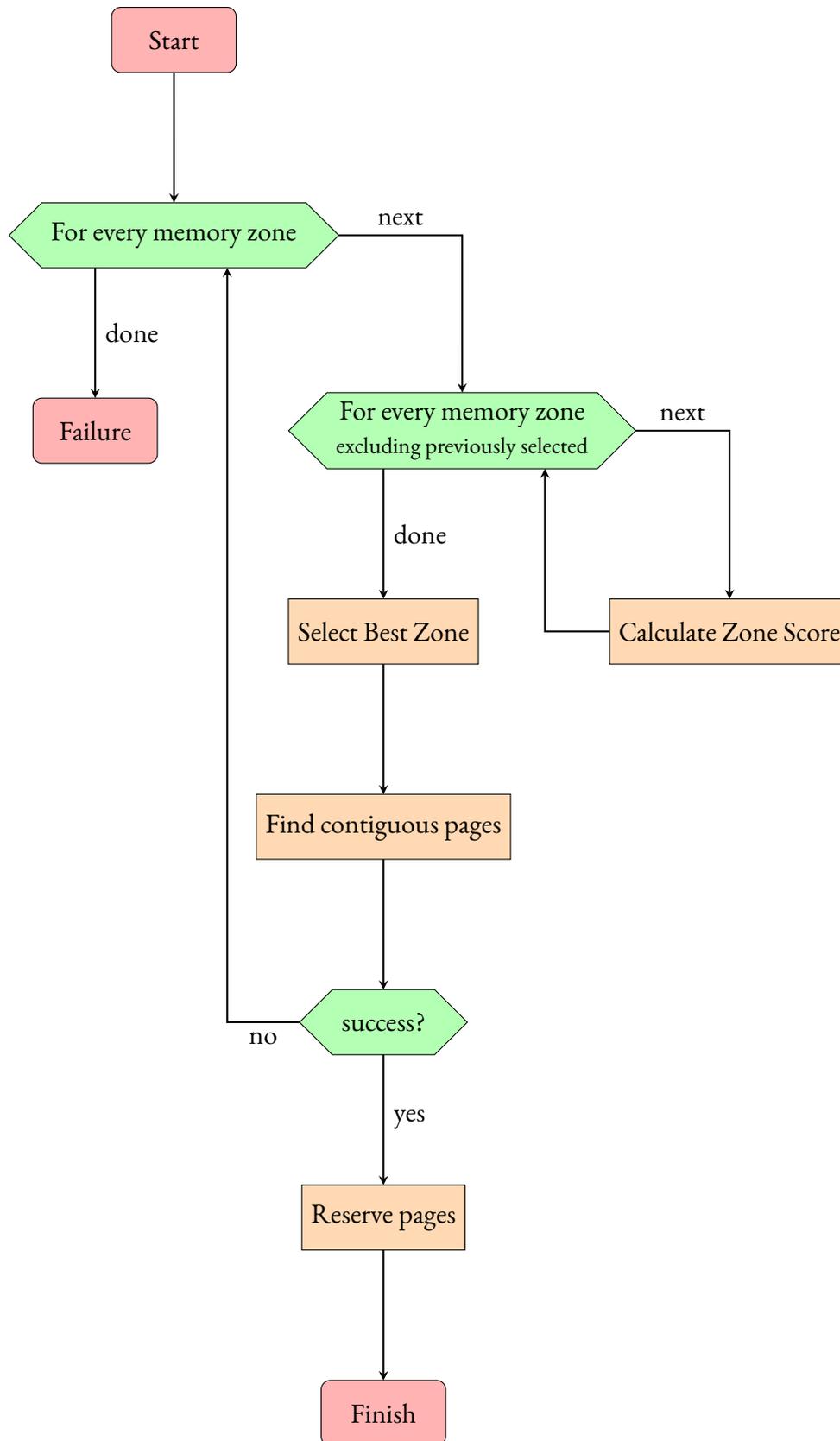


Figure 5.6: Memory Allocator: Overview

**Figure 5.7:** Memory Allocator: Making a Reservation

## 5.9 The Scheduler

The system's scheduler is charged with finding a suitable accelerator slot for a task wishing to execute. The scheduler is completely dynamic; it will accept the queuing of a task at anytime and possess no knowledge of future requests. The scheduler queue is implemented using the Concurrency Managed Workqueue API of the Linux kernel. Using this API provides a safe and well-tested work queue implementation that automatically handles synchronization and forward progress. Furthermore, the API makes use of pre-allocated kernel worker threads, which eases memory pressure on resource constrained environments.

The work queue is not strictly FIFO, although it can be made so with trivial source modifications\*. When the user enqueues a task, it is placed in the work queue to be processed asynchronously. A task execution includes the following steps, listed in order of execution:

1. Call the scheduler to get an accelerator slot. The scheduler may instruct a slot reconfiguration, therefore the call is possible to block for a few milliseconds but will always return successfully.
2. Query, configure and start the accelerator.
3. Configure the transmit and receive channels for the new transaction.
4. Submit the transaction descriptors to the DMA Engine API.
5. Configure the completion notification (async\_tx API).
6. Force issue of all pending DMA transactions.
7. Block waiting for completion (or timer expiry).
8. Check DMA return status.
9. Check the accelerator return value.
10. Release the accelerator slot.

---

\*The creation of the work queue by `alloc_workqueue()` should enable the `WQ_UNBOUND` flag and set the maximum active workers to 1.

Depending the system load and work queue configuration, the kernel will wake up a number of threads to process the work queue. Since this number is not predictable (only upper bounded) the scheduler does not make any assumptions and take all necessary synchronization measures to ensure atomicity and critical region locking. The scheduler pseudocode is presented below, in algorithm listing 1.

We can see that an execution loop consists of two attempts to reserve a slot, each passing a different bitmask as an argument. The bitmask represents the slots that *ReserveSlot()* is allowed to consider. The common term, *Task.EffectiveAffinity*, represents the user defined task affinity further constrained by the interconnect restraints (see Task Affinity and Partition Affinity Mask in section 5.8).

The first attempt used the additional mask *CoreMask*, which we construct by placing a 1 wherever the already programmed core (*Slot.Core*) is the same with the one the task intends to use (*Task.Core*). Essentially, this restricts the search space only the slots that will not require reconfiguration. If the search is unsuccessful, the second attempt will be with the *Task.Core.Availability* mask, i.e. all the slots that have a bitstream available for this core variant, which after constrained by *Task.EffectiveAffinity* would be equal to *SchedulabilityMask*, i.e. all the possible legal slots for this task. Of course, this would require us to re-configure the slot and suffer the corresponding time penalty.

The *SelectSlot* is actually a function pointer, and the selection algorithm can be configured by the system administrator as we saw at the figure 5.2 at section 5.2.1. The functions that implement the two calls are common, however the configuration options are separate since the context is a different and as most are replacement algorithms, they have no purpose in the first stage. A detailed list of algorithms implemented is given below, of whom only the first two are for the first stage selection while all are offered for second stage (victim) selection.

- **Least Popular** represents the slot where the smallest number of tasks can be scheduled into. The system maintains a popularity counter for each slot and it is calculated by counting the tasks for whose their *SchedulabilityMask* contains that slot. A slot with low popularity is expected to not be claimed frequently as is suitable for fewer tasks. In a way, it is a form of an *a priori* variant of LFU.

Being the least desirable is likened to being the “smallest” and by this analogy is used by the Best-Fit configuration option of the first selection.

- **First Fit / Available** is the simplest algorithm, as it returns the first free and suitable slot.
- **Lowest Priority** algorithm will return the slot with the lowest priority (least important, therefore best victim).
- **FIFO** will chose the oldest slot, counting from the time of reconfiguration. The time ticks at every scheduler call.
- **Least Recently Used** is similar to FIFO, but counts from the last time it was chosen by the scheduler.
- **Least Frequently Used** uses as a measure of period the ratio of the time since the slot was last chosen over the “hit counter” which is incremented every time it is chosen in the first stage, i.e. for re-use with no reconfiguration. The counter is initialized to 1 for an empty (never configured) slot and to 2 during reconfiguration – there is no persistence between reconfigurations.

**Algorithm 1** The Scheduler algorithm

---

```

1: procedure SCHEDULER(Task, AcceleratorSlots, Config)
2:   for all Slot  $\in$  AcceleratorSlots do            $\triangleright$  needed by most replacement algo
3:     if Slot.Core  $\neq \emptyset$  then
4:       Slot.CreationAge  $\leftarrow^+$  Slot.Core.Priority
5:       Slot.AccessAge  $\leftarrow^+$  Slot.Core.Priority
6:     end if
7:   end for
8:
9:   repeat
10:    CoreMask  $\leftarrow$  0
11:    for all Slot  $\in$  AcceleratorSlots do
12:      if Slot.Core = Task.Core then            $\triangleright$  slot is already programmed
13:        CoreMask  $\mid =$  Slot.id
14:      end if
15:    end for
16:    SelectedSlot  $\leftarrow$  ReserveSlot(CoreMask
17:      and Task.EffectiveAffinity)
18:    if SelectedSlot  $\neq \emptyset$  then
19:      if TryLock(SelectedSlot) then
20:        Slot.HitCount++                          $\triangleright$  needed by LFU algo
21:        break
22:      else
23:        SelectedSlot.State  $\leftarrow \emptyset$ 
24:      end if
25:    end if
26:    SelectedSlot  $\leftarrow$  ReserveSlot(Task.Core.Availability
27:      and Task.EffectiveAffinity)
28:    if SelectedSlot  $\neq \emptyset$  and TryLock(SelectedSlot)=FALSE then
29:      SelectedSlot  $\leftarrow \emptyset$ 
30:    end if
31:
32:    if SelectedSlot  $\neq \emptyset$  then
33:      Reconfigure(SelectedSlot, Task.Core)
34:    else
35:      YieldProcessor                              $\triangleright$  wait for progress to be made
36:    end if
37:  until SelectedSlot  $\neq \emptyset$ 
38:  SelectedSlot.AccessAge  $\leftarrow$  0
39:  return SelectedSlot
40: end procedure

```

---

## 5.10 Partial Reconfiguration

The mechanism of performing FPGA configuration, partial or not, is undergoing major restructuring as of the time of writing. There are two FPGA programming interfaces supported by Xilinx, none of them fully.

The older interface is `devcfg`. It is a Xilinx driver which implements a character device. The user may perform read/write operations to the appropriate device file from userspace and a degree of configuration is possible through `sysfs`. For example, to perform partial reconfiguration from a Linux console, one should type

```
> echo 1 > /sys/class/xdevcfg/xdevcfg/device/is_partial_bitstream
> cat bitstream_file.bin > /dev/xdevcfg
> cat /sys/class/xdevcfg/xdevcfg/device/prog_done
1
```

**Figure 5.8:** Programming a partial bitstream to an FPGA from Linux console.

The driver will read the contents of the bitstream (only in `.BIN` format, the `.BIT` is not supported) and will use the PCAP DMA to transfer it.

There is a critical design flaw with this programming interface. It is a very basic and purpose-built driver, made to address the need of programming the FPGA from the command line, and nothing more. It offers no kernel API at all and therefore it cannot be integrated with any Linux subsystem. Of course, it is Xilinx-specific and completely incompatible with any other vendor.

As both major FPGA manufacturers, Xilinx and Altera/Intel offer FPGA SoCs and are trying hard to penetrate the supercomputing sector, the need of a complete and standardized interface between an FPGA and the Linux kernel became apparent.

In 2015, a novel approach was presented at ELC [27], the “FPGA Manager Framework”. The framework abstracts the FPGA to the Linux kernel, offering a stable API which is vendor and hardware agnostic. However, the FPGA Manager is not just a hook that FPGA vendors attach a driver function; it is integrated to the kernel and it makes it aware of the FPGA presents. It abstracts the notion of the FPGA, the reconfigurable partition, the isolation logic. Furthermore, a parallel development in DT support, the Device Tree Overlay (DTO)s [28], enabled the dynamic modification\* of a live FDT

---

\* The Linux kernel already supported dynamic FDT update, albeit in a very basic level and used

and the automatic reaction of the Linux bus driver model (i.e. associated device probing, removal, etc). Although the work primarily targeted the BeagleBone and its removable extension boards, it permitted the FPGA Manager to provide automated handling of FPGA state change, e.g. partition isolation or driver probing/removal upon full or partial reconfiguration.

The FPGA Manager was admitted to the mainline kernel recently (2016, v4.4) whereas DTOs were accepted a bit earlier (2015, v3.19). At the time of writing, the most recent developments may be found at [29]. Support for Intel and Xilinx SoC FPGAs is here, with ZynqMP not yet mainlined. PCIe attached FPGAs are not yet supported but support is worked on. At the vendor front, both Altera/Intel and Xilinx support the framework. Xilinx has deprecated the `devcfg` and has completely removed it from its latest kernel. Xilinx offers an FPGA Manager backend driver for both Zynq and ZynqMP, albeit the driver lacks partial reconfiguration functionality for ZynqMP as it is not yet supported by the `xilinxfpga` library which implements FPGA programming through PCAP. Although it is expected to change soon, currently it seems there is no way to perform partial reconfiguration in ZynqMP from the PS under either Linux or bare-metal. This blocks our porting effort for this platform.

Xilinx proposed `devcfg` for partial reconfiguration in Zynq and it is this interface it uses on both partial reconfiguration application notes [30] and [31]. Combined with the at that time lack of proper understanding of FPGA Manager's nature, lead us to use `devcfg`, a choice that only much later became apparent that it was a mistake.

### 5.10.1 Using the `devcfg` Interface

First of all, using `devcfg` means that the kernel will not be aware of the presence of the FPGA. That could be a significant problem if any bus master peripherals resided in the reconfigurable partition as we would need to find a way to properly initialize and de-initialize their drivers externally. Thankfully this was not the case, as the accelerator that resides inside the reconfigurable partition is a slave to its corresponding AXI DMA, which we take care to manipulate properly in our driver.

The major issue with `devcfg` is that it has no kernel API at all. Even worse, its

---

only in very specific corner cases. It does not offer any advanced feature that DTO offer; modifications are not atomic, cannot be reverted and the bus driver model does not react to the changes.

structure does not permit an easy modification for external linkage, as everything is done within the implementation of `read()` file operation.

Accessing a file from kernel space is technically possible but it is a practice always frowned upon. Furthermore, the reconfiguration time could be affected. If the filesystem resides in an SD card, the file access time would dominate the reconfiguration process. File caching would help, but performance would be unpredictable.

The idea of a cooperating userland library was considered, but was not realized as, despite it is a safer approach, it is complicated and it still suffers the same, in fact worse, performance penalty.

In order to bypass this obstacle, a not so clean workaround was employed. The `devcfg` driver implementation of the `read()` system call was broken down to its preamble (accessing the file data), its main body (performing the reconfiguration) and epilogue (clean up), and moved the main body as an independent function with external linkage. Our kernel driver creates the data structures expected by the reconfiguration part and calls it directly. The reconfiguration part is still callable by the original system call implementation, therefore the `devcfg` is still reachable from userspace.

This approach has an advantage in performance. First, no file I/O is done at reconfiguration time; actually the whole file system and the Linux VFS layers are circumvented. Second, it was observed that when accessed from userspace, file reads were done in 4096-byte steps and each caused a PCAP DMA transaction. This means that for a half-megabyte partial bitstream, there need to be done 128 DMA transactions. As we were not restricted by such a limit, our driver performs a single big DMA transaction.

To demonstrate the reconfiguration performance, we measured the reconfiguration time of our driver. Interestingly, reconfiguration throughput was usually just marginally less than *half* of maximum theoretical, which is 400MB/s for non-encrypted bitstreams. Infrequently, the attained throughput was further halved. A summary is shown in table 5.9.

Throughput (MB/s)	Occurrence (%)
$\geq 200$	0.0
199 to 200	88.0
85 to 115	10.3
other	1.7

**Figure 5.9:** P.R. performance of our system.

The overall mean was 188.36MB/s. The measurement was taken from a sample size of 40K reconfigurations of almost evenly distributed accelerator slots, using the symmetrical 6-core design. To make a comparison, we also measured the reconfiguration time from userspace, copying the partial bitstreams from a RAM-based filesystem to `/dev/xdevcfg`. For a sample size of 10K iterations per core slot, for two module variants, we measured an average of 31.58 to 31.80 MB/s for the five 530KB sized bitstreams, and 33.88MB/s for the larger 682KB one. A speedup of 6x was actually more than what was expected.

# Chapter 6

## Application and Evaluation

In order to demonstrate and evaluate the proposed system, an example application was created. The chosen field was digital image processing, and a number of common filters were implemented in Vivado HLS to be used as accelerators.

Vivado HLS is an attempt from Xilinx to use a software language to describe hardware. It accepts a subset of C/C++ and produces HDL code, that can be synthesized with the existing synthesis tools.

The advantage of this approach is that it allows a very fast production of a working prototype, in order to assess the feasibility of an idea. Since the code describes an algorithm function and not its implementation, design exploration can be automated.

### 6.1 Accelerator Description

In image processing, from computational aspect, there are two main categories of filters. The first applies a transformation at pixel level. Using a mathematical formula or a look-up table, it maps one pixel value to another irrespectively of any other factor. The second category is spatial, in the sense that the new pixel value does not depend only on its old value, but also on its neighbors. To quantify this dependency, a “mask” or “kernel” is created, which is usually an odd-dimension square (or less often, rectangular) matrix whose each value represents a contribution weight. The kernel will scan the whole image pixel after pixel, and for each pixel it will calculate the sum of the products of the kernel’s weights and the corresponding pixel values. This mathematical

procedure is called “two-dimensional discrete convolution” and is symbolized with an asterisk. To summarize,

$$p'(x, y) = f [p(x, y)] \quad p'(x, y) = \sum_{m=-M/2}^{M/2} \sum_{n=-N/2}^{N/2} h_{m,n} \cdot p(x - m, y - n)$$

**(a)** Pixel transformation

**(b)** 2D Filtering

Convolution has a very interesting property: It corresponds to element-wise multiplication in the frequency domain. Therefore, we could calculate a discrete Fourier transform our image and our kernel, after padding the latter to the size of the former, multiply the transformed images, and apply the inverse discrete Fourier transform to return to spatial domain. In simpler words,

$$p' = IDFT \{DFT\{h\} \cdot DFT\{p\}\}$$

In respect to filter size, the latter method has a per pixel complexity of  $O(1)$ , whereas the former has  $O(n \cdot m)$ . As crossing the spatial and frequency domains has a computational cost, the latter method is cheaper for small  $m, n$ . For CPUs, the OpenCV library uses the kernel dimension of  $11 \times 11$  as the turning point.

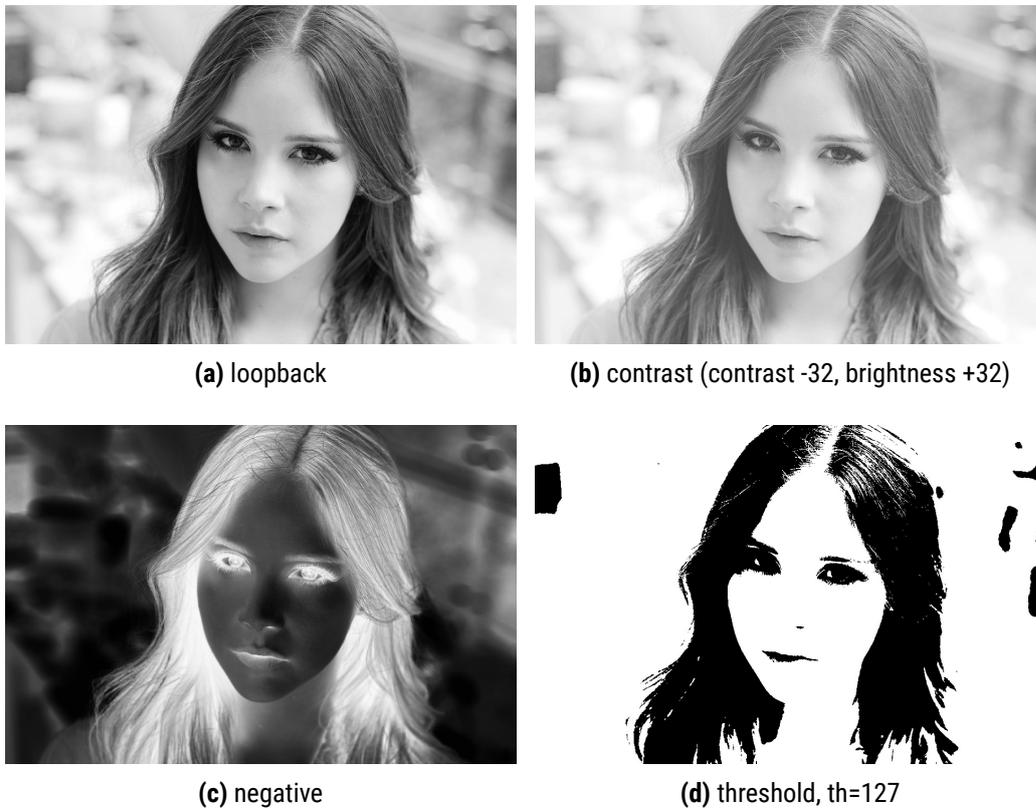
In our work, we use only  $3 \times 3$  and  $5 \times 5$  kernels. However, the principal deciding factor is that the former method has an obvious implementation in a streaming environment with no random memory access capability like ours. A line buffer that keeps the last  $n$  lines is sufficient\*. For a 1080p and a  $5 \times 5$  kernel, this translates to a BRAM usage of around 42kibits or 3 BRAM18 instances, which is perfectly affordable.

It should be noted that this method normally requires some special treatment of the border pixels, as the convolution kernel will attempt to access pixels outside the image. Several techniques exist, from padding with zeros, mirroring the edges, etc. For simplicity sake, we do not take care of this. As a consequence, a thin line of undefined contents will be visible at the upper and the right border.

For pixel transformations, an actual computation was used instead of look-up tables. This is because the transformations are either trivial (negative, threshold) or user

---

\*Theoretically, only a  $n - 1$  line buffer is necessary but the implementation would be more complex.



**Figure 6.2:** Pixel Transformations. Test image is in public domain.

configurable (contrast) and it is impractical to create table entries for every possible user choice.

### 6.1.1 Trivial Pixel Transformations

In total, there are three pixel transformation accelerators:

- **loopback:**  $p'(x, y) = p(x, y)$  – A dummy accelerator that just returns the input image. It is used for debugging and has no actual use.
- **negative:**  $p'(x, y) = 255 - p(x, y)$  – It returns the negative of the input image.
- **threshold:**  $p'(x, y) = \begin{cases} 255, & p(x, y) \geq th \\ 0, & p(x, y) < th \end{cases}$  – It replaces all pixel values above a certain threshold to maximum (255 for 8 bit depth) and all others to minimum (zero). Essentially, it transforms a greyscale image to a black and white.



**Figure 6.3:** Two-dimensional Filtering Kernels

None of these accelerators posed any difficulty to implementation.

### 6.1.2 Contrast and Brightness Transformations

The “contrast” accelerator adjusts the contrast and the brightness of the image. It uses the following equation:

$$\begin{aligned}\alpha &= 259 \cdot (c + 255) / [255 \cdot (259 - c)] \\ \beta &= b + 128 \\ p' &= \alpha \cdot (p - 128) + \beta\end{aligned}$$

As it can be seen, it uses both multiplication and division. An attempt to implement this transform with fixed point arithmetic was fruitless, as the accelerator was not placeable. The formula was simplified, with some precision loss, to ease the computation.

$$\begin{aligned}\alpha &= [256 \cdot (c + 256)] / (256 - c) \\ \beta &= b + 128 \\ p' &= [\alpha \cdot (p - 128) + \beta] / 256 + \beta\end{aligned}$$

This simplification allowed the use of integer arithmetic, while multiplication and division by 256 were done with shift operations. This version is placeable, however it is very sensitive in routing. Changing the width of  $\alpha$  and  $\beta$  by a couple of bits may render the accelerator from achieving timing closure to fail with unrouted nets. Reducing  $b$  and  $c$  to byte sized operands cause the simplification of the divider from 32b to 8b, which reduced core LUT6 utilization by more than 30%, ensuring successful routing and timing closure.

However, despite that a pblock LUT6 utilization of only 50% was achieved for the accelerator core count oriented design, this core remained very sensitive at routing with respect to the partition pins selection. For this reason, it was chosen to implement the initial configuration, despite the presence of much bigger cores, like the Gaussian Blur accelerator which reaches an 80% utilization.

### 6.1.3 The Sharpen, Emboss and Outline Filters

These three accelerators are grouped together as they all use a 3x3 kernel, the computation was quite straightforward and implementation was easy. The “sharpen” filter,

as the name suggests, is used for sharpening the image. It is quite often used at low intensity for improving the aesthetic result of photographs or for compensating excessive blurriness due to out-of-focus errors, albeit with moderate results. The “emboss” kernel is used to increase the sense of depth in an image. The “outline” kernel is a simple method of edge detection, used in computer vision applications.

$$\begin{array}{ccc} \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix} & \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix} & \begin{bmatrix} -2 & -1 & 0 \\ -1 & 1 & 1 \\ 0 & 1 & 2 \end{bmatrix} \\ \text{(a) Sharpen} & \text{(b) Outline} & \text{(c) Emboss} \end{array}$$

#### 6.1.4 The Sobel/Scharr Filter

The Sobel filter is a filter used to emphasize local pixel differences, for edge detection in computer vision. The filter uses two 3x3 kernels, one for horizontal difference and one for vertical. Essentially, there are two 2D convolutions taking place concurrently on the same dataset. The two convolution results produced for every pixel are combined to form the output pixel. After the original kernel by Sobel and Feldman, there were subsequent attempts to optimize its properties. A common variant, by Scharr, is also implemented within the same accelerator and is user selectable at run-time.

$$\begin{array}{cc} h_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} & h_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \\ \text{(a) Horizontal Difference Filter} & \text{(b) Vertical Difference Filter} \end{array}$$

Figure 6.5: The Sobel kernel

$$\begin{array}{cc} h_x = \begin{bmatrix} 3 & 0 & -3 \\ 10 & 0 & -10 \\ 3 & 0 & -3 \end{bmatrix} & h_y = \begin{bmatrix} 3 & 10 & 3 \\ 0 & 0 & 0 \\ -3 & -10 & -3 \end{bmatrix} \\ \text{(a) Horizontal Difference Filter} & \text{(b) Vertical Difference Filter} \end{array}$$

Figure 6.6: The Scharr kernel

The final pixel that represents the gradient at that spatial point is calculated by combining  $p'_x$  and  $p'_y$  as:

$$p' = \sqrt{p_x'^2 + p_y'^2} \approx |p'_x| + |p'_y|$$

The approximation above is quite frequently used even for software implementations. It is also used in this accelerator of course, as the space of 800 LUT6 would not permit the implementation of square root calculation. The visual difference is negligible.

The implementation was made possible only by forcing the HLS to use DSP48 tiles to perform the multiplication. For this reason, it is compatible only with the “big” reconfigurable partitions of the 16-accelerator design.

### 6.1.5 The Gaussian Blur Filter

The Gaussian Blur filter is an image smoothing filter. It is used for various purposes, mostly for noise suppression. The kernel derives from the two-dimensional Gaussian function:

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

The  $\sigma$  is the standard deviation of the Gaussian distribution and effectively controls the intensity of the blurring. For our accelerator, we attempted to implement the following kernel:

$$\begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$

Note that the sum of all weights are 256, and the final pixel will be divided by this value to avoid saturation and maintain image brightness.

As in Sobel, the use of DSP48 tiles were essential in order to fit the accelerator in 800 LUT6s. The initial attempts were unsuccessful, until it was discovered by pure chance

that the following kernel is implementable to our requirements:

$$\begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 25 & 16 & 4 \\ 6 & 25 & 32 & 25 & 6 \\ 4 & 16 & 25 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$

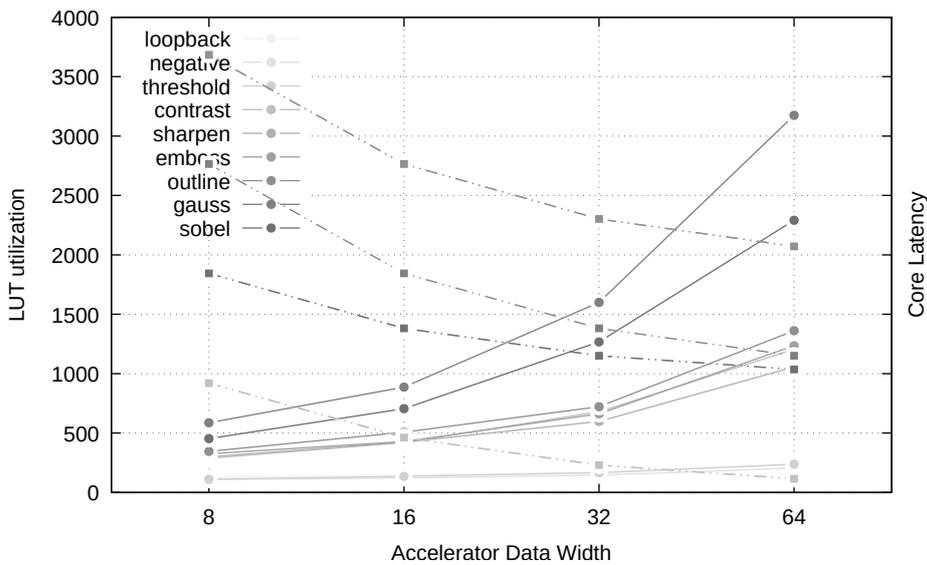
Note that the central term was reduced to 32, and the difference of four was spread equally to its four perpendicular neighbors. Without any other change in source code or synthesizer settings, the estimated utilization was reduced by 11% for LUTs and 9% for flip-flops, most probably due to increased symmetry.

### 6.1.6 Resource Utilization and Latency

An effort is made to quantify the resource utilization of the acceleration cores, as well as to obtain a primary performance indicator. In table 6.7 we can see the results coming from two sources: The latency is an estimation which is provided after compiling the HLS code to HDL. The resource utilization may be also provided post-compilation, however it appears the estimation is highly inaccurate and a post-synthesis result is presented, after exporting the core to a .dcp file. A comparison between resource utilization and expected latency is displayed in figure 6.8. As we see, accelerator size increases almost linearly with data width whereas latency gain is linear only at pixel transformations. On accelerators performing 2D convolution the gain decreases with the increase of width. This can be attributed to a sub-optimal BRAM access by the code HLS generates.

		<i>loopback</i>	<i>negative</i>	<i>threshold</i>	<i>contrast</i>	<i>sharpen</i>	<i>emboss</i>	<i>outline</i>	<i>gauss</i>	<i>gauss/dsp</i>	<i>sobel</i>	<i>sobel/dsp</i>
8 bit	LUT	125	110	111	288	300	326	346	587	466	454	635
	FF	142	114	111	303	290	357	394	553	502	492	628
	DSP	0	0	0	0	0	0	0	4	8	0	2
	BRAM	0	0	0	0	3	3	3	5	5	3	3
	Latency	2.07	2.07	2.07	2.07	4.15	8.29	8.29	6.22	6.22	4.15	4.15
16 bit	LUT	147	122	136	420	423	429	507	887	770	706	1048
	FF	182	154	137	398	386	443	578	942	741	765	970
	DSP	0	0	0	0	0	0	0	8	16	0	4
	BRAM	0	0	0	0	3	3	3	5	5	3	3
	Latency	1.04	1.04	1.04	1.04	3.11	6.22	6.22	4.15	4.15	3.11	3.11
32 bit	LUT	171	146	166	680	598	662	722	1600	1339	1268	1968
	FF	262	234	189	561	620	702	873	1658	1267	1294	1950
	DSP	0	0	0	0	0	0	0	12	28	0	8
	BRAM	0	0	0	0	3	3	3	5	5	3	3
	Latency	0.52	0.52	0.52	0.52	2.59	5.18	5.18	3.11	3.11	2.59	2.59
64 bit	LUT	231	206	238	1206	1057	1235	1362	3175	2541	2291	3784
	FF	422	394	293	907	1099	1342	1608	3168	2585	2413	3559
	DSP	0	0	0	0	0	0	0	20	52	0	16
	BRAM	0	0	0	0	3	3	3	5	5	3	3
	Latency	0.26	0.26	0.26	0.26	2.33	4.66	4.66	2.59	2.59	2.33	2.33

**Figure 6.7:** Accelerator resource utilization (post-synthesis) and latency (HLS estimation). Target FPGA was Z-7020 at 7.5ns. BRAM is in 18k units and latency is measured in millions of cycles. The “/dsp” suffix denotes that the integer multiplication was assigned manually to a DSP48 core.



**Figure 6.8:** Resource Utilization vs. Core Latency

## 6.2 Accelerator Interface

Each accelerator connects to the system via two pathways – one AXI-Stream that exchanges data with the AXI DMA using two channels, a read for incoming data and a write for outgoing, and one AXI-Lite for status and control. As already discussed in section 3.6.1, the reconfigurable module must remain consistent throughout all implementations. In Vivado HLS this translates to:

- The width of the AXI-Stream interface must remain consistent throughout reconfigurable modules.
- All reconfigurable modules will have exactly one read and one write channel.
- The address width of AXI-Lite interface must remain consistent for all reconfigurable modules.
- The implementation tools expect the module name to be consistent throughout all reconfigurable module implementations.

The width of the data stream is the most important design choice for the accelerator performance. Our system does not care for the value and the kernel driver does not see it; still, it must remain consistent throughout all reconfigurable module variants of each reconfigurable partition. The reconfigurable partitions themselves however, do not need to match each other.

In our application, we made the accelerator width a user configurable parameter. The script that builds the HLS modules (see appendix B.1) will generate a C header file that is included by all HLS C++ source files and contains the appropriate definitions to allow a 8b, 16, 32b or 64b accelerator width. Higher widths are possible but will require source modifications.

The accelerator will take advantage of wider stream by loading two, four or eight pixel at once, and process them in parallel. Whether the HLS synthesizer does this efficiently and if performance scales linearly, it will have to be determined by benchmarking.

Regarding the number of channels, it was decided to be two. This was an obvious choice, given the exact mapping with AXI DMA channels – one receive, one transmit.

```

int CORE_NAME(axi_stream_t& src, axi_stream_t& dst, int brightness, int contrast)
{
#pragma HLS INTERFACE axis port=src bundle=INPUT_STREAM
#pragma HLS INTERFACE axis port=dst bundle=OUTPUT_STREAM
#ifdef CLK_AXILITE
# pragma HLS INTERFACE s_axilite clock=axi_lite_clk port=brightness bundle=
    control offset=0x10
# pragma HLS INTERFACE s_axilite clock=axi_lite_clk port=contrast bundle=control
    offset=0x18
# pragma HLS INTERFACE s_axilite clock=axi_lite_clk port=return bundle=control
    offset=0x38
#else
# pragma HLS INTERFACE s_axilite port=brightness bundle=control offset=0x10
# pragma HLS INTERFACE s_axilite port=contrast bundle=control offset=0x18
# pragma HLS INTERFACE s_axilite port=return bundle=control offset=0x38
#endif
#pragma HLS INTERFACE ap_stable port=brightness
#pragma HLS INTERFACE ap_stable port=contrast
/* module implementation */
}

```

**Figure 6.9:** An HLS reconfigurable module function declaration.

However, it is still the simplest choice and a lot more sophisticated architectures can be built. For a discussion on this, see section 7.2.3.

As for the AXI-Lite channel, its width is strictly 32 bits, as defined by the protocol. However, the address width is variable and depends on the addressable space of the accelerator. This is decided indirectly by the mapping of the accelerator input and output variables.

In order to demonstrate how all this is done, in listing 6.9, a reconfigurable module function declaration is displayed, among with its HLS directives.

It is an example from the “contrast” accelerator. The first thing to notice is that the function name does not mention the reconfigurable module’s variant and it is a generated constant. Its contents are actually related to the “axi\_stream\_t” type which is a user defined type. A few lines of code are worth a thousand words of comments:

```

/* generated.h */
#define CORE_NAME zcore16
typedef uint16_t axi_data_t;
/* ... */

```

```

/* common.h */
typedef ap_axiu<sizeof(axi_data_t)*8, 1, 1, 1> axi_elem_t;
typedef hls::stream<axi_elem_t> axi_stream_t;
/* ... */

```

The file “generated.h” is generated by the script shown in appendix B.1. The “axi\_data\_t” is essentially our accelerator word size, and we see it is used to define “axi\_elem\_t” which describes the full AXI-Stream signal set. From left to right, it defines the effective data (TDATA) width, the user-defined (TUSER) width, the master/-source identity (TID) and the slave/sink identity (TDEST). All quantities are in bits and names in parenthesis express the standard’s nomenclature. The “axi\_stream\_t” represents the AXI stream itself and it is a standard C++ IO stream. The implemented accelerator functionality, i.e. the contrast and brightness adjustment, is not represented in the function name as it will be the name of the reconfigurable module in design hierarchy, not solely of this variant. All variants have to share the same name. However, the word width is represented – this is to allow the system to implement reconfigurable partitions of varying sizes. However, no such an implementation was done in this work.

Back to our listing, we can now discuss the AXI-Lite status/control port. Every such an input must be mentioned in three places:

1. At function declaration. If the variable is an output, it must be declared as a pointer.
2. At HLS INTERFACE s\_axi\_lite pragma, where each variable is defined as a member of a specific AXI-Lite interface (“port” and “bundle” properties) and is assigned an address (“offset” property). It is called an offset as its value represents the address of this variable in respect to the start of this AXI-Lite interface’s address space.
3. At HLS INTERFACE ap\_stable. This forms an explicit promise on our part that the bus master will not modify the contents of these variables during module execution.

It is the value of the “offset” property that actually defines the address width of the AXI-Lite interface, i.e. the size of AWADDR for the write channel and ARADDR for the read channel. Their size will be the minimum sufficient to accommodate the maximum

offset among all memory mapped registers that represent the variables. The compiler will not differentiate between read-only or write-only variables, so a safe way to impose the width of both channels is to place the `return` variable at the end of the addressable space. Here, by placing `return` at `0x38` (variables are aligned at 8 bytes boundaries) we are defining a space of `0x00..0x3F`, which corresponds to 6 bits of address width. Within this address space, the first 16 bytes are reserved by HLS. The first accelerator argument is placed at `0x10`, and subsequent arguments at 8 bytes distance between them. This convention may be altered but a header file (`param.h`) modification of the kernel driver is necessary.

The property “`clock`” was made optional and it allows the AXI-Lite to be clocked by a different source from the input and output streams. This is because generally AXI-Lite slaves achieve lower frequency than AXI-Stream ones and it might be worthy to explore if higher clock can be achieved by using separate clock domain.

Apart from the explicit memory mapped registers, some implicit are instantiated that form the status and control protocol of the HLS core, the default signal set being the `ap_ctrl_hs`. The programming sequence consists of the steps described in the following list, and an example memory map is given in figure 6.10.

1. The signal `ap_idle` should be asserted, indicating the core is idling.
2. All memory mapped registers corresponding to core parameters must be written now.
3. The user asserts `ap_start`, commencing the execution. In response, the core will de-assert `ap_idle`.
4. When all input is read, the `ap_ready` is asserted, and when all output is written, the `ap_done` is also asserted. The return value may now be read.
5. The `ap_idle` is asserted, indicating that the core has finished execution and is now idle.



times bigger yielded a gain of around 2%.

size→ ↓QD	Throughput (MB/s)			
	600kiB		2025kiB	
	loopback	sharpen	loopback	sharpen
1	1437.9	158.4	1547.3	159.5
2	1511.2	316.4	1571.9	319.0
3	1595.8	474.2	1598.6	478.4
4	2090.8	632.7	2103.0	637.8
5	2564.5	791.1	2606.5	797.2
6	3090.5	948.8	3134.8	956.7
7	3092.5	945.2	3127.6	958.4
8	3027.3	947.4	3131.3	958.7
9	3019.2	938.3	3124.6	957.7
10	3029.7	951.5	3130.0	958.8

**Figure 6.11:** The effect of DMA transaction size on performance. Test: 6-core design, 10K iterations of each queue. QD: Queue depth, i.e. the number of concurrently executing tasks.

Next, we assess the effectiveness of our scheduler algorithms. As a load we make equal use of all accelerator variants. We compare all replacement algorithms but the “Priority” as it has only specific uses that do not apply here, and the test is repeated for “First Fit” and “Best Fit” as the first-stage empty slot selection algorithm.

The results are shown in table 6.12. As for the 16-core design, the biggest surprise is that the “First Fit” slot selection algorithm performs better than Best Fit. Among the replacement algorithms, the “Least Popular” performed significantly better than any other.

As a reminder, the “Least Popular” chooses its victim slot by finding which slot has least probability to be selected by another currently registered task and therefore minimizing the probability to cause a reconfiguration at a later time. It is identical to “Best Fit”, which uses this probability as the fitness criterion. Conversely, “First Available” and “First Fit” simply return the first suitable free slot. Interestingly, the algorithm that is best for the first stage is the worst for the second stage, and vice versa.

All other algorithms are implementations of the generic replacement algorithms mostly used in caching applications. They fail to perform as good, most probably because they lack the hardware awareness the “Least Popular” possesses – they are based only on the so far collected statistics.

The overall picture in 6-core design is different. Given that the design homogeneous and we did not alter the core or task affinity mask, the “popularity” criterion degenerates to something similar to first-fit, losing its advantage to other algorithms. Generally, differences among all algorithms were not characteristic and they could be reversed at another load.

Algorithm	Throughput (MB/s)			
	Homogeneous 6-core		Heterogeneous 16-core	
↓2nd stage 1st stage→	First Fit	Best Fit	First Fit	Best Fit
Least Popular	318.9	319.6	926.0	832.6
First Available	317.8	324.7	743.7	721.5
First In First Out	318.8	320.4	732.7	780.8
Least Recently Used	320.0	324.9	778.8	735.6
Least Frequently Used	311.3	313.6	787.6	749.5

**Figure 6.12:** The effect of scheduler algorithms on performance (QD=45).

Last but certainly not least, we will test how the application scales with the queue depth, or in effect, with the number of slots. We use a single accelerator variant for each test in different queue depths, and we repeat the procedure for both 6-core and the 16-core design. To remind, out of the 16 cores, only 10 are capable of holding a linebuffer for 2D filters.

The results are displayed in figure 6.13. Looking at the 6-core design, the first to observe is the significantly sub-linear increase of throughput with the number of active slots for the pixel transformation accelerators. The culprit here is the single flat memory zone we chose to define. Despite that the design utilizes all four HP ports which see two separate ports of the memory controller, this is not visible to the driver’s memory allocator (see section 5.8). Therefore it assigns the next few tasks at address regions that correspond to the same HP port, which is saturated at 2272MB/s (142MHz by 64b per direction). The effect is not observed in 2D filter accelerators, as the cumulative throughput of all slots running concurrently is not sufficient to saturate the port.

On the contrary, in 16-core design we segmented the memory address space to four zones, one per HP port. As we saw in design architectural diagram in figure 4.2, the memory controller ports are shared by the HP ports in pairs. For this reason, we purposefully designed the interconnect so the reader of HP0 cannot be writer to HP1,

which both share the same memory controller port. Thus, as it is seen in the results, throughput scales perfectly linearly.

On 16-core design, the pixel transformation accelerators saturate quite early due to the interconnect, as the Zynq ports are clocked lower and are configured to 32b. Still, given the count of ports we used (all four HPs and both S\_GPs), we expected the saturation to happen later. Moreover, the ripple at higher queue depths cannot be explained.

However, the 2D filter accelerators scale perfectly up to 10 slots, which is the number of slots that are capable of supporting them. Therefore, for these accelerators the saturation comes from the computational resources and not the I/O.

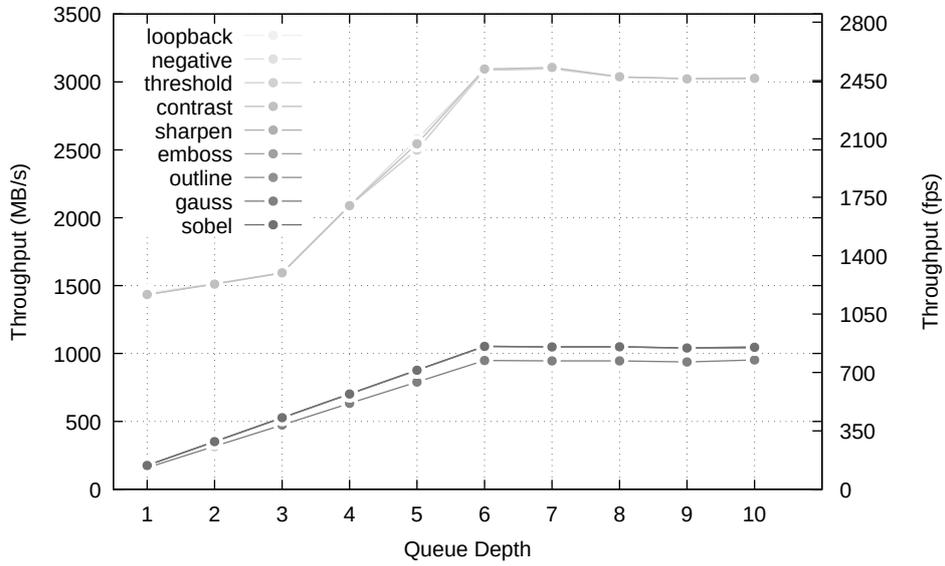
Studying the saturation points of these two groups, we can conclude that the design of the 16-core system was not an optimal fit for our application. The group of pixel transformation cores is eight times faster than of 2D filters, so unless the task count of the former is overwhelmingly higher than of the latter, the 6 cores that cannot execute 2D filtering are severely underutilized. But even in the scenario that pixel transformation tasks are so many that these 6 cores are fully occupied, their performance would be capped by our I/O capability before we even factor in the traffic from the other 10 cores.

Within the accelerator, we see that the throughput ratio of pixel transformations to 2D filtering is set to 8-to-1 on 16-core design. However, this advantage fades away to a mere 1.33-to-1 in the 6-core design. Apparently the BRAM is saturated by the 8-pixel pipeline of the 6-core design, giving an advantage to the 2-pixel pipelined 16-core design which has more 2D filtering capable slots by a 1.66-to-1 ratio.

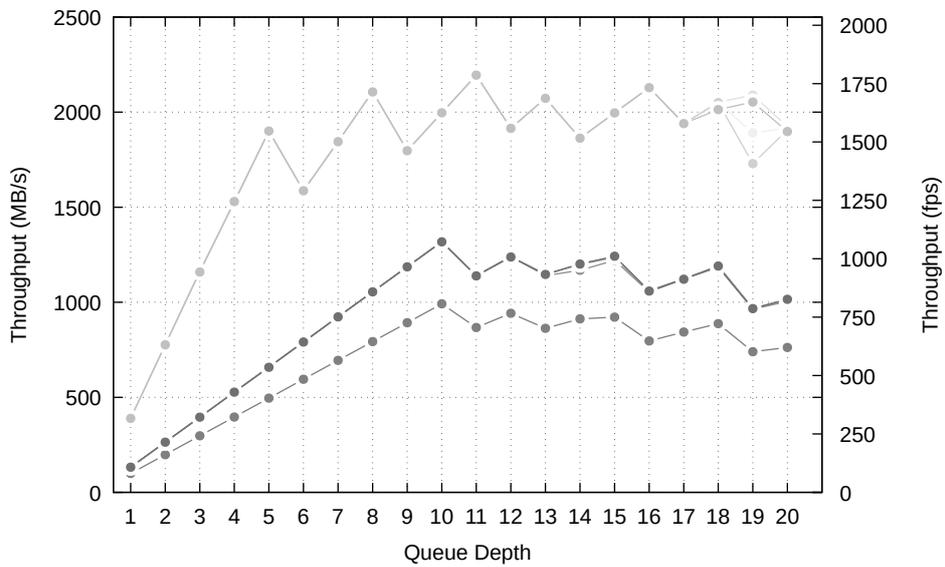
Thus, in pixel transformation workloads, the 16-core design saturates at around 2.1 GB/s or 1700 fps whereas the 6-core saturates at 3.1GB/s or 2500 fps. Conversely, in 2D filtering the 16-core can reach 1250MB/s while the 6-core is limited at 1050MB/s. As we observe from table 6.12, in a balanced scenario the performance of the 16-core design is overall higher than of the 6-core design.

Concluding, if we were to chose a better architecture for our benchmark application, it would be beneficial to sacrifice the 6 small core slots of the 16-core design in favor of allowing larger slots for the remaining 10. On the other hand, we saw that the 8-pixel pipeline of the 6-core design saturates the BRAM. The middle ground of a 32-bit (4-pixel pipeline) accelerator appears attractive. We can assume that the BRAM

will still saturate the 2D filter performance, but we should factor in our 16-core design all interconnects are already 32-bit wide, as well as the memory-mapped section of the AXI DMA controllers. If we convert our 16-bit accelerators to 32-bit, we would double the resource consumption of the reconfigurable partitions themselves but not of the static part. This is a rather modest price to pay, compared to the transition from 32b to 64b (or to 128b in ZynqMP) where all DMA controllers and interconnect need to be upgraded to the new data width. Finally, the fewer accelerator slots might ease routing and make possible to attain a slightly increased PL clock.



(a) 6-core design



(b) 16-core design.

**Figure 6.13:** Accelerator performance scaling with queue depth increase. Upper line group consists of pixel transformation accelerators whereas lower group is 2D filters. The line at the very bottom is the gaussian blur filter (5x5 kernel). QD=45, iterations: 5k+, image frame size 600kiB (960x640x8b).



# Chapter 7

## Conclusion and Future Work

We developed a complete reconfigurable system, capable of providing hardware accelerators on-demand to the end user. The system was designed with a server environment in mind – that is, a capable GNU/Linux system performing random user tasks in an unpredictable timing but of somewhat predictable nature, whose computational kernel can be off-loaded to the FPGA. As such, it is capable of handling any dynamic load within the system’s memory limits, with no pre-defined static schedule or any other prior knowledge.

Usage scenarios may include a web server, which can use the FPGA to perform the common encryption/decryption and compression/decompression tasks that are typical to such a workload. Our specific application could be made useful in a server that performs image or other media processing over a vast number of user content. Finally, it could also find place in an interactive multimedia application, allowing high-performance video editing or other media transformation.

The design of the system tried to cover many possible scenarios, including the need of accelerators with a dedicated path to the memory, accelerators of different degree of criticality, heterogeneous memory resources of different speed, etc. It is designed right from the start with concurrent memory access in mind and it takes advantage of the platform hardware capabilities for parallelization. Acknowledging the variety of usage scenarios, several scheduling and accelerator replacement algorithms were implemented. Additionally, the administrator may define an affinity mask of the accelerator variants to the available slots, whereas the user may define a similar mask for their tasks.

This way it is possible to enforce isolation, either for quality of service and predictability or for security.

A system shared library is provided that abstracts all system details, making it easy of the application developer to use the system without any need of hardware knowledge.

The framework was successfully implemented in the Zynq-7000 based Zedboard from AVNET. Two different architectures were realized in order to assess design parameters such as the interconnect architecture, the memory layout, the accelerator data width and pipeline depth, etc.

A digital image processing application was implemented and used as a benchmark. Certain interesting observations were made from which we can deduct the validity of our design choices.

An attempt to port the system to the UltraScale+ based zcu102 was thwarted at the software porting phase, as Xilinx had yet to provide required software support for Partial Reconfiguration on this device class. Nonetheless, the hardware design was complete, and this allowed us to assess the new architecture's capabilities, including the complete overhaul of partial reconfiguration support.

## 7.1 Challenges and Lessons Learned

The development of this system was not without unpredicted or underestimated problems. In this section we will try to discuss some of the noteworthy issues that arose during development.

### 7.1.1 The Implementation Workflow

The first and foremost is our underestimation of the difficulty that arises with the hardware implementation of such a system and of partial reconfiguration in general. A basic working system to be used as a proof-of-concept is rather easy to implement. However, as the clock frequency and the number of reconfigurable partitions rise, things change a lot. A major issue is that there is no partition placer tool despite that proper size, shape and placement is critical for attaining the highest performance. Thus, the work of the placer had to be done manually. The effort of this endeavor increases exponentially with the target clock. For example, our 16-core design could possibly be successfully routed

with the first attempt at 100MHz. Increasing the clock to 125MHz requires many hours or tweaking. In order to reach the final goal of 133MHz it required several days of experimentation with partition moving or swapping, changing tool settings, choosing the correct initial configuration, finding per-variant optimal implementation settings, etc. As performance difference proved to be minimal, it is a question if it was worth the effort.

The partition sizing was a decision that if it was to be taken again, it would be taken differently. In section 4.2.3 we discussed the sizing trade-offs. A factor not properly weighed was the availability and capability of routing resources. Each partition comes with a fixed cost which does not constitute only the LUTs of AXI DMA, but also the wires that connect it, the interconnect crossbar, the wires that implement the control interfaces of both the accelerator and the AXI DMA, as well as their own, separate interconnects. Routing can become difficult and at some point, it will necessitate a clock frequency reduction.

### 7.1.2 The Tool Quality

An important issue proved to be the implementation tool maturity. The partial reconfiguration TCL scripts that Xilinx offers are not optimized at all. They are not multi-threaded even in cases it would be easy to do so, e.g. a parallel OOC synthesis of module variants or parallel implementation of module variants. Some work could be reused, e.g. the implementation of each variant needs carving out the modules of initial configuration, and this is repeated for each variant when the carved configuration could be saved and reused.

As of version 2017.2, the Vivado implementation tool support for UltraScale devices appear to be immature. Partial reconfiguration frequently causes locking of special resources (BRAM and DSP tiles) during initial configuration, causing the placement failure of the next. This is irrespectively of the variant used for each configuration. The issue was worked around by reshaping the affected partitions, but no golden rule could be figured out for what consists a favorable shape, and no help could be gained from Xilinx forums.

A final and very important issue is the tool speed. At the GUI front, designing larger systems visually is virtually impossible. The slightest action causes a sequence

of object property propagation which can be an excruciating for a large design. This can manifest at some unexplainable ways, like refreshing the custom IP directory data when the user changes the floorplanner's color palette. The designer must not hesitate to learn operating Vivado from the TCL console – the effort invested will pay out fast.

At the backend front, the workflow on ZynqMP target is sluggish in every possible way. Implementation is much more time consuming than the four-fold increase of LUT count between Z-7020 of Zedboard and XCZU9EG of zcu102 could justify. Even marking a module as a black-box is considerably slower; a process that takes less than a minute on Z-7020 would take more than an hour on XCZU9EG. A full partial re-configuration synthesis, implementation, verification and bitstream generation would need several hours at most for the former but around a week for the latter in a dual Xeon X5660.

Clearly, in any new project an UltraScale-class device should not be used for initial development. It would be preferable that first stages to be done in a 7-class device if possible, and then switch to an UltraScale-class device when the design requires device-specific development.

### 7.1.3 The Efficiency of HLS

Finally, the usefulness of the HLS tool should be reconsidered. The HLS was chosen for accelerator implementation as it allows quick prototyping. The time from the initial algorithm conception to a validated working bitstream can be more than an order of magnitude less than of a traditional HDL workflow.

However, from that stage to the final optimized version is a completely different story. The resource consumption and attained latency of HLS is difficult to control and even more difficult to predict. As we saw at section 6.1.5, a minor change of filter values reduced overall LUT utilization by 11%.

Extreme caution should be paid to variable sizes as they dictate the inferred execution unit size – e.g. in contrast accelerator (see 6.1.2) we used `int` types to express the brightness and contrast parameters, as from the processor we only do 32-bit load/stores. This inadvertently later caused a 32-bit divider to be inferred when an 8-bit would suffice. In many cases, when the bounds of the values a variable will receive in its lifetime can be known at compile-time, HLS will trim the variable accordingly – a typical ex-

ample is the index of a constant bounds loop. Integer promotion oftentimes may be inconsistent. For example, if a product of two shorts (16 bits) is stored to a char (8 bits), an 8-bit multiplier will be inferred. However, if operands were ints (32 bits), a 32-bit multiplier would be inferred. Usually these discrepancies will be sorted out by the synthesizer. Finally, the effect of setting the clock target can be odd. If HLS fails to achieve the desired period, one could try an even more difficult target and the HLS may restructure the code to achieve the original, previously failed, target.

To make things worse, the HLS resource utilization and latency estimates should be treated as randomly generated numbers. In order to confirm resource utilization, the designer should export the compiled output to a synthesized device checkpoint. The post-synthesis resource utilization report is far more precise. Regarding latency, numbers may be deceptive. For example, in figure 6.13 we see that the measured throughput does not match the HLS latency estimation shown in table 6.7.

At the end of the day, the use of HLS was a regretful decision. The effort spent to control resource utilization and latency was unexpectedly high and the result was mediocre. It was viewed as a quick and easy solution to implement a simple test application to evaluate our system but it proved to be neither easy, and certainly, nor quick.

## 7.2 Future Work

There are several opportunities to optimize or extend this work. They range from simple additions and improvements to revising significant architectural decisions. In this section, we will discuss some of these ideas.

### 7.2.1 Integration with FPGA Manager

We discussed FPGA Manager in section 5.10. It is imperative to replace the partial re-configuration driver from `devcfg` for two main reasons:

- **Portability:** FPGA Manager is a Linux kernel API and that offers hardware abstraction to FPGA manipulation. Using it, it will not only provide future support for ZynqMP, but it will allow porting the system to other architectures like Microblaze or even to Altera/Intel SoCs.

- **Integration:** Making Linux to be aware of the FPGA is a big step forward. We already discussed the advantages of this framework but we should stress again that this is not just an FPGA programming interface. As the framework matures, more and more FPGA related functionality will be added, obsoleting custom and out-of-tree solutions.

If that reasoning does not sound compelling enough, we should add that Xilinx has stopped development of `devc fg` and has completely removed it in its latest kernel (v4.14).

### 7.2.2 Use of Advanced DMA Modes

The AXI DMA IP core was used in Direct Register mode because it is the simplest solution, not the most efficient. The drawback of this method is that there is a significant time interval between one transaction completion and next transaction start, during which the controller is idling.

The interrupt service latency should not be blamed as it is usually at the 100 $\mu$ s range, a value quite small compared to the transaction processing latency or the reconfiguration time, both at the order of a milliseconds. The problem arises from the fact that we do not immediately know what transaction is going to be processed afterwards. The corresponding slot will be marked as free, and it will wait to be chosen at a later scheduler invocation. How long this will be, depends on the system load and how frequently the Linux process scheduler is called, which is configurable.

Even if we manage to quantify and/or reduce it, we still need to analyze whether it is avoidable. If the system includes many accelerators and few slots, it is highly probable that the slot that has just finished will be reconfigured, which closes the door for any optimization. On the contrary, if there is a good probability this slot will re-use the same accelerator, it would be very beneficial if we could immediately schedule next transaction.

The Linux DMA Engine API supports scatterlists, an abstraction of DMA Scatter-Gather. It is a software construct; an array of transaction descriptors that the API uses the program the DMA controller, to avoid the unnecessary controller pauses. It does not require hardware support, but if present, it will be taken advantage of. The AXI

DMA ip core does support scatter-gather, but at a cost of an one-third increase of resource utilization (see table 3.4).

Regardless from whether it is beneficial in most situations, the driver should provide this possibility.

### 7.2.3 Rethinking the Accelerator - DMAC Relationship

Initially, the system was envisioned to be capable of supporting a handful of accelerator slots, with four being the initial target. With that in mind, using one AXI DMA for each accelerator slot was not an issue. An assumption that may still hold true for the 6-core design, but was first challenged at the 16-core design. When the capacity of the XCZU9EG permitted the implementation of 63 accelerator slots, it cast a doubt on whether 63 AXI DMA controllers were really necessary.

There are two reasons that drive us to object the current design:

1. An accelerator is not always capable of pulling an AXI beat per cycle. Our convolutional filters pull data once per 8 cycles, so the AXI DMA IP is forced to wait for 7 out of every 8 cycles. If we could time-share the DMA controller, we could theoretically support 8 accelerators from a single DMA controller.
2. It is common that a single computational task is consisted of multiple stages of processing, each being represented as an accelerator core. Instead of pushing in and out data from and to every accelerator, each accelerator could drive directly the next stage. If we could do this, we would have another major gain: the data stream would flow directly within the PL realm, without reaching main memory until the last link of the processing chain has finished. This would greatly reduce latency and relieve the pressure on the memory controller.

For the first case, we can propose exploring Multi-Channel DMA operation. In this configuration, a DMA controller can control multiple AXI streams from a single interface, by time-sharing its resources. The streams can then be routed to their correct destination after setting the user AXI signal TDEST. The AXI DMA IP has this capability and recently Xilinx released the AXI MCDMA IP core. The routing can be performed using an AXI Stream Switch. Of course, both the mutli-channel support and the routing take up FPGA resources. For the former, the table 3.4 can be advised,

and as for the latter one may refer to [32]. Despite the idea is to share a DMA controller among multiple accelerators, the same principle can also be applied in case we would need multiple input or output streams for a single accelerator.

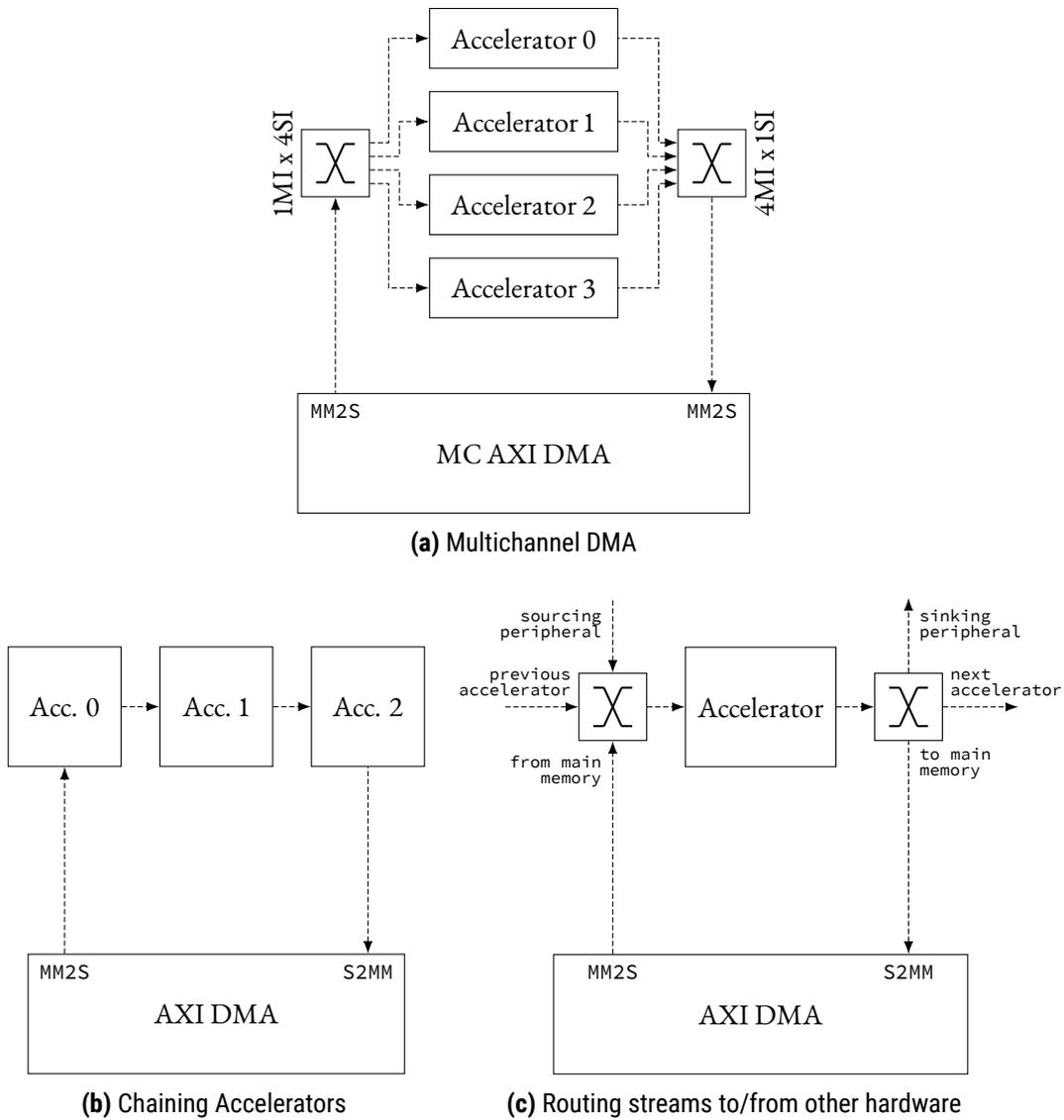
The second case is more complex and the ideas behind it may be materialized in several ways, depending the target application. The possible approach would be to chain accelerator slots. It could be implemented either as a set of peers or as a big main core, accompanied by smaller pre and post processing cores. In any case, the output stream of the one core is driven to the input stream of the other. It would be a direct connection, no interconnect is necessary. If a core within a chain is not needed, it will be loaded with a loopback module.

One could object that in a scenario that we have few and large accelerator slots serving tasks that cannot be easily broken down to elementary computations, such an arrangement would lead to far too many wasted FPGA resources by cores loaded with a loopback module. We could withdraw our idea of having only a single DMA controller but keep our idea to directly connect cores, just to avoid unnecessary main memory access. We could even extend it, to include the possibility of sourcing or sinking data from/to a hardware peripheral, to support a real-time embedded setting.

#### 7.2.4 Task Buffer Migration

The system allocates a buffer pair each time the user application configures a task. If a configured task declares a new configuration, the old buffers are released and new buffers, of the new requested size will be allocated. The new buffers will be placed wherever the memory allocator deems best – there is no connection to the old buffers, and data will not be preserved.

Since this action is initiated by the user, the data loss is acceptable since we assume the user will only issue such a request when they have finished processing and/or saved their data. However, there might be cases it would be beneficial to perform an involuntary buffer move. For example, the external fragmentation of a memory zone might not permit a creation of a new task despite that there is enough free space. In another case, there might be free and contiguous space in a zone where a task is not allowed to be placed due to hardware or affinity restrictions. Finally, a re-arrangement of user buffers among memory zones may help re-balance data traffic to maximize parallelization.



**Figure 7.1:** Alternative accelerator and DMA controller arrangements.

Since these scenarios are determined by the kernel without the consent of the user application, they have to be made transparently and without data loss. Indeed, a key reason we implemented a custom memory allocator was to make this feature easier and simpler to implement. Implementation should be straightforward:

1. Ensure that no DMA transaction is currently taking place on behalf of this task and prevent this from happening while the migration takes place.

2. Find the new optimal buffer location.
3. Update allocator bitmap to reserve the new space.
4. Migrate the page frames to the new location, updating the user process page table.
5. Update allocator bitmap to release the old space.

### 7.2.5 Accelerator Control

The AXI-Lite protocol rightfully gained its name as it requires only a few dozen of LUT instances to implement a slave interface. Nonetheless, this slave interface consists of 94 wires. Despite that we have so modest demands for throughput from a configuration port, the interface still implements separate address, read and write channels, as all members of the AXI family do. On top of that, we should add the implementation of the required interconnect.

It is worth exploring the use of a simpler protocol, even a serial one. The I<sup>2</sup>C is rather slow but it uses only two wires (clock and data) shared between all slaves. The SPI needs a clock, a data-in and a data-out plus one slave select per slave, but it is significantly faster. A more efficient solution would be to embed configuration data in the data stream. It adds some implementation complexity and might be a bit restrictive in some cases, but it comes at no cost in routing resources.

Moreover, currently, the accelerator control is modeled after the Vivado HLS control signals. While it would not be very difficult to generate these signals from an HDL or another HLS tool, a cleaner solution would be to define a custom interface and offer wrappers for any HLS tool.

### 7.2.6 Accelerator Interrupts

We assumed an accelerator would have a input and an output stream, with no other side-effects. However, an accelerator may need to generate interrupts. Support of this feature may take the form of offering a system call that blocks until the interrupt is acknowledged, as it is done in EPEE[6]. Another approach would be by delivering

POSIX Signals, in order to allow the user application to perform useful work while waiting for the notification.

### 7.2.7 Portability

The system was designed using the Xilinx tools on a Zynq platform. As it is already discussed, FPGA Manager will solve the unportable nature of partial reconfiguration.

Two additional points that tie us to a specific architecture are the AXI interconnect and the ARM processor. The former is not expected to be an issue, at least if we fix the accelerator control protocol (see above). The latter is a bit more a sensitive issue. While we do not bind to the processor architecture, we extensively use the FDT. GNU/Linux has decoupled it from Open Firmware and the current trend is to expand support to more platforms. Despite that support for x86 was introduced back in 2011, it is uncertain how easy would be the transition to the future x86-connected FPGAs.

However, the biggest -by far- burden is the heavy dependence on Xilinx IP cores. It is certain that Intel/Altera would have an equivalent core for every basic core of Xilinx, however they would differ on configuration and interface.

While it is unavoidable to get rid of all of them, as some provide support for vendor specific hardware, they could be reduced to a minimum by replacing generic functionality (like a DMA controller) with other academic and/or open-source alternatives.

### 7.2.8 Scheduler Improvements

So far, the scheduler may decide only by taking into account the past (frequency of use, time elapsed from an event) or a user wish (e.g. priority, affinity). The scheduler has no knowledge of the future except from the tasks that are in-flight trying to find a free core.

A useful feature would be to help the scheduler by stating an intent to execute a certain list of tasks. This way it can collect information that will more accurately predict which accelerator core would less likely be needed or it could postpone or re-arrange scheduling of tasks so to minimize the number of slot reconfigurations.

### 7.2.9 Bitstream Size

In our work, for each module variant we produced a partial bitstream for every reconfigurable partition. One might wonder, why not just use one for every variant?

In modern FPGAs, the routing of the static design is permitted to cross a reconfigurable partition in order to reduce pressure on router. This had as a consequence that every partial bitstream is totally different for each reconfigurable partition it is implemented. Xilinx does not provide a way to restrict this routing apart from constraining all static routing to a static pblock, with whatever impact it would have on the quality of the result. The recent Altera Stratix 10 has a different architecture what permits bitstream relocation. There are several academic works that try to approach the problem from different angles, either at run-time or with bitstream manipulation prior to storage. For a detailed discussion on bitstream relocation and other current PR-related topics, one can address to [33].

In our work, bitstream storage was a significant burden for the memory constrained Zedboard, especially since we use a ramdisk for all storage. For 16-core design, to sum of all bitstreams were 30MB. Using XZ compression we reduced it to 2.2MB, but we automatically decompress a bitstream when it is first registered to the kernel driver. If we chose to decompress the bitstream just before programming, it would incur a latency penalty at the order of 25ms, which is an order of magnitude of the partial reconfiguration itself, usually a couple of milliseconds.

A future improvement should explore possibilities for relieving this issue. Xilinx offers bitstream compression, where unused reconfiguration frames are not stored. This method is not very efficient in compression ratio, but a compressed bitstream can be programmed directly, therefore a minimal, if at all, latency cost is paid.

The XZ algorithm has very high compression ratio but it is slow. There are several high speed algorithms, for example lz4, zstd or snappy, that can be tested and if the additional latency is tolerable we could decompress the bitstream just before programming.

The on-board SD card is usable for permanent bitstream storage but it is unacceptable for run-time use. Another, high performance storage medium could be tried. If the board accepts DDR memory, it is an ideal solution.

### 7.2.10 Clock Management

The system was not designed with power consumption in mind. There can be found opportunities for lowering the power envelope. For static power, there is little we can do. The Zynq platform does have programmable voltage regulators but they are global to the PL and therefore we cannot power down a single reconfigurable partition.

However, it is possible to reduce dynamic power consumption. Even without an external clock source, Zynq and ZynqMP can create and distribute four configurable clocks. Either by delivering multiple pre-set clocks or by using clock converters, it would be possible to offer more than one clock to an accelerator slot.

Depending user desire, policy or simply whether system is operating from battery, the clock source could be switched to save energy. Furthermore, it would be beneficial for more complex accelerator designs that may fail timing closure at maximum clock to be given the opportunity to run at a slower one.

### 7.2.11 Extending Heterogeneity

In our work, support for heterogeneity is minimal. It is simply offered with the notion that if a certain module variant doesn't fit, it's ok, we'll schedule the task somewhere else.

This model has to be extended and the scheduler must be aware of the slot size and the special resources it contains. We already take into account the slot parameters for defining a slot attractiveness, but if the scheduler had more information about the slots and/or the task demands, it could even affect its decision taking into account time slacks, power consumption, and even choose an appropriate slot data width.

### 7.2.12 Random-Access Model

We used the streaming model under the assumption it is the most appropriate for an FPGA, as they typically lack sufficient memory resources.

However, Xilinx recently presented UltraRAM, an on-chip memory resource for the UltraScale+ that offers significantly higher capacity but lower performance compared to BlockRAM. Additionally, we should add that a designer may implement a memory controller in the programmable logic. Xilinx offers the "Memory Interface

Generator” IP core to create such an interface to any supported FPGA. Altera/Intel offers hard blocks for memory controllers in their Arria V SoCs. As the manufacturers advance their capabilities in 2.5D CMOS, it is reasonable to assume that FPGAs will gain access to large on-chip memories.

Support for a random-access co-processor model will become significant. Furthermore, cache-coherency is possible and should be offered in such a model. For a discussion on cache-coherence in Zynq and ZynqMP, please see section 3.3.

### 7.2.13 Scaling to Multiple Boards

A very interesting possibility for exploration would be extending the system to more than one FPGA. A master-slave architecture could allow the master Zynq to utilize the PL of other boards, not necessarily equipped with a SoC. Essentially all control and main memory resides in a single node whereas all other FPGAs would be used to stream data to, process, and stream back the output to main node. If all boards are Zynq, a peer-to-peer architecture could be implemented, where any Zynq could reserve and utilize an accelerator slot from any other Zynq.

Exchanging control signals throughout all the boards would be a trivial problem which can be solved even using the FPGA pins exposed at the Pmod connectors. Higher level connectivity could be used, like Ethernet, but with a significant impact in latency. The high-speed data transport would be a more complex issue. As a physical layer for the link, single-ended or LVDS I/O pins can be used but if the FPGA contains serial transceivers they would be a superior choice. An AXI channel can pass over the Aurora link-layer protocol, which can utilize the FPGA transceivers. Xilinx offers two IPs for bridging an AXI Stream, the Aurora 8B/10B and Aurora 64B/66B.

It is worth to mention that Xilinx also offers the AXI Chip2Chip IP core. This core can utilize the Aurora IPs to allow bridging of the full, memory-mapped AXI. This would make possible for a processor to access memory buffers residing in another board, essentially creating a non-uniform shared memory system of FPGA nodes. However, this would not be just an extension but a radically new approach.

# Appendices



# Appendix A

## Partial Reconfiguration Scripts

The Partial Reconfiguration workflow, currently (early 2018) is not fully integrated at the Vivado GUI. In newer versions a designer may use the GUI to create an HDL-only design, but the use of IP Integrator is still not supported. For now, only the initial design can be created in Vivado GUI. After floorplanning, the synthesized result will be saved in a DCP and the main P.R. workflow has to be done with the help of TCL scripts.

Xilinx has implemented a sample design that uses a single reconfigurable module for implementing an imaging filter. It is an integrated embedded system, that accepts a video stream from a HDMI capable daughtercard connected via the FMC, it performs the filtering and it outputs it to an HDMI capable output. It is implemented and documented in both ISE (see [31]) and Vivado (see [30]).

From this application, the core TCL scripts were extracted, and the client script was rewritten to match our application.

All scripts were created in GNU/Linux and use the UNIX directory structure. They will not work in a Microsoft Windows environment.

### A.1 Creating static design Device Checkpoint

This script, run from within Vivado with the synthesized design open, will replace the instantiated accelerators with black-boxes and save the result to a DCP.

```
set top [string trim [get_cells *_i] "*_i"]
set modules [list [get_cells -hierarchical zcore16_?]\
  [get_cells -hierarchical zcore16_??]]
```

```

foreach module $modules { update_design -cell $module -black_box }
write_checkpoint -force dcp/base/$top

```

## A.2 Generating the Project Files

After static design floorplanning and synthesis, a project file must be created for each reconfigurable module version of each reconfigurable partition. These project files are a requirement for Xilinx TCL scripts.

```

#!/usr/bin/env sh

proj_dir="./base/"
proj_name="base"
top="zed_asym_cc_alt"

instance_name="zcore16"
solution="solution_16"

cores_dir="./cores"
INSTANCE_LIST=`printf "${instance_name}_%d " {0..15}`
CORE_LIST="loopback gauss sobel emboss outline sharpen negative contrast
          threshold"

rm -rf ./prj/* ./synth ./bit ./hd_visual ./impl ./dcp/*.dcp *.log \
      *.html *.xml vivado* fsm_encoding.os

design_dir="${proj_dir}/${proj_name}.srcs/sources_1/bd/${top}"
for instance in $INSTANCE_LIST
do
    module_name=`grep -E ${top}.*${instance}$ ${design_dir}/hdl/${top}.v |
                gawk '{print $1}'`
    module_file="${design_dir}/ip/${module_name}/synth/${module_name}.v"

    for core in $CORE_LIST
    do
        core_files="${cores_dir}/${core}/${solution}/impl/ip/hdl/verilog/*"
        for file in ${core_files} ${module_file}
        do

```

```

        [ ! -f ${file} ] && echo "File ${file} was not found, please fix
        the script!" && exit
    echo "verilog xil_defaultLib ${file}" >> ./prj/${instance}_${core}
    }.prj
done
done
done

```

## A.3 TCL Client Script

This script is the client to Xilinx partial reconfiguration TCL scripts. It has to be modified to match the FPGA and hardware design of the designer. It was made to be as flexible as possible for future adaptation to other projects.

```

#!/usr/bin/env vivado -mode batch -source
# Xilinx-provided scripts
set tclDir "./tcl"
source $tclDir/design_utils.tcl
source $tclDir/log_utils.tcl
source $tclDir/synth_utils.tcl
source $tclDir/impl_utils.tcl
source $tclDir/hd_floorplan_utils.tcl

### FPGA part number
set zedboard "xc7z020c1g484-1"
set zcu102 "xczu9eg-ffvb1156-2-i"
set part $zedboard
check_part $part

### Setup Variables
# set tclParams [list <param1> <value> <param2> <value> ... <paramN> <value>]
set tclParams [list hd.visual 1]

### Workflow control
set run.rmSynth 0
set run.prImpl 1
set run.prVerify 0
set run.writeBitstream 0
set run.flatImpl 0

```

```

### Report and DCP controls - values: 0-required min; 1-few extra; 2-all
set verbose      1
set dcpLevel     1

### Output Directories
set synthDir     "./synth"
set implDir      "./impl"
set dcpDir       "./dcp"
set bitDir       "./bit"
set srcDir       ""

#####
### Top Definition
#####

set proj_dir     "./base/"
set proj_name    "base"
set top          "zed_asym_cc_alt"

set top_dcp      "${dcpDir}/base/${top}.dcp"
set top_xdc [list "${proj_dir}/${proj_name}.srcs/${top}/new/pblocks.xdc" \
                  "${proj_dir}/${proj_name}.srcs/${top}/new/io.xdc" ]

set static       "${top}_static"
add_module $static
set_attribute module $static moduleName      $top
set_attribute module $static top_level       1
set_attribute module $static synthCheckpoint $top_dcp

#####
### RP Module Definitions
#####

set core_basename "zcore16"
set core_easiest  "loopback"
set core_hardest  "contrast"
set cores_with_alt_settings [list "gauss" "sobel"]

```

```

set core_list [list "gauss" "loopback" "contrast" "sobel" "sharpen" "emboss"
    "outline" "negative" "threshold"]
set pblock_list [list 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15]

# Heterogeneous case: Define which modules require big partitions...
set core_big_list [list "gauss" "sobel" "sharpen" "emboss" "outline"]
# ...and which partitions cannot fit them
set pblock_small_list [list 4 5 10 11 14 15]
# Homogeneous case: $core_big_list is equal to $core_list and
# pblock_small_list is empty
### set core_big_list $core_list
### set pblock_small_list [list]

#####
### Configuration generation
#####

foreach core $core_list {
    # The hardest to route module shall be used to define the partition pins
    # between static and reconfigurable regions. It will be the "initial
    # configuration".
    set partition_list [list [list $static $top [expr {$score == $score_hardest
        ? "implement" : "import"}] ] ]

    foreach pblock $pblock_list {
        # The generic reconfigurable module,
        # in the form of ${module}_${pblock}
        set instance "${core_basename}_${pblock}"

        # The logic implemented in that reconfigurable module
        set variant "${instance}_${core}"

        # HDL modules created by Vivado have the form of
        # ${design_name}_${module}_n_m, usually n=[0,1,2...] and m=0,
        # but if module was copied-pasted in IP-Integrator,
        # naming is not well predictable and therefore we do not attempt
        # to guess it. The prj file has discovered it by parsing
        # the top module, therefore it will be correct.
    }
}

```

```

set module_name [exec /bin/grep "${instance}_-" ./prj/${variant}.prj |
sed s@.*/@@ | sed s/.v$//]

# Assume two sizes of pblocks. If the logic is `big` and the
# pblock is `small`, then use a simple placeholder logic.
# The generated bitstream will be discarded.
if {(((${core} in ${core_big_list}) && (${pblock} in ${
pblock_small_list})))} {
    lappend partition_list [list "${instance}_${core_easiest}" "${top
}_i/${instance}" "implement"]
} else {
    lappend partition_list [list ${variant} "${top}_i/${instance}" "
implement"]
}

# Create Xilinx TCL variables for this module variant
add_module $variant
set_attribute module $variant moduleName ${module_name}
set_attribute module $variant prj      "./prj/${variant}.prj"
set_attribute module $variant xdc     "${proj_dir}/${proj_name}.srcs/
sources_1/bd/${top}/ip/${module_name}/constraints/${core_basename
}_ooc.xdc"
set_attribute module $variant synth ${run.rmSynth}
}

# Define configuration. There will be one configuration for each
# module design, that will be implemented in all reconfigurable
# partitions. If the design is too big for the module,
# placeholder was assigned in previous step
set config "config_${core}"
add_implementation $config
set_attribute impl $config top      $top

# Implementation parameters.
# Xilinx scripts were created for Vivado 2014.2 and therefore some
# newer options are recognized or some older may no longer be valid.
# Modify tcl/implementation.tcl if that problem arises.

set_attribute impl $config opt_directive "Explore"

```

```

if {${core} in ${cores_with_alt_settings}} {
    set_attribute impl $config place_directive "ExtraTimingOpt"
    set_attribute impl $config phys_directive "Explore"
} else {
    set_attribute impl $config place_directive "ExtraPostPlacementOpt"
    set_attribute impl $config phys_directive "AlternateFlowWithRetiming
        "
}
set_attribute impl $config route_directive "Explore"

# Xilinx PR script parameters. The user-configurable ones (ie run.*) are
# defined above
set_attribute impl $config pr.impl 1
set_attribute impl $config implXDC ${top_xdc}
set_attribute impl $config impl ${run.prImpl}
set_attribute impl $config partitions ${partition_list}
set_attribute impl $config verify ${run.prVerify}
set_attribute impl $config bitstream ${run.writeBitstream}
set_attribute impl $config cfgmem.pcap 1
}

# Configuration is done, call Xilinx PR scrips
source $tclDir/run.tcl
exit

```

## A.4 Partial Bitstream Manipulation

The Xilinx TCL scripts will generate a large number of files, including all partial bitstreams. This script will install them to our software framework, after first renaming according to the naming conventions set by the system library and compressing them with the XZ compression tool.

```

#!/usr/bin/env bash
TARGET="../../yocto/meta-local/recipes-bsp/zdma-firmware/files"

rm -f $TARGET/*

for f in bit/*pcap*bin

```

```
do
  core=`echo $f | awk -F_ {'print $2'}`
  pblock=`echo $f | awk -F_ {'print $4'}`
  if ls bit/write_cfgmem_config_${core}_*_${pblock}_${core}_pcap.log 1> /
    dev/null 2>&1; then
    xz --check=crc32 --lzma2=dict=512KiB --stdout $f > $TARGET/${core}.${pblock}.bin.xz
  fi
done

install -m 644 bit/config_loopback.bit ../../image/download.bit
```

# Appendix B

## HLS Compiler Scripts

### B.1 Generating and Exporting an Accelerator Module

```
#!/opt/Xilinx/Vivado_HLS/2017.2/bin/vivado_hls
set cores [list gauss sobel emboss outline sharpen contrast threshold
            loopback negative]
set sizes [list 16 64]

foreach core $cores {
    foreach size $sizes {
        set fd [open "generated.h" "w"]
        puts $fd "#pragma once\n#define FORCE_DSP48\n#define CORE_NAME zcore$
                {size}\ntypedef uint${size}_t axi_data_t;"
        close $fd
        open_project -reset $core
        set_top zcore$size
        add_files ${core}.cpp
        add_files -tb testbench.cpp
        add_files -tb sample.jpg
        add_files -tb csim/build/out.jpg

        open_solution -reset "solution_${size}"
        set_part {xc7z020clg484-1}
        ##set_part {xczu9eg-ffvb1156-2-i}
```

```
create_clock -period 6 -name default

csynth_design
export_design -rtl verilog -format ip_catalog \
  -description "ZDMA Core $core/${size}" \
  -vendor "tuc" \
  -version "3.6" \
  -display_name "ZDMA Core $core/${size}"
close_project
}
}
exit
```

# Glossary

## ACE

*AXI Coherency Extensions*: A cache coherent port in AMBA AXI version 4. It offers full (two-way) coherency between a processor and a peripheral with caches. 17, 28, 33, 149, 151, *see* ACE-Lite

## ACE-Lite

*AXI Coherency Extensions, Light*: A lightweight version of the ACE protocol. It offers only IO (one-way) coherency between a processor and a peripheral without caches. 17, *see* ACE

## ACP

*Accelerator Coherency Port*: An AXI port that offers IO-Coherency to a non cache-aware AXI Master. 17, 24, 27, 28, 32, 33, 154

## AFI

*AXI FIFO Interface*: FIFO buffers placed in silicon just after the HP ports. Their purpose is to smooth out traffic in order to make DDR access more efficient. 18, 25

## AHB

*Advanced High-performance Bus*: A high-performance, single-channel multiple-master shared bus, introduced with AMBA version 2. 16, 149

## AHB-Lite

*Advanced High-performance Bus, Light*: A lightweight version of AHB, introduced with AMBA version 2. It simplifies the protocol by allowing only one

master in the bus. 21

## **AMBA**

*Advanced Microcontroller Bus Architecture*: A family of interconnect protocols that originates from the ARM microcontrollers but now it is used widely in modern ARM SoCs, including the FPGA-SoCs from Xilinx. 15–18, 21, 87, 149, 150

## **APB**

*Advanced Peripheral Bus*: A low-complexity, low-performance shared-bus, defined in the original specification of AMBA. 21

## **APU**

*Application Processing Unit*: The hardware unit that encompasses the application processors (ie the Cortex-A cores), the SCU and the cache. 28

## **AXI**

*Advanced eXtensible Interface*: A high-performance burst capable protocol introduced in AMBA version 3. In AMBA version 4 it was extended to support a burst size of up to 256, up from 16 in version 3. It does not support cache coherency. 9, 11, 15–25, 29, 30, 33, 34, 36, 37, 39, 42, 43, 66, 114, 132, 149, 150, 152, *see* AXI-Lite

## **AXI-Lite**

*Advanced eXtensible Interface, Light*: A lightweight version of the AXI protocol. It does not support burst transactions. 10, 16, 18–21, 29, 39, 43, 66, 87, 88, 112–115, 132, *see* AXI

## **AXI-Stream**

A streaming version of AXI. It has no notion of address spaces and therefore it exchanges no addressing information. This results in both higher performance and (much) lower complexity. 9, 10, 16–20, 30, 35, 43, 112, 114, 115, *see* AXI

## **beat**

A transfer of an elementary datum through the Full AXI or AXI-Lite channel. 16, 129, 151

**BPD**

*Full Power Domain:* A small power domain in UltraScale+ that contains the most basic components, like the oscillator, the RTC, etc. *see* power domain

**BRAM**

: A specialized type of on-chip FPGA memory resource with one cycle access latency. Each tile may be used as a single 36kibit element (BRAM36) or as two 18kibit (BRAM18). Optionally, BRAM may be configured as a FIFO and is capable of ECC. *see* 16, 33, 37, 38, 49, 50, 58–60, 104, 157

**buddy allocator**

The standard Linux kernel memory allocator. It is called so after the allocation algorithm it implements. 83, 89, 90

**burst**

A sequence of AXI Beats that transmits the transaction payload. Does not include handshaking and address information.. 16, 18, 21, 29, 150

**CCI**

*Cache Coherent Interconnect:* A cache-coherent interconnect by ARM. It can provide both full (two-way) cache coherency between the processor and a peripheral with caches or IO (one-way) coherency to cache-less peripherals. Used in UltraScale+ devices. 27, 28, 32, 33, 154, *see* IO Coherency

**CHI**

*Coherent Hub Interface:* An evolutionary step from ACE, this protocol the newer multi-core SoCs. It is not present in UltraScale+. 17, *see* ACE

**CLB**

*Configurable Logic Block:* The fundamental configurable block in an FPGA. It consists of a few Slices (the number depends on the architecture) and a switch matrix that provides access to the general routing matrix. 58, 157, *see* slice

**CMA**

*Contiguous Memory Allocator*: A Linux kernel facility that allows the allocation of indefinitely large physically contiguous memory. 83, 84, 89

**DAP**

*Device Access Port*: An external debug interface to an ARM core. It can debug a running system without the intervention of the CPU. It is available in Xilinx FPGA-SoCs. 24

**DCP**

*Device Check-Point*: The saved state of a design. It may be at post-synthesis, post-placement or post-routing. 46, 47, 139

**devc**

A configuration controller for the Zynq-7000. It contains an AXI to PCAP bridge and therefore is the intermediary for programming the FPGA during partial reconfiguration. 24, 33

**distributed RAM**

*Distributed RAM or LUT RAM*: Memory implemented in generic LUTs that are distributed over all the FPGA silicon area. 37, 152, *see* BRAM & LUT

**DRC**

*Design Rule Checking*: A process of design verification in order to conform to the requirements of the implementing technology. 49, 50

**DSP**

A specialized FPGA resource that implements an integer pre-adder, a multiplier and an accumulator. In 7-series a DSP48E1 is implemented, where the pre-adder is 25 bits wide, the multiplier is 25 by 18 bits wide, and the accumulator is 48 bits wide (hence the name). The UltraScale/UltraScale+ feature a more advanced version, the DSP48E2, which features a 27 by 18 bit wide multiplier. 49, 50, 58, 60, 157

**DT**

An hierarchical device structure that describes hardware topology, originally developed by Sun for the system. 68, 76, 77, 81, 84, 85, 88, 91, 92, 99

**DTO**

An extension to the Flattened Device Tree that allows the dynamic (run-time) modification of the kernel's live Device Tree. 99, 100, *see* DT

**fabric**

The programmable structural units of an FPGA. 13, 14, 21, 24, 68, 159

**FDT**

A Device Tree representation used by Linux that may also be passed to the kernel by the bootloader, thus supporting systems that do not implement . 5, 51, 68, 69, 72, 75, 76, 99, 133, *see* DT & Open Firmware

**FIFO**

First-In First-Out, a method of queueing access, however almost universally refers to the buffering hardware that implements this policy. 18

**FPD**

*Full Power Domain*: The UltraScale+'s power domain where the A53 cores reside, along with the high-speed peripherals (PCIe, SATA, GPU, etc). 25, 27, 29, *see* power domain

**GP**

*General Purpose port*: An AXI compatible port, found in Zynq-7000 but not in UltraScale+. It is present both as a slave to PS and as a master. It is 32 bit wide and runs at up to 150MHz. *see* M\_GP & S\_GP

**GPIO**

General Purpose I/O, ie unstructured data port. The handshaking, synchronization and error correction is left as a responsibility of the communicating endpoints. 21, 66

**GSR**

*Global Set/Reset*: A globally routed signal that forces logic to quiescence while the device is being programmed. 65

**hard-IP**

A non-programmable hardware component implemented in silicon. *see*

**HLS**

*High Level Synthesis*: A Xilinx tool that can compile an algorithm expressed in C/C++ to a hardware description language (Verilog) ready for synthesis by the standard toolchain. 112, 113, 115

**HP**

*High Performance port*: A high-performance non-coherent port that interfaces the PS with the PL in Xilinx Zynq FPGA-SoCs. In Zynq-7000 it can be configured as a 32 or 64 bit port, running at up to 150MHz. In Zynq UltraScale+ it can also be configured as a 128 bit port and the maximum frequency is raised to 333MHz. 18, 22, 24, 25, 27, 29, 31, 32, 37–39, 54, 118, 119, 149, 154

**HPC**

A coherent version of HP, present in UltraScale+. The coherency is provided by the CCI. It is destined to replace the ACP in many usage scenarios. 27, 32, 33, *see* HP & ACP

**HPM**

*High Performance Master port*: A high-performance non-coherent port of UltraScale+. It offers equivalent performance to HP ports but it a master port to the PL. In Zynq-7000 this functionality was offered by the M\_GP ports. 27–30, *see* M\_GP

**ICAP**

*Internal Configuration Access Port*: The port to the FPGA configuration interface from the PL side. 157, *see* PCAP

**IO Coherency**

IO Coherency, or one-way coherency, is weaker form of coherency where the peripheral can snoop the processor but the processor cannot snoop the peripheral. That is, the peripheral can read directly from the processor caches while writing to memory will automatically invalidate them. The processor cannot, therefore the peripheral may either do not possess any caches at all, or they are non-coherent and manually managed. 17, 22, 27, 32

**LPD**

*Full Power Domain:* The UltraScale+'s power domain where the R5 cores reside, along with the I/O peripherals and the OCM memory. 27, 28, 30, *see* power domain

**LUT**

*Look-Up Table:* A programmable function generator. It decides  $n$  outputs according to  $m$  inputs over constant time. The number of the inputs, the outputs and the legal configurations are architecture dependant. Essentially, it is a bit-addressable array of SRAM memory cells that stores the boolean function truth table. 54, 66, 110, 159

**M\_GP**

*General Purpose Master port:* A General Purpose port that is a master from the PS. In Zynq-7000 there are available two of such ports. 24, 27, 29, 154, *see* GP

**memory locking**

Marking a memory region as not swappable. The locked pages can still be migrated, therefore a soft page fault may still occur. *see* migration & memory pinning

**memory pinning**

Marking a memory region as not moveable. That is, the pages may not be migrated or swapped out. Access to this memory region is guaranteed not to produce any page fault. 83, *see* migration & memory locking

**MGT**

*Multi Gigabit Transceiver*: A programmable high-speed serial interface. 57, 64

**migration**

The move of an allocated page to a different PFN while updating reference to maintain consistency. This is useful in NUMA systems, where if a process is moved to another node, its allocated pages may also be transparently moved at a memory bank which has closer physical proximity to the new node. 83

**OCM**

*On-Chip Memory*: A small on-chip SRAM in the PS of both Zynq-7000 and UltraScale+. Its primary role is a low-latency synchronization point between the ARM cores. 22, 24, 28–30, 155

**OOO synthesis**

*out-of-context synthesis*: A bottom-up hierarchical approach in design synthesis. Each design module is synthesized independently of the final design. This approach disables optimizations across the boundaries of the synthesized module as it has no knowledge of the final design. This characteristic is essential to a P.R. design, however it is also used in normal workflow as it greatly reduces total synthesis time by re-synthesizing only the affected logic. 46, 47, 125

**Open Firmware**

A computer firmware standard, originally created by Sun and commonly implemented in PowerPC and SPARC architectures. 68, 133

**partition pin**

A logical pin and physical site of a P.R. design that serves as a connection point between the static and the reconfigurable logic. 47–50, 107

**pblock**

*Partition Block*: A group of physical FPGA resources. It may be dynamically reconfigurable or not. If it is, the architecture will impose several shape, content

and placement restrictions. If it is not, it acts as its assigned module container but it may as well share logic from other modules. 42, 46–50, 57, 58, 60, 65, 107, 158

## **PCAP**

*Processor Configuration Access Port*: The port to the FPGA configuration interface from the PS side. In Xilinx FPGA-SoCs it is the default port from boot-up. In any time, the processor may relinquish the control to Internal Configuration Access Port (ICAP), as well as reclaim it back. It is not available on the non-SoC FPGAs. 33, 152, *see* ICAP

## **PFN**

*Page Frame Number*: An index to the physical memory where a page is physically stored. 156

## **PLPD**

*Full Power Domain*: The UltraScale+'s power domain that contains the programmable logic, including the high-speed transceivers. *see* power domain

## **power domain**

A logical partition of the UltraScale+ system that can be individually isolated and powered. A power cut-off in one power domain, even accidental, can be configured to not affect the correct functional behavior of the others. There are four power domains in total. 151, 153, 155, 157, *see* PLPD, FPD, LPD & BPD

## **reconfigurable frame**

The smallest reconfigurable physical region of the FPGA. In 7-series it is one element (CLB, BRAM or DSP) wide by one clock region high. In UltraScale+ it is a pair of DSPs or a single BRAM/DSP tile and their neighboring CLBs. 49, 58, 59

## **reconfigurable logic**

Logic that is part of a reconfigurable module. 46, *see* reconfigurable module

**reconfigurable module**

An hierarchical design module that is defined as partially reconfigurable. 46–50, 65, 66, 112–114, 139, 140, 157, 158

**reconfigurable partition**

A physical region of the FPGA, defined by pblocks, that instantiates a single reconfigurable module. 11, 46–50, 52, 54, 57–61, 64, 65, 67, 77, 78, 99, 100, 109, 112, 114, 120, 124, 134, 135, 140, *see* pblock & reconfigurable module

**RPU**

*Real-time Processing Unit*: The hardware unit that encompasses the real-time processors (ie the Cortex-R cores). 27, 28, 33

**S\_GP**

*General Purpose Slave port*: A General Purpose port that is a slave to the PS. In Zynq-7000 there are available two of such ports. 25, 33, 54, 119, *see* GP

**scatterlist**

A software construct in the Linux kernel that abstracts the DMA scatter-gather functionality. Depending the architecture it may optimize transfers by coalescing or by processing the list in hardware if the DMA controller can support it. 8, 83, 87

**SCU**

*Snoop Control Unit*: The logic that implements cache coherency at the processor side. 24, 27, 28, 33, 150

**SEU**

A bit flip in memory caused by ionizing radiation penetrating the semiconductor and releasing charge inside a transistor diffusion terminal. It is non-destructive for the semiconductor itself but if the FPGA configuration memory is affected, its operation will be altered. 66

**slice**

The “Slice” in Xilinx terminology or “Adaptive Logic Module” in Altera’s, is the most basic group of configurable elements. It contains a few LUTs, fixed logic such as multiplexers and fast carry propagation logic, as well as some memory elements (flip-flops). An FPGA architecture may define several slice types that co-exist in the same chip. 151, *see* LUT

**soft-IP**

A programmable hardware component implemented in the FPGA fabric. 13, 18, *see*

**TNS**

*Total Negative Slack*: A metric of how far the timing closure is. It represents the sum of negative slacks of all paths that fail timing constraints. 50

**transaction**

A complete set of handshaking, addressing, data transfer and acknowledge on an AXI channel. 16, 40, 151

**VFS**

Virtual File System (VFS), An abstraction layer between a concrete file system and the rest of the operating system. All file access is routed to the VFS before it reaches the kernel block layer. 101, 159



# Bibliography

- [1] A. Agne, M. Happe, A. Keller, E. Lübbers, B. Plattner, M. Platzner, and C. Plessl, “ReconOS: An operating system approach for reconfigurable computing,” *IEEE Micro*, vol. 34, pp. 60–71, Jan 2014.
- [2] K. Eguro, “SIRC: An extensible reconfigurable computing communication API,” in *2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, pp. 135–138, May 2010.
- [3] G. Marcus, W. Gao, A. Kugel, and R. Männer, “The MPRACE framework: An open source stack for communication with custom FPGA-based accelerators,” in *2011 VII Southern Conference on Programmable Logic (SPL)*, pp. 155–160, April 2011.
- [4] M. Jacobsen, D. Richmond, M. Hogains, and R. Kastner, “RIFFA 2.1: A reusable integration framework for FPGA accelerators,” *ACM Trans. Reconfigurable Technol. Syst.*, vol. 8, pp. 22:1–22:23, Sept. 2015.
- [5] R. Bittner, “Speedy bus mastering PCI express,” in *22nd International Conference on Field Programmable Logic and Applications (FPL)*, pp. 523–526, Aug 2012.
- [6] J. Gong, J. Chen, H. Wu, F. Ye, S. Lu, J. Cong, and T. Wang, “EPEE: An efficient PCIe communication library with easy-host-integration property for FPGA accelerators (abstract only),” in *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, FPGA ’14, (New York, NY, USA), pp. 255–255, ACM, 2014.
- [7] Xilinx, *LogiCORE IP Virtex-6 Integrated Block for PCI Express – User Guide*.

- [8] Xilinx, *Virtex-5 FPGA Integrated Endpoint Block for PCI Express Designs – User Guide*.
- [9] Xilinx, *7 Series FPGAs Integrated Block for PCI Express v3.3 – LogiCORE IP Product Guide*.
- [10] Xilinx, *UltraScale Devices Gen3 Integrated Block for PCI Express v4.4 – LogiCORE IP Product Guide*.
- [11] Xilinx, *Virtex-6 FPGA Integrated Block for PCI Express – User Guide*.
- [12] Xilinx, *DMA/Bridge Subsystem for PCI Express v4.1 – Product Guide*.
- [13] Xilinx, *Resource Utilization for DMA/Bridge Subsystem for PCI Express (PCIe) v4.1*.
- [14] D. de la Chevallerie, J. Korinth, and A. Koch, “ffLink: A lightweight high-performance open-source PCI express Gen3 interface for reconfigurable accelerators,” *SIGARCH Comput. Archit. News*, vol. 43, pp. 34–39, Apr. 2016.
- [15] J. Korinth, D. d. l. Chevallerie, and A. Koch, “An open-source tool flow for the composition of reconfigurable hardware thread pool architectures,” in *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, pp. 195–198, May 2015.
- [16] TU-Darmstadt, “TaPaSCo - The Task Parallel System Composer.”  
[git.esa.informatik.tu-darmstadt.de/tapasco/tapasco](http://git.esa.informatik.tu-darmstadt.de/tapasco/tapasco).
- [17] K. Vipin, S. Shreejith, D. Gunasekera, S. A. Fahmy, and N. Kapre, “System-level FPGA device driver with high-level synthesis support,” in *2013 International Conference on Field-Programmable Technology (FPT)*, pp. 128–135, Dec 2013.
- [18] K. Vipin and S. A. Fahmy, “DyRACT: A partial reconfiguration enabled accelerator and test platform,” in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–7, Sept 2014.

- [19] 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)*, pp. 1–9, Aug 2016.
- [20] C. Vatsolakis and D. Pnevmatikatos, “RACOS: Transparent access and virtualization of reconfigurable hardware accelerators,” in *2017 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pp. 11–19, July 2017.
- [21] Xilinx, *Zynq-7000 All Programmable SoC – Technical Reference Manual*.
- [22] Xilinx, *Zynq UltraScale+ Device Technical Reference Manual*.
- [23] Xilinx, *AXI Interconnect v2.1 – LogiCORE IP Product Guide*.
- [24] Xilinx, *AXI DMA v7.1 – LogiCORE IP Product Guide*.
- [25] G. Charitopoulos, “Implementing a run-time system manager on partially reconfigurable FPGA systems,” Master’s thesis.
- [26] A. Aljaani, *ZedboardOLED IP*.
- [27] A. Tull, “Reprogrammable hardware under linux,” in *Embedded Linux Conference*, 2015.
- [28] P. Antoniou, “Transactional device tree & overlays,” in *Embedded Linux Conference*, 2015.
- [29] M. Fischer, “What’s new with FPGA manager,” in *FOSDEM*, 2018.
- [30] Xilinx, *Partial Reconfiguration of a Hardware Accelerator with Vivado Design Suite for Zynq-7000 AP SoC Processor*.
- [31] Xilinx, *Partial Reconfiguration of a Hardware Accelerator on Zynq-7000 All Programmable SoC Devices*.
- [32] Xilinx, *AXI4-Stream – Infrastructure IP Suite v2.2*.

- [33] K. Vipin and S. A. Fahmy, “FPGA dynamic and partial reconfiguration : a survey of architectures, methods, and applications,” *ACM Computing Surveys*, March 2018.