Πανεπιστημιο Θεσσαλιας

Σύνθεση Αρχιτεκτονικών για ένα Ετερογενές Σύστημα βασισμένο σε **FPGA**

Συγγραφέας:
Παναγιωτης Σκριμπονης

Επιβλέπον:
Νικολαος Μπελλας
Αναπληρωτής Καθηγητής

Συνεπιβλέπον:
Χρηστος Σωτηριου
Αναπληρωτής Καθηγητής

Πανεπιστημιο Θεσσαλιας

Σύνθεση Αρχιτεκτονικών για ένα Ετερογενές Σύστημα βασισμένο σε **FPGA**

Συγγραφέας:
Παναγιωτης Σκριμπονης

Επιβλέπον:
Νικολαος Μπελλας
Αναπληρωτής Καθηγητής

Συνεπιβλέπον:
Χρηστος Σωτηριου
Αναπληρωτής Καθηγητής

Εγκρίθηκε από την διμελή εξεταστική επιτροπή στις 2 Οκτωβρίου 2015

....

Επιβλέπον:
Νικολαος Μπελλας
Αναπληρωτής Καθηγητής

....

Συνεπιβλέπον:
Χρηστος Σωτηριου
Αναπληρωτής Καθηγητής

....
Παναγιώτης Σκριμπόνης
Διπλωματούχος Μηχανικός Ηλεκτρονικών Υπολογιστών, Τηλεπικοινωνιών και Δικτύων του Πανεπιστημίου Θεσσαλίας

UNIVERSITY OF THESSALY

High-Level Synthesis for an FPGA-based Heterogeneous System

Author:
Panagiotis SKRIMPONIS

Supervisor:
NIKOLAOS BELLAS
Associate Professor

Co-Supervisor:
CHRISTOS SOTIRIOU
Associate Professor

Acknowledgments

The work of this thesis has been one of the most significant academic challenge I have faced so far. Without the support, patience and guidance of the people around me this thesis would not have been completed.
First of all, my enormous debt of gratitude goes to my thesis advisor and mentor Professor Nikolaos Bellas. I am thankful for his knowledge and his belief in me, which provided the support required to overcome the initial problems encountered and keep working with passion. I would also like to thank my Professor Christos Sotiriou, for providing important feedback for my thesis. Throughout the period of my studies, they were there for me to actively support and guide me toward taking the best possible decisions.
I would also like to thank Professor Paolo Ienne and Dr. Muhsen Owaida for giving me the opportunity to work in such an inspiring research unit at EPFL and more importantly for their guidance, motivation and provisioning of the funding required for my research.
Especially, I would like to thank my friends, for supporting me every day. I totally believe that great achievements require friends to celebrate, and throughout my lifetime I was always trying to meet new people and make new friends.
Most importantly, I would like to thank my family and Maria for their support all of these years. There are no words that can express my gratitude and appreciation for all they have done for me. Without their patience and support, I would not have been able to finish my thesis. The least I can do in recognition is to dedicate this thesis to them.

To my family, Maria and my friends

Abstract

Recently, FPGA-based acceleration is becoming more appealing for application developers and cloud computing platform manufacturers. Integrating the FPGA in a cloud platform requires the ability to map simultaneously multiple applications on the FPGA and dynamically share its resources across multiple users. Every user sees a virtual space of the FPGA resources that is completely isolated and independent of other users. One of the problems we tact for the incorporation of FPGAs inside this heterogeneous system is the communication interface, which has to be high-throughput and low-latency, by using the PCIe link. As partial reconfiguration becoming a standard feature of modern FPGA devices, dynamic sharing of resources between multiple users is feasible.

Contents

1 Introduction **7**

2 Background **9**
2.1 FPGA . . . 9
2.1.1 Architecture . . . 10
2.2 Virtex Series . . . 12
2.3 Microblaze . . . 13
2.4 PCIe . . . 14
2.5 AMBA® AXI4 . . . 14

3 Architectural Exploration **16**
3.1 Blur . . . 17
3.2 Monte Carlo Simulation . . . 20
3.3 Results . . . 21
3.3.1 Blur . . . 21
3.3.2 Monte-Carlo simulation . . . 23

4 CPU-FPGA Communication **25**
4.1 RIFFA . . . 25
4.1.1 Architecture . . . 27
4.1.2 Hardware Interface . . . 29
4.1.3 C/C++ Interface . . . 30

5 Memory Management **31**
5.1 Virtual Memory . . . 31
5.2 Memory Management Unit . . . 32
5.3 Page Table Architecture . . . 33
5.4 Memory Allocation . . . 34
5.4.1 First Fit . . . 34
5.4.2 Best Fit . . . 34
5.4.3 Buddy Allocation . . . 34

5.5 Memory Deallocation 35

6 Application Manager **36**

6.1 Manager API 36

6.2 Application Scheduling 37

6.3 Partial Reconfiguration 38

6.3.1 Xilinx ICAP 38

7 Conclusion and Future Work **40**

List of Figures

1.1 Modern Cloud Computing CPU-FPGA Abstraction 7

2.1 FPGA Architecture 9
2.2 FPGA Architecture 10
2.3 VC707 Evaluation Board 11
2.4 VC707 Internal Architecture 12
2.5 Microblaze Internal Architecture 13
2.6 AXI4 Memory Mapped 15

3.1 Our System-Level Architecture 16
3.2 Baseline platform architecture used for our experimental evaluation. The dark shaded area shows the customizable logic. . 18
3.3 Duplication of the baseline architecture using two RIFFA channels 19
3.4 Using AXI streaming interface between RIFFA channels, the two accelerators and the DRAM. 19
3.5 Experimental performance results for the Blur, numbers above the bars are improvement over the optimized CPU implementation 21
3.6 Experimental performance results for the Monte-Carlo applications, numbers above the bars are improvement over the optimized CPU implementation 23

4.1 RIFFA Architecture 25
4.2 Graph of Bandwidth vs Transfer Size 26
4.3 RIFFA Upstream 28
4.4 RIFFA Downstream 29

5.1 Virtual Memory 31
5.2 Page Table Architecture 33

6.1 CPU-FPGA Application API 37

6.2 Partial Reconfiguation of an Application with Multiple Kernels on one Partial Reconfigurable Region 38

7.1 Our System-Level Architecture 40

Listings

3.1 Blur Horizontal kernel 17
3.2 Blur Vertical kernel 17
4.1 RIFFA C/C++ example 30
6.1 Host-CPU / FPGA-Manger Communication API 36
6.2 Application Kernel Scheduling 37

Glossary

FPGA Field Programmable Gate Array. 7–9

HLS High Level Synthesis. 7

HPC High Performance Computing. 7, 8

OTP One-Time Programmable. 10

Chapter 1

Introduction

Recent advances in FPGA technology and HLS methodologies have placed re-configurable systems on the road-map of heterogeneous HPC. FPGA accelerators offer superior performance, power and cost characteristics compared to a homogeneous Central Process Unit (CPU) based platform, and are more energy efficient than Graphics Process Unit (GPU) platforms. As a result FPGA-based acceleration is becoming more appealing for application developers and cloud computing platform manufacturers.

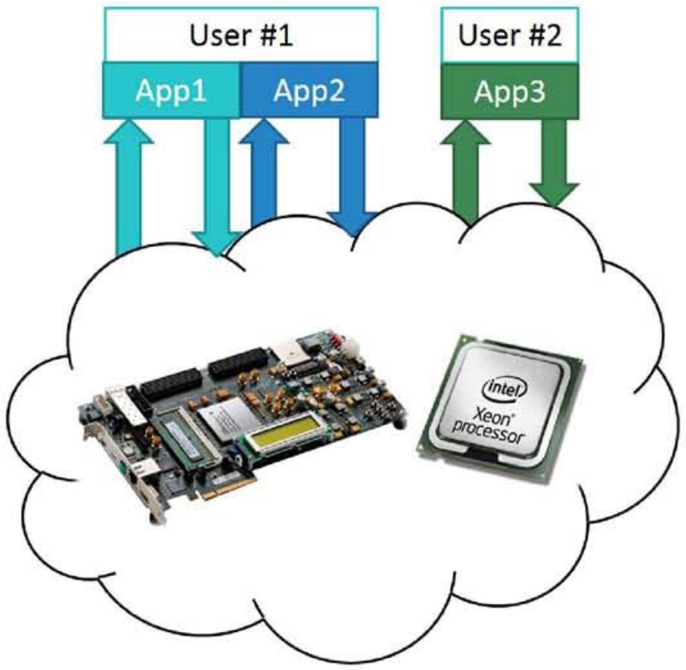

Figure 1.1: Modern Cloud Computing CPU-FPGA Abstraction

However, one big obstacle for the adoption of FPGA technology in a CPU-FPGA heterogeneous system is that FPGA programming still requires intimate knowledge of low-level hardware design and can lead to long development cycles. These characteristics make Hardware Description Languages (HDL) an unsuitable technology to implement a HPC application on an FPGA.
This problem can be easily tact with HLS tools, which allow designers to use high-level languages and programming models such as C/C++ and OpenCL. By elevating the hardware design process at the level of software development, HLS not only allows quick prototyping, but also enables architectural exploration inside a component. Most of the HLS tools offer optimization directives to inform the HLS synthesis engine about how to optimize parts of the source code. The HLS synthesizer implements a hardware accelerators optimized for performance or area according to these directives
Another big obstacle for the adoption of FPGA technology in a CPU-FPGA heterogeneous system, is that there is not straightforward way for integrating FPGA devices in a CPU-based platform.
This thesis intend to tact this specific problem, by designing infrastructure for FPGAs on the Cloud, a system-level architecture capable of executing multiple applications simultaneously, virtualizing and sharing the resources between the applications. At first we will take a set of applications in order to exploit different architectures, and find which the best architecture for our system. Then based on this architecture we will start developing our features

Chapter 2

Background

2.1 FPGA

FPGA are programmable semiconductor devices that are based around a matrix of Configurable Logic Blocks (CLB) connected through programmable interconnects. FPGAs allow designers to change their designs very late in the design cycle, even after the end product has been manufactured and deployed in the field.

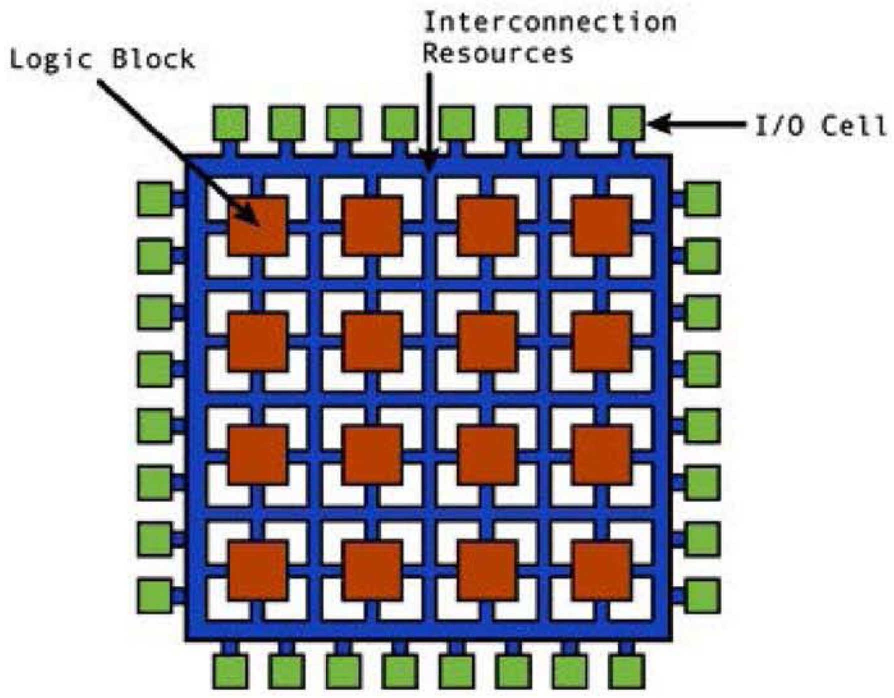

Figure 2.1: FPGA Architecture

FPGAs can be programmed to the desired application or functionality requirements, opposed to Application Specific Integrated Circuits (ASIC), where the device is custom built for the particular design. FPGAs are ideal for a wide variety of applications, from high-volume applications to state-of-the-art products. Each series of FPGA includes different features, such as embedded memory, Digital Signal Processing (DSP) blocks, high-speed transceivers, or high-speed I/O pins, to cover a broad range of end products. Although OTP FPGAs are available, the dominant type are SRAM-based which can be reprogrammed as the design evolves.

2.1.1 Architecture

The FPGA architecture consist of an array of CLB, a hierarchy of interconnects that enables the cooperation of those blocks,I/O banks which are able to support many I/O standards, DSP components for high-performance computation, memory elements like flip-flops and blocks of RAM and Clock Management Tiles (CMTs). A few modern FPGAs even include embedded microprocessors and related peripherals to form a System on a Chip (SoC).

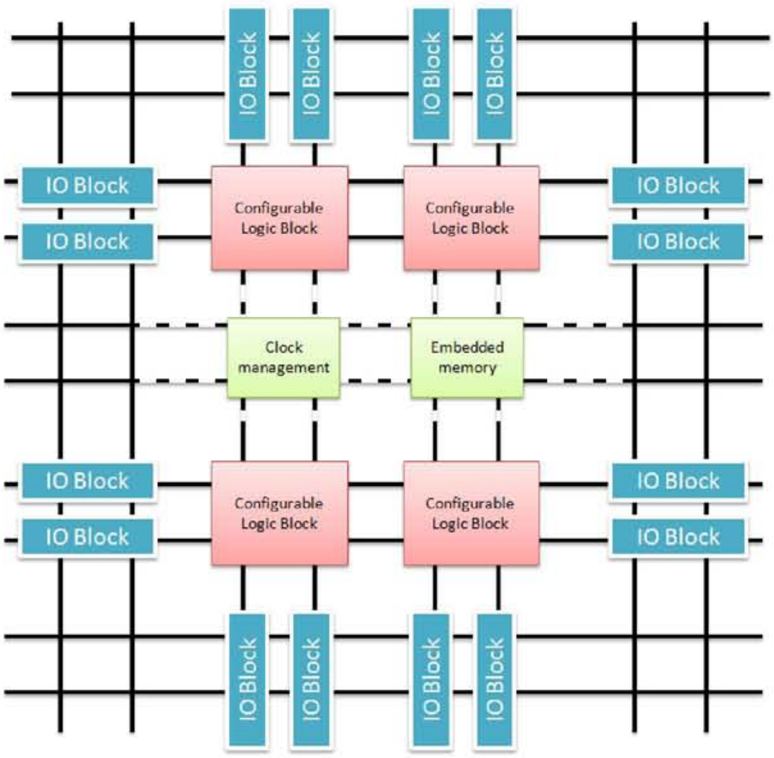

Figure 2.2: FPGA Architecture

The CLB is the basic logic unit in a FPGA. Exact numbers and features vary from device to device, but every CLB consists of a configurable switch matrix with 4 or 6 inputs, some selection circuitry (MUX, etc), and flip-flops. The switch matrix is highly flexible and can be configured to handle combinational logic, shift registers or RAM.
The routing channels are responsible for routing the signals between the clock, CLBs,RAMs and I/Os. In order for these routes to be optimal and fast, the routing task is hidden from the user and is completed solely by the tool, applying any optimization needed for the design.
The I/O features of an FPGA vary from device to device. Most of them support USB, video outputs; VGA or/and HDMI, audio lines in and out, Ethernet and connectors for many other features or devices such as cameras, sensors and many more. Digital clock management provides users the ability to manage the original clock generated from an oscillator on the FPGA and create new clocks, with lower or higher frequency.
Most FPGAs support very powerful soft-core and on-chip processors. With these abilities, these devices combine the software programmability of an embedded processor with the hardware programmability of an FPGA, resulting in outstanding system performance, and power efficiency.

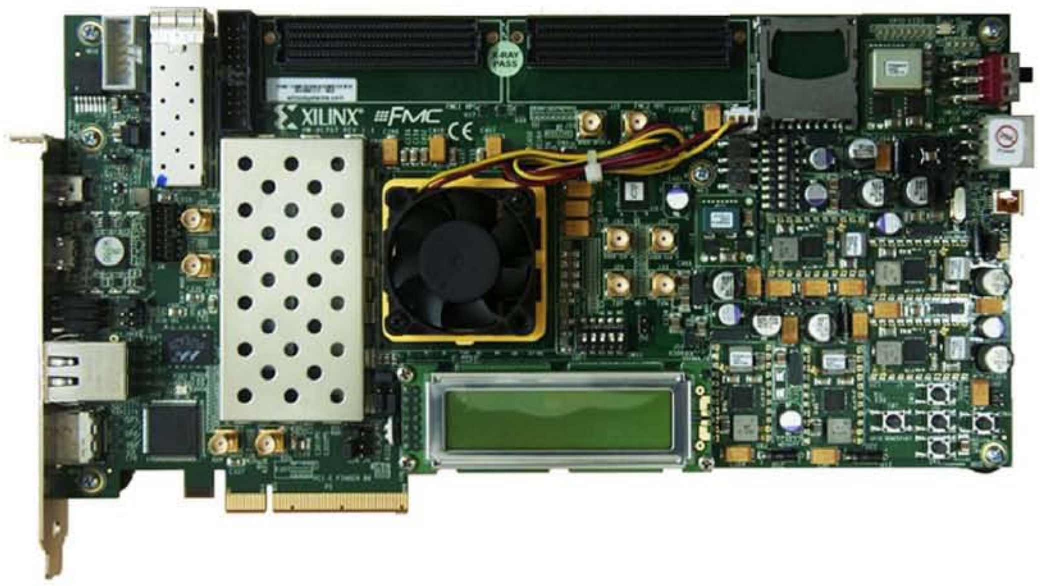

Figure 2.3: VC707 Evaluation Board

2.2 Virtex Series

The VC707 evaluation board for the Virtex®-7 FPGA provides a hardware environment for developing and evaluating designs targeting the Virtex-7 XC7VX485T-2FFG1761C FPGA. The VC707 board provides features common to many embedded processing systems, including a DDR3 SODIMM memory, an 8-lane PCI Express® interface, a tri-mode Ethernet PHY, general purpose I/O, and two UART interfaces. Other features can be added by using mezzanine cards attached to either of two VITA-57 FPGA mezzanine connectors (FMC) provided on the board. Two high pin count (HPC) FMCs are provided.

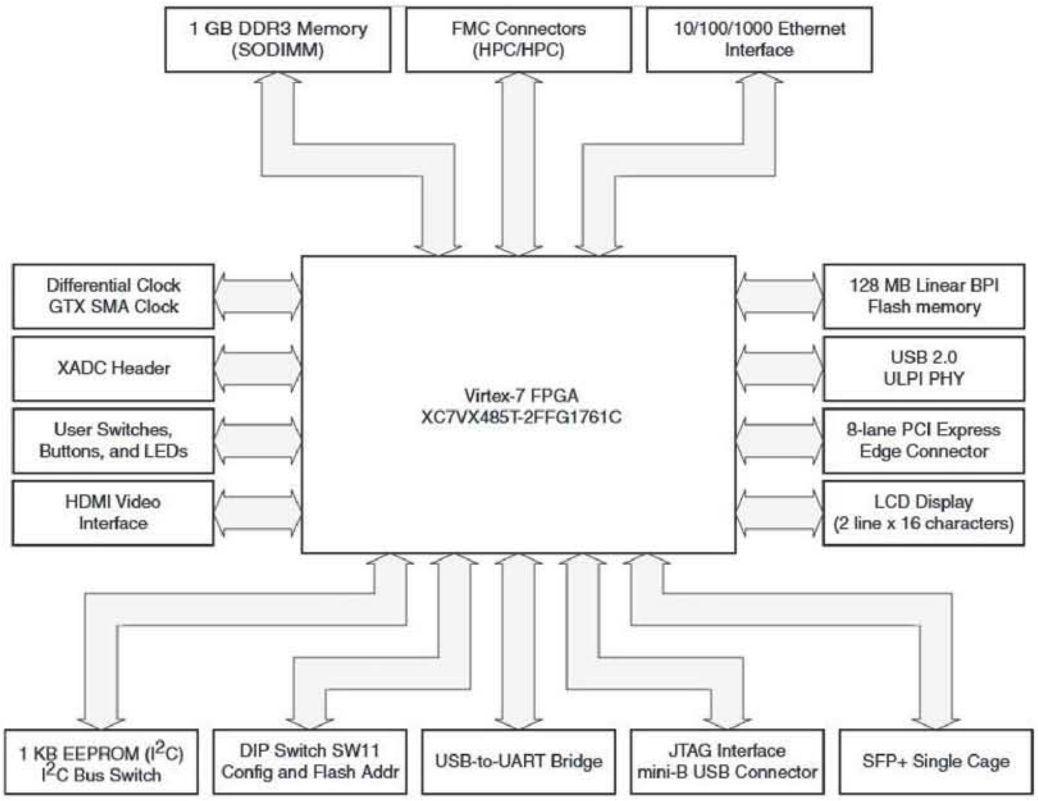
1 GB DDR3 Memory (SODIMM)
FMC Connectors (HPC/HPC)
10/100/1000 Ethernet Interface
Differential Clock GTX SMA Clock
XADC Header
User Switches, Buttons, and LEDs
HDMI Video Interface
Virtex-7 FPGA XC7VX485T-2FFG1761C
128 MB Linear BPI Flash memory
USB 2.0 ULPI PHY
8-lane PCI Express Edge Connector
LCD Display (2 line x 16 characters)
1 KB EEPROM (I2C) I2C Bus Switch
DIP Switch SW11 Config and Flash Addr
USB-to-UART Bridge
JTAG Interface mini-B USB Connector
SFP+ Single Cage

Figure 2.4: VC707 Internal Architecture

2.3 Microblaze

The MicroBlaze is a soft microprocessor core designed for Xilinx FPGAs from Xilinx. As a soft-core processor, MicroBlaze is implemented entirely in the general-purpose memory and logic fabric of Xilinx FPGAs.

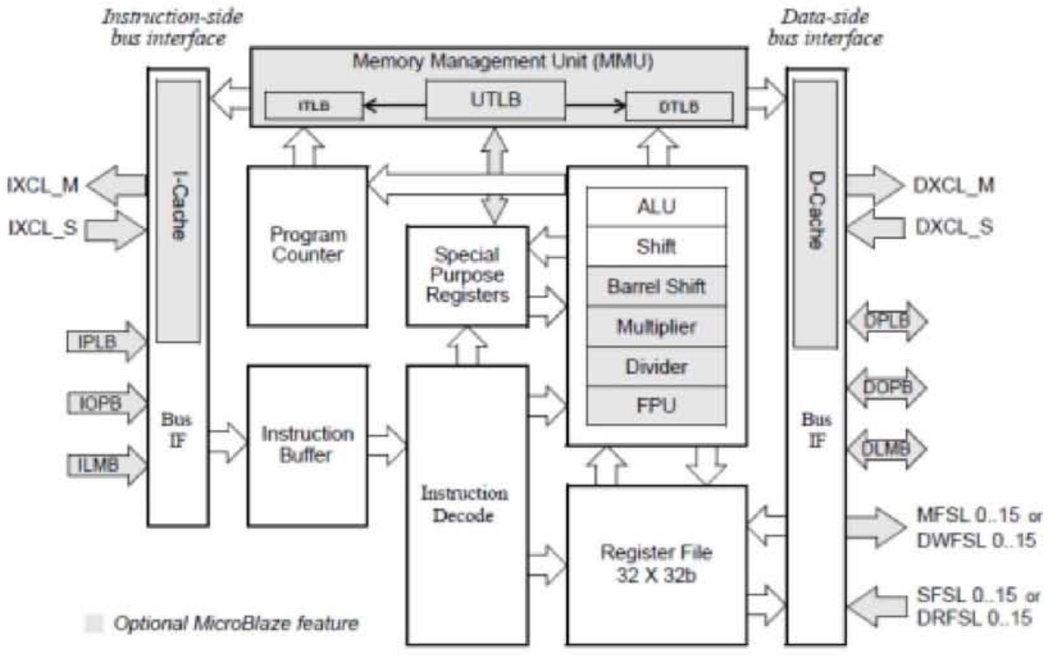

Figure 2.5: Microblaze Internal Architecture

The MicroBlaze soft core processor is highly configurable, allowing you to select a specific set of features required by your design. Such as cache size, pipeline depth (3-stage or 5-stage), embedded peripherals, memory management unit, and bus-interfaces can be customized. Microblaze is a critical component to our system because the application manager we developed runs on the Microblaze.

2.4 PCIe

PCI Express (Peripheral Component Interconnect Express), officially abbreviated as PCIe, is a high-speed serial computer expansion bus standard designed to replace the older PCI, PCI-X, and AGP bus standards. PCIe has numerous improvements over the older standards, including higher maximum system bus throughput, lower I/O pin count and smaller physical footprint, better performance scaling for bus devices, a more detailed error detection and reporting mechanism (Advanced Error Reporting, AER[1]), and native hot-plug functionality. More recent revisions of the PCIe standard provide hardware support for I/O virtualization.

2.5 AMBA® AXI4

Advanced Microcontroller Bus Architecture (AMBA)® Advanced eXtensible Interface 4 (AXI4) is the fourth generation of the AMBA interface specification from ARM®. AXI targets system designs with high clock frequencies. It has separate address/control and data phases and supports non-aligned data transfers using byte signals. Hosts can issue multiple addresses for more efficient bus utilization, and burst-based transactions only need to supply the start address.

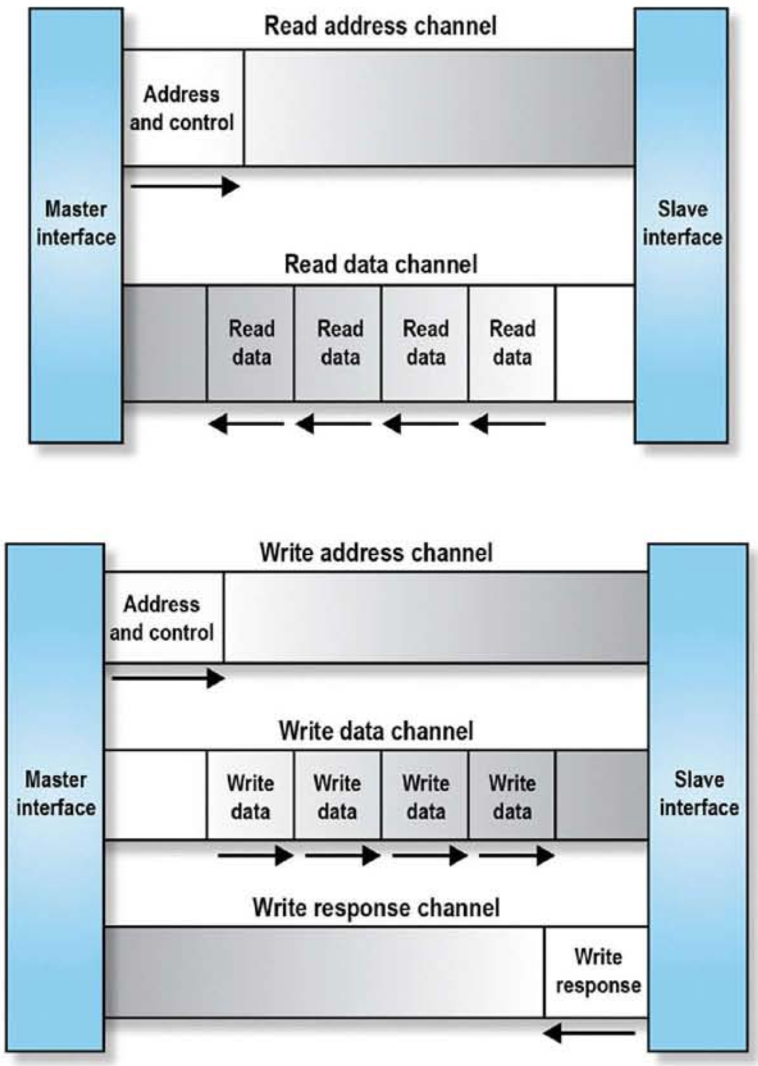

Figure 2.6: AXI4 Memory Mapped

The AXI architecture allows additional register stages so designers can provide timing closure.AXI is a host/client interface that can be extended using a switch or fabric. The AXI interconnect can be implemented in a number of ways with varying levels of performance and complexity. Interconnects can support one or more AXI masters. Obviously, a single master interconnect will be easier and less complex to implement.

Chapter 3

Architectural Exploration

In order to incorporate FPGAs as part of a modern heterogeneous system, we exploited the standardization of communication abstractions provided by modern high-level synthesis tools like Vivado HLS and SOpenCL to create our system. Our final system-level architecture is on Figure 3.5.

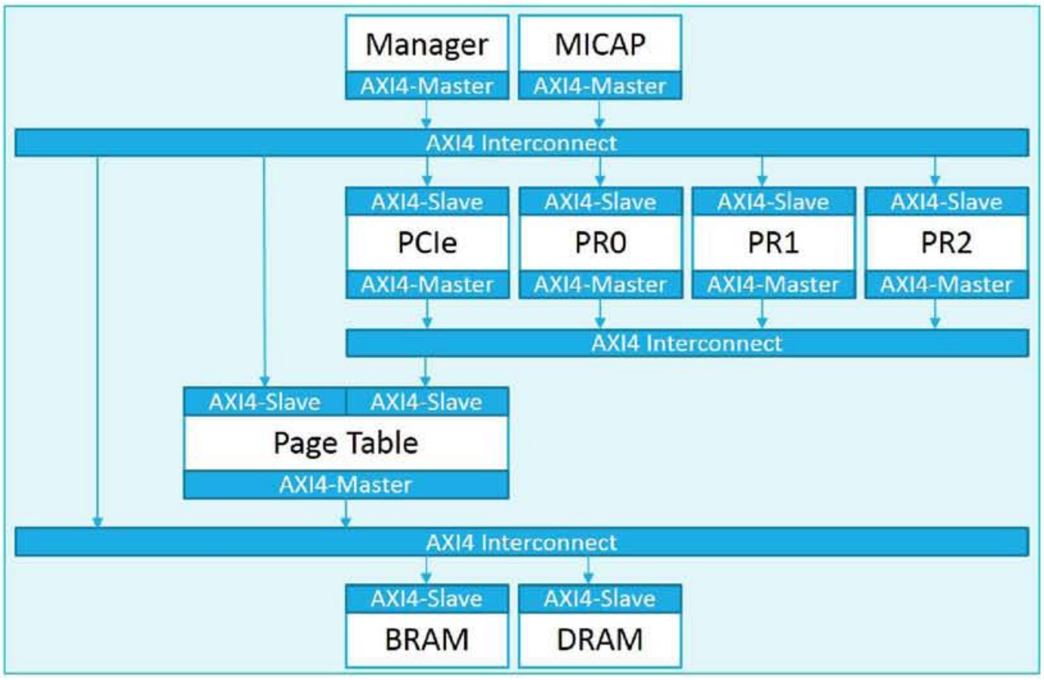

Figure 3.1: Our System-Level Architecture

3.1 Blur

Listing 1.1 and Listing 1.2 shows the pseudocode of the Blur filter, one of the applications under evaluation. The algorithm first applies a horizontal and then a vertical 3-tap low pass filter to an incoming image, temporarily storing the output of the horizontal filter to the memory. This pseudocode is optimized for a CPU execution, not for a hardware design, which leads to drawbacks when it is processed by HLS tools. This code results into two hardware accelerators, which have to communicate via a large memory implemented either via DRAM or as an on-chip BRAM(if the size of the BRAM is large enough).

The dark shaded logic of Figure 3.2 is generated by the Vivado toolset, based on instructions by the system developer. Two accelerators are instantiated and are connected with a BRAM memory through an AXI4-master interconnect. This baseline architecture is extended by automatically exploring the number and type of the various resources. Such resources include the accelerators in terms of throughput, area, latency and number. It also includes the bus structure and number of separate buses, the number and type of memories, and the interface to the Host unit.

In addition to the customizable part of the architecture, extra resources are required for communication with the Host unit. We use an open-source framework, RIFFA to provide an abstraction for software developers to access the FPGA as a PCIe-based accelerator

Listing 3.1: Blur Horizontal kernel

```
for(i = 0; i < Height; i++)
        for(j = 0; j < Width; j++)
                tmp(i, j) = (inp(i, j-1) + inp(i, j) +
                   inp(i, j+1)) / 3;
```

Listing 3.2: Blur Vertical kernel

```
for(i = 0; i < Height; i++)
        for(j = 0; j < Width; j++)
                out(i, j) = (tmp(i-1, j) + tmp(i, j) +
                   tmp(i+1, j)) / 3;
```

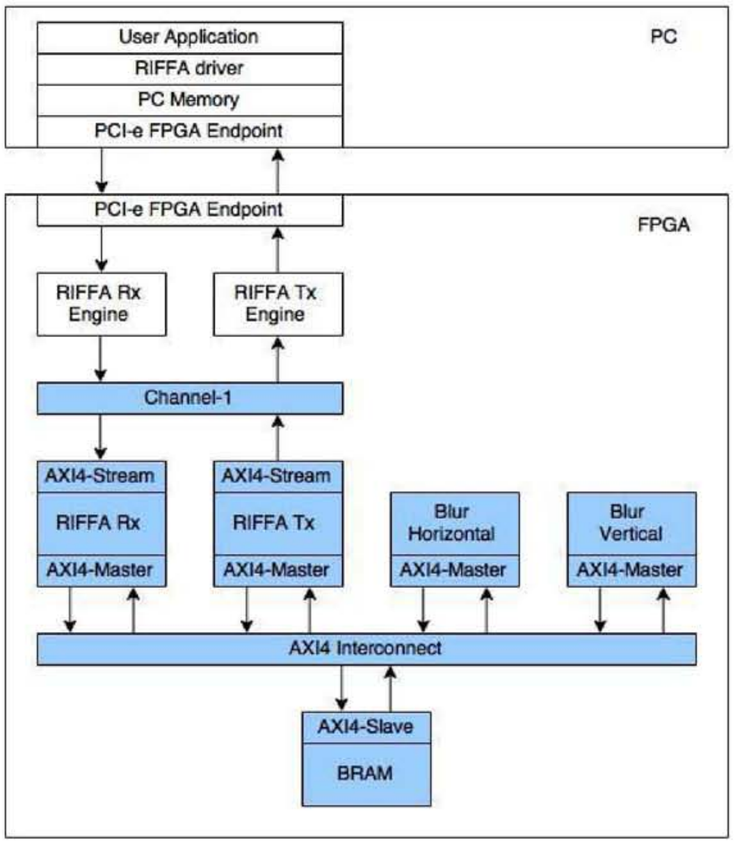

Figure 3.2: Baseline platform architecture used for our experimental evaluation. The dark shaded area shows the customizable logic.

The RIFFA hardware implements the PCIe Endpoint protocol so that the user does not need to get involved with the connectivity details of the accelerator. From the accelerator side, RIFFA provides a set of streaming channel interfaces that send and receive data between the CPU main memory and the customizable logic. On the Host unit, the RIFFA 2.0 architecture is a combination of a kernel device driver and a set of language bindings. RIFFA provides a very simple API to the user that allows for accessing individual channels for communicating data to the accelerator logic.
Figure 3.3 and Figure 3.4 shows two indicative architectural scenarios, expanding the baseline architecture of Figure 3.2. We can effectively duplicate the customizable logic using an extra RIFFA channel Figure 3.3. Even better, the streaming, point-to-point nature of the Blur application allows us to use the AXI4-Stream protocol to channel data between consumer and producers Figure 3.4. Some configurations may use external DDR3 memory (which includes an FPGA DDR3 memory controller) to be able to accommodate

larger images at the cost of increasing latency and worse performance.

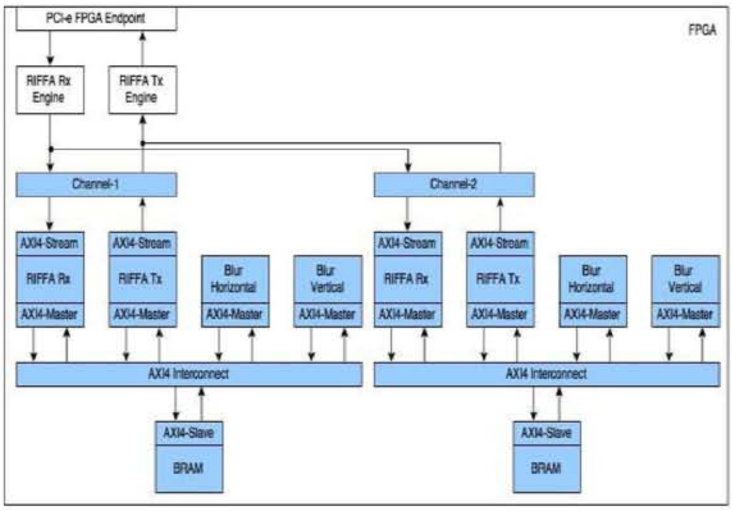

Figure 3.3: Duplication of the baseline architecture using two RIFFA channels

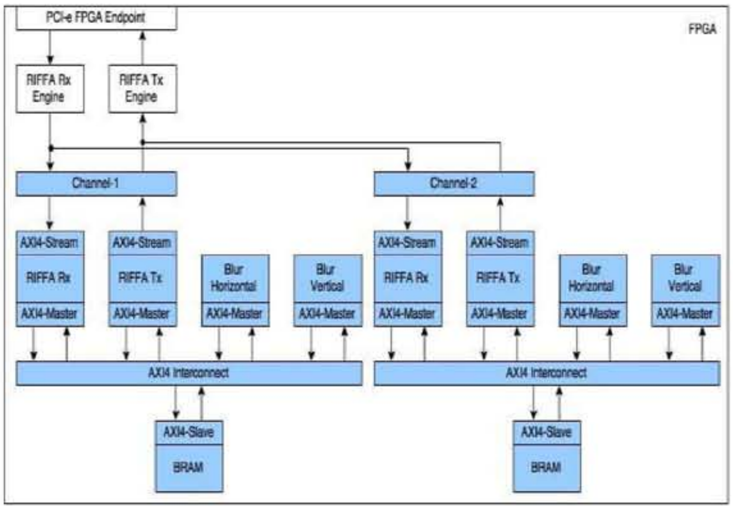

Figure 3.4: Using AXI streaming interface between RIFFA channels, the two accelerators and the DRAM.

To navigate through the large design space smartly, we devised a set of heuristics for making design decisions:

1. Keep data local as close as possible to the accelerator. The goal here is to minimize read/write latency between the accelerator and data memory.

2. Minimize shared resources between independent accelerators. Following this guideline helps eliminating collisions between multiple accelerators while accessing communication channels and memory ports.

3. Overlap data transfers with accelerators execution. The objective here is to minimize accelerators idle time waiting for data to be available. Our design space exploration approach starts from the baseline architecture of Figure 3.2. We then incrementally make design decisions while considering the aforementioned heuristics and evaluating the effects of the taken decisions on overall system performance.

3.2 Monte Carlo Simulation

Monte-Carlo simulation is a compute intensive kernel with minimal memory accesses. Hence the different implementation scenarios are made of different accelerator configurations and by instantiating multiple instances of the accelerator. In the baseline scenario, the accelerator is configured to perform a single walk of a single point per invocation, and a single accelerator is allocated (MC_1_walk_1_point_1_chnl). This scenario performs much worse than the CPU implementation. Double precision operations have larger latency on FPGAs than a CPU. Also, Vivado HLS libraries of trigonometric operators (sin, cos) are not pipelined and less efficient than their CPU counterparts. The strength of the FPGA is to perform more computations in parallel. Unfortunately, the MC kernel computations of a single walk are sequential and cannot be parallelized. As such, the FPGA baseline implementation performs badly. To improve performance we need to exploit coarser-grain parallelism across multiple points and walks. The second scenario allocates two accelerator instances to parallelize the computations of walks for a single point (MC_1_walk_1_point_2_chnl). It reduces the execution time to half, but still worse than the CPU. We need to allocate around 40 accelerator replicas to reach the CPU performance.
Another aspect to improve on is to minimize the accelerator invocation overhead by coarsening the computations granularity per a single accelerator instance. This allows for pipelining computations of multiple walks, which will have a strong impact on performance. The third scenario minimizes accelerator invocation overhead by configuring the accelerator to perform all the walks of a single point per invocation (MC_all_walks_1_point_1_chnl). The fourth scenario allocates five accelerators to parallelize computations (MC_all_walks_1_point_5_chnl) across multiple points. The last scenario saturates the FPGA resources and almost achieves near CPU performance.

3.3 Results

3.3.1 Blur

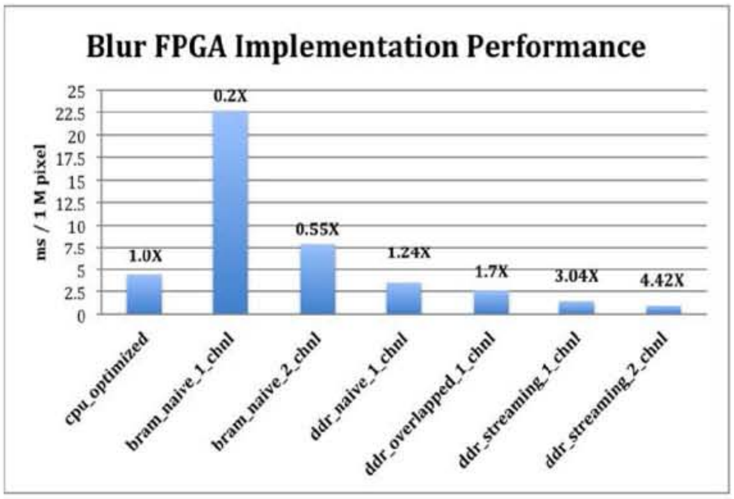

Figure 3.5: Experimental performance results for the Blur, numbers above the bars are improvement over the optimized CPU implementation

Six implementation scenarios are studied for the Blur application. The first scenario represents the baseline architecture in Figure 3.2 (bram_naive_1_chnl). The host CPU sends a single row for the horizontal blur kernel for processing and waits for the result from the accelerator before sending the next row until the entire image is processed. The same is done for the vertical blur, but here 3 rows are needed for the vertical blur to start. This scenario is reasonable when there is not enough on-chip or off-chip memory to save the whole image, then we partition the image into smaller partitions that fit on the available memory resources. Another version of this scenario (bram_naive_2_chnl) replicates the hardware of a single RIFFA channel on 2 channels to exploit parallelism in the Blur application. While the second scenario improves on the performance of the BRAM naive implementation it is still worse than that of the optimized CPU implementation. Sending small chunks of data over the PCIe is not efficient because we pay the overhead of initiating a PCIe read/write transaction many times. As a result, the PCIe transactions occupy two thirds of the total execution time.
Scenario three of the Blur (ddr_naive_1_chnl) makes use of the off-chip DDR to store the whole image instead of partitioning it into multiple portions. Using a DDR provides few benefits; The PCIe read/write transactions consume less time compared to the first scenarios because we eliminate the overhead

of initiating PCIe transactions. The second benefit of the DDR is that we do not need to send the horizontal blur output back to the host CPU, but keep it in the DDR for the vertical blur to process it, then write back to the host CPU the result of the vertical blur. As such, the third scenario achieves improvement over optimized CPU time.
To improve performance of scenario #3, we allow the horizontal blur to start as soon as the first row of the image is stored in the DDR and not wait for the whole image to be loaded. We also allow writing data back to the host CPU even before the vertical blur accelerator finishes execution. This is demonstrated in the fourth scenario (ddr_overlapped_1_chnl). The overlapping of accelerators execution with FPGA-Host data transfers is possible because of the regular access patterns of the blur kernels. While this scenario eliminates most data transfers overhead, moving data between DDR and the accelerators introduces a non-negligible overhead.
To improve further and minimize the DDR-Accelerator communication overhead, we use AXI-stream interfaces for horizontal and vertical blur accelerators (ddr_streaming_1_chnl, ddr_streaming_2_chnl). Instead of storing the image in the DDR, the horizontal blur accelerator uses AXI-stream interface to read data from the channel FIFOs and write result to the DDR. The vertical blur will read data from the DDR, process it and send the results directly to the RIFFA channel through an AXI-stream interface. In this scenario we eliminate 60% of the DDR-Accelerator data movements. This is possible because of the streaming nature of the blur kernels. This scenario achieves the best performance compared to the CPU implementation. Moreover, this scenario consumes less area than the DDR naive and overlapped implementations, which allows allocating more replicas of the accelerators to exploit parallelism and improve performance as the case in ddr_streaming_2_chnl.

3.3.2 Monte-Carlo simulation

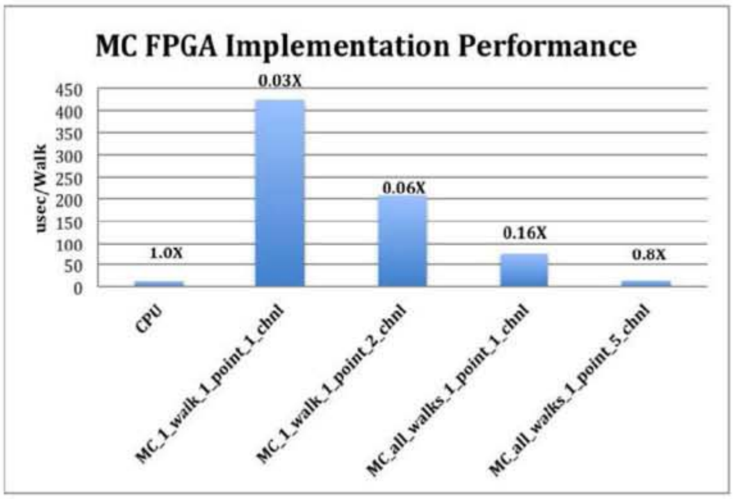

Figure 3.6: Experimental performance results for the Monte-Carlo applications, numbers above the bars are improvement over the optimized CPU implementation

MC is a compute intensive kernel with minimal memory accesses. Hence the different implementation scenarios are made of different accelerator configurations and by instantiating multiple instances of the accelerator. In the baseline scenario, the accelerator is configured to perform a single walk of a single point per invocation, and a single accelerator is allocated (MC_1_walk_1_point_1_chnl). This scenario performs much worse than the CPU implementation. Double precision operations have larger latency on FPGAs than a CPU. Also, Vivado HLS libraries of trigonometric operators (sin, cos) are not pipelined and less efficient than their CPU counterparts. The strength of the FPGA is to perform more computations in parallel. Unfortunately, the MC kernel computations of a single walk are sequential and cannot be parallelized. As such, the FPGA baseline implementation performs badly. To improve performance we need to exploit coarser-grain parallelism across multiple points and walks. The second scenario allocates two accelerator instances to parallelize the computations of walks for a single point (MC_1_walk_1_point_2_chnl). It reduces the execution time to half, but still worse than the CPU. We need to allocate around 40 accelerator replicas to reach the CPU performance.
Another aspect to improve on is to minimize the accelerator invocation overhead by coarsening the computations granularity per a single accelerator instance. This allows for pipelining computations of multiple walks, which will have a strong impact on performance. The third scenario minimizes

accelerator invocation overhead by configuring the accelerator to perform all the walks of a single point per invocation (MC_all_walks_1_point_1_chnl). The fourth scenario allocates five accelerators to parallelize computations (MC_all_walks_1_point_5_chnl) across multiple points. The last scenario saturates the FPGA resources and almost achieves near CPU performance.

Chapter 4

CPU-FPGA Communication

4.1 RIFFA

Reusable Integration Framework for FPGA Accelerators (RIFFA) is a simple framework for communicating data from a host CPU to a FPGA via a PCI Express bus. The framework requires a PCIe enabled workstation and a FPGA on a board with a PCIe connector. RIFFA supports Windows and Linux, Altera and Xilinx, with bindings in C/C++, Python, MATLAB and Java.

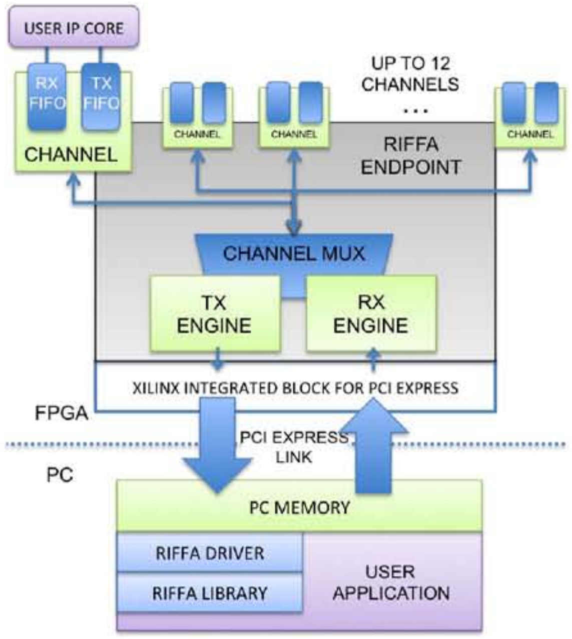

Figure 4.1: RIFFA Architecture

On the software side there are two main functions: data send and data receive. These functions are exposed via user libraries in C/C++, Python, MATLAB, and Java. The driver supports multiple FPGAs (up to 5) per system. The software bindings work on Linux and Windows operating systems. Users can communicate with FPGA IP cores by writing only a few lines of code.
On the hardware side, users access an interface with independent transmit and receive signals. The signals provide transaction handshaking and a first word fall through FIFO interface for reading/writing data to the host. No knowledge of bus addresses, buffer sizes, or PCIe packet formats is required. Simply send data on a FIFO interface and receive data on a FIFO interface. RIFFA does not rely on a PCIe Bridge and therefore is not subject to the limitations of a bridge implementation. Instead, RIFFA works directly with the PCIe Endpoint and can run fast enough to saturate the PCIe link.
RIFFA communicates data using direct memory access (DMA) transfers and interrupt signaling. This achieves high bandwidth over the PCIe link. In our tests we are able to saturate (or near saturate) the link in all our tests. The RIFFA distribution contains examples and guides for setting up designs on several standard development boards.

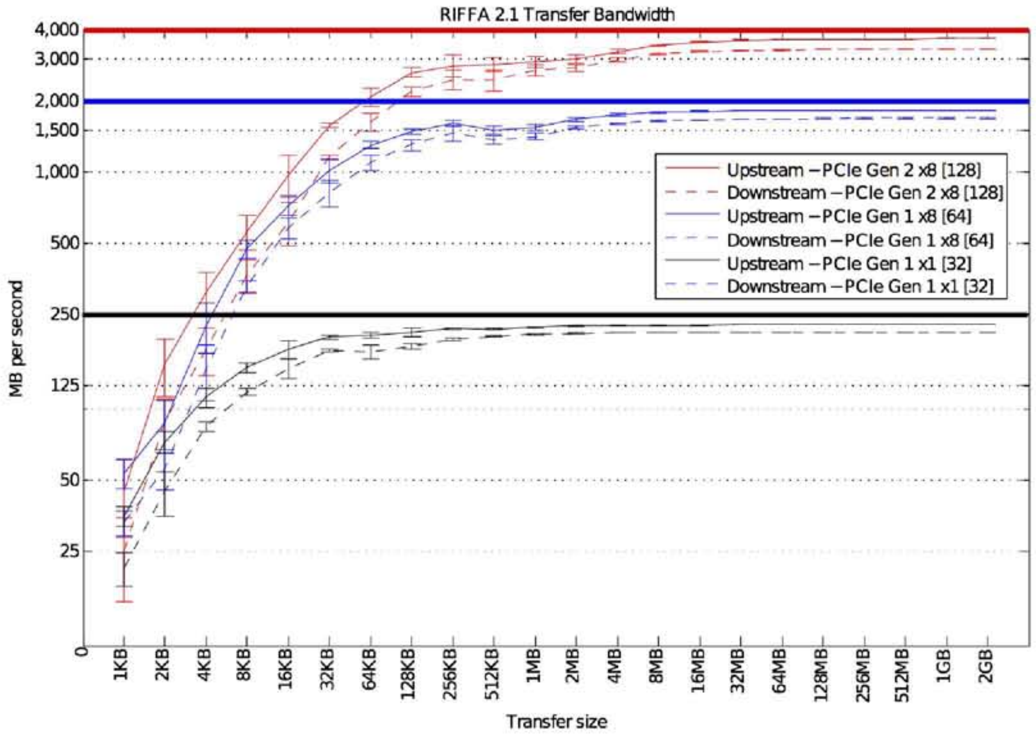

Figure 4.2: Graph of Bandwidth vs Transfer Size

RIFFA 2.2 is significantly more efficient than its predecessor RIFFA 1.0. RIFFA 2.2 is able to saturate the PCIe link for nearly all link configurations supported. Figure 4.2 shows the performance of designs using the 32 bit, 64 bit, and 128 bit interfaces. The colored bands show the bandwidth region between the theoretical maximum and the maximum achievable. PCIe Gen 1 and 2 use 8 bit / 10 bit encoding which limits the maximum achievable bandwidth to 80% of the theoretical. Our experiments show that RIFFA can achieve 80% of the theoretical bandwidth in nearly all cases. The 128 bit interface achieves 76% of the theoretical maximum.

4.1.1 Architecture

A sequence diagram for an upstream transfer is shown to the right An upstream transfer is initiated by the FPGA. However, they will not begin until the user application calls the user library function fpga_recv. Upon doing so, the thread enters the kernel driver and begins the pending upstream request. If the upstream request has not yet been received, the thread waits for it to arrive (bounded by the timeout parameter). On the diagram, the user library and device driver are represented by the single node labeled "RIFFA Library".
Servicing the request involves building a list of scatter gather elements which identify which pages of physical memory correspond to the receptacle byte array. The scatter gather elements are written to a shared buffer. This buffer location and content length are provided to the FPGA. Each page enumerated by the scatter gather list is pinned to memory to avoid costly paging. The FPGA reads the scatter gather data then issues write requests to memory for the upstream data. If more scatter gather elements are needed, the FPGA will request additional elements via interrupt. Otherwise, the kernel driver waits until all the data is written. The FPGA provides this notification, again via an interrupt.
After the upstream transaction is complete, the driver reads the FPGA for a final count of data words written. This is necessary as the scatter gather elements only provide an upper bound on the amount of data that is to be written. This completes the transfer and the function call returns to the application with the final count.

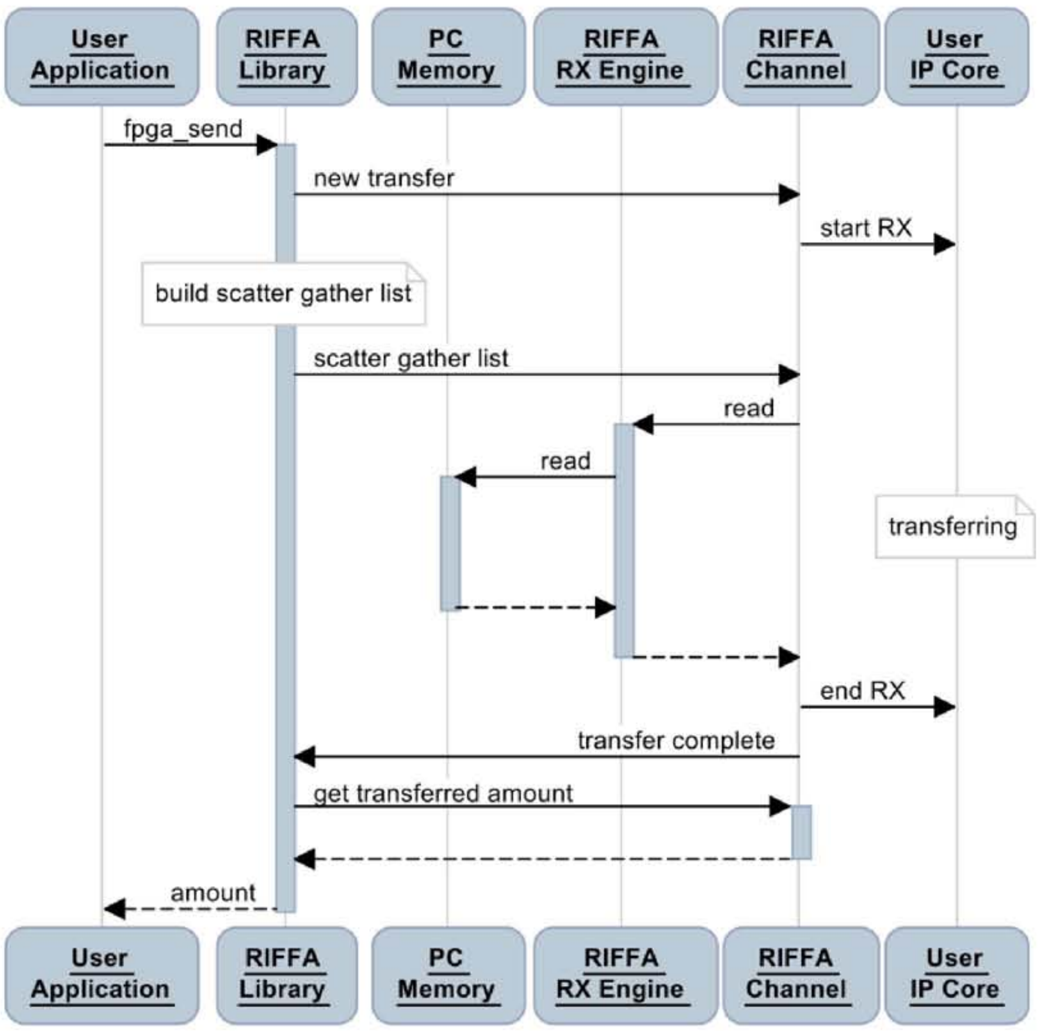

Figure 4.3: RIFFA Upstream

A similar sequence exists for downstream transfers. The left figure illustrates this sequence. In this direction, the application initiates the transfer by calling the library function fpga_send The thread enters the kernel driver and writes to the FPGA to initiate the transfer. Again, a scatter gather list is compiled, pages are pinned, and the FPGA reads the scatter gather elements. Each of the elements results in one or more read requests by the FPGA. The read requests are serviced and the kernel driver is notified only when more scatter gather elements are needed or when the transfer has completed.

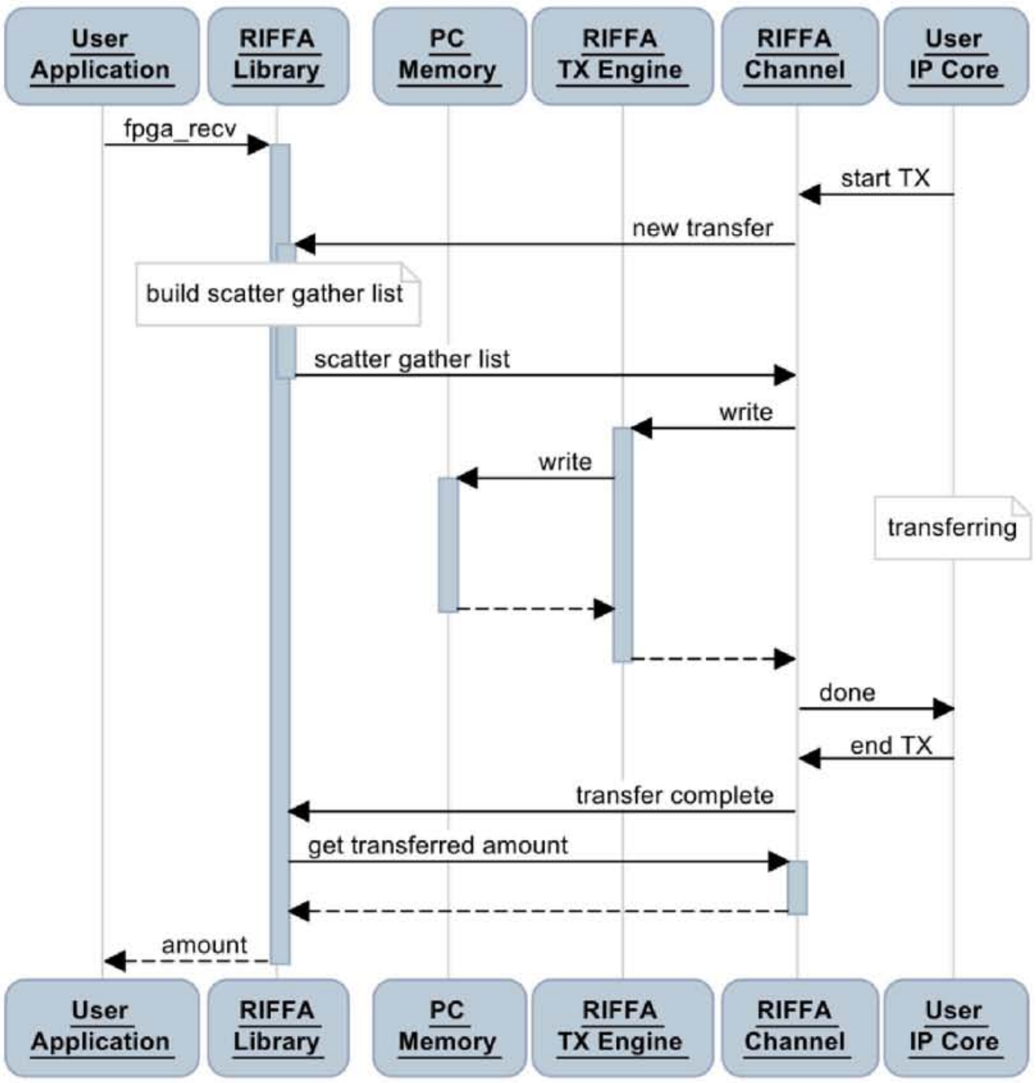

Figure 4.4: RIFFA Downstream

Upon completion, the driver reads the final count read by the FPGA. In error free operation, this value should always be the length of all the scatter gather elements. The final count is returned to the application.

4.1.2 Hardware Interface

A single RIFFA 2 channel has two sets of signals, one for receiving data (RX) and one for sending data (TX). RIFFA 2 has simplified the interface to use a minimal handshake and receive/send data using a FIFO with first word fall through semantics (valid+read interface). The clocks used for receiving and sending can be asynchronous from each other and from the PCIe interface (RIFFA clock). The table below describes the ports. The input/output designations are from your user core's perspective (i.e. the core(s) you write and connect to the RIFFA 2.0 channel).

4.1.3 C/C++ Interface

The software interface is provided by bindings for C/C++. After installation all bindings are available in their respective runtime environments. The API is based on the notion of channels. RIFFA 2 can be configured to support between 1 - 12 independent channels. Each channel connects to an IP core and can be addressed by specifying the channel number from the user application. The channels are independent and thread safe. At most one thread should be used to access a single channel.
The C/C++ bindings are used by including the ¡riffa.h¿ header file and linking with the -lriffa library. Below is a complete example and an API listing.

Listing 4.1: RIFFA C/C++ example

```
#include <stdio.h>
#include <stdlib.h>
#include "riffa.h"

#define BUF_SIZE ( 1 * 1024 * 1024 )
unsigned int buf[BUF_SIZE];

int main()
{
        fpga_t * fpga;
        int fid = 0;
        int channel = 0;

        fpga = fpga_open(fid);
        fpga_send(fpga, channel, (void *)buf, BUF_SIZE, 0, 1, 0);
        fpga_recv(fpga, channel, (void *)buf, BUF_SIZE, 0);
        fpga_close(fpga);
        return 0;
}
```

Chapter 5

Memory Management

Memory management is the act of managing computer memory at the system level. The essential requirement of memory management is to provide ways to dynamically allocate portions of memory to programs at their request, and free it for reuse when no longer needed. This is critical to our cloud computing system where more than a single process might be underway at any time. The quality of the memory manager can have an extensive effect on overall system performance.

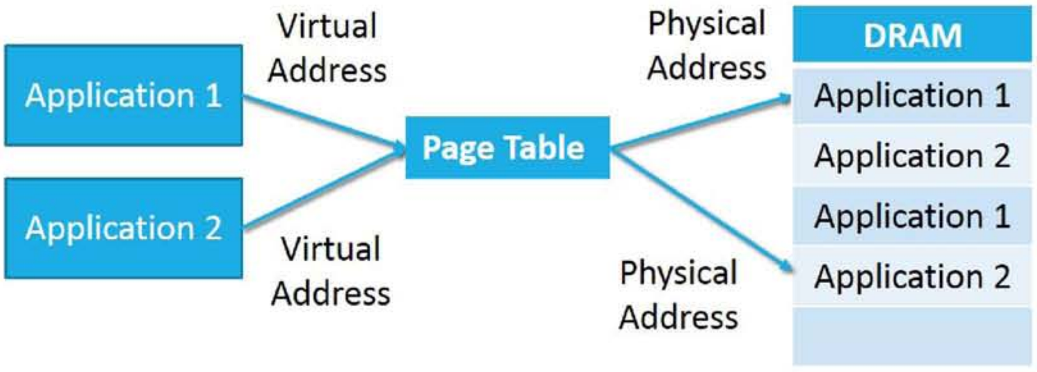

Figure 5.1: Virtual Memory

5.1 Virtual Memory

Virtual Memory is a memory management technique that is implemented using both hardware and software resources. It maps memory addresses used by an application, called virtual addresses, into physical addresses in the FPGA and DDR3 memory. Main storage as seen by an application appears as a contiguous address space or collection of contiguous segments.

Although, these address space is not contiguous in the FPGA and DDR3 memory. The memory manager that runs on the microblaze manages virtual address spaces and the assignment of real memory to virtual memory.
In a cloud computing system we need to have multiple applications running simultaneously on the same machine. Each application needs to be completely isolated and independent of the other applications. As a result, the applications will run in a Virtual Memory (VM) system. The applications will create virtual addresses which will be translated by the Memory Management Unit (MMU) into physical addresses that will be used to access the physical memory.
Address translation hardware in the FPGA, often referred to as a MMU, which automatically translates virtual addresses to physical addresses. Software within the operating system may extend these capabilities to provide a virtual address space that can exceed the capacity of real memory and thus reference more memory than is physically present in the computer.

5.2 Memory Management Unit

Page Tables are used to translate the virtual addresses seen by the application into physical addresses used by the hardware to process instructions; such hardware that handles this specific translation is often known as the memory management unit. Each entry in the page table holds a flag indicating whether the corresponding page is in real memory or not. If it is in real memory, the page table entry will contain the real memory address at which the page is stored. When a reference is made to a page by the hardware, if the page table entry for the page indicates that it is not currently in real memory, the hardware raises a page fault exception, invoking the paging supervisor component of the operating system.

5.3 Page Table Architecture

The internal architecture of the page table is simple and efficient in terms of area and performance. We use the axi side channel to identify if the memory request is made by the system, or by an application that runs on the partial reconfigurable regions. In order to store the translations of each application we use on-chip memory(BRAM). The rest of the AXI signals go directly to the register.

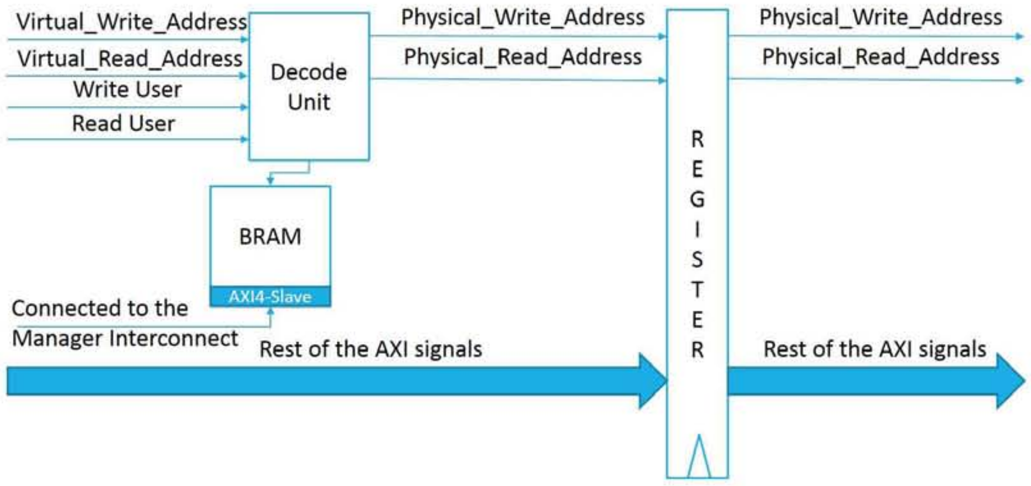

Figure 5.2: Page Table Architecture

Every time we have a new application request or a dynamic memory allocation request, the memory manager that runs on the microblaze has to update the page table with the new values. Also when an application finishes or we have a dynamic memory deallocation request we have to mark the proper translations of this application as not valid.

5.4 Memory Allocation

Memory allocation is the process of assigning blocks of memory on request. Typically the allocator receives a memory request from the PCIe controller or from the application and has to allocate the proper amount of memory. Our memory system is divided into pages, in order to make the allocation easier.

1. First Fit
2. Best Fit
3. Buddy Allocation

5.4.1 First Fit

In the first fit algorithm, the allocator keeps a list of free pages (known as the free list) and, on receiving a request for memory, scans along the list for the first pages that is large enough to satisfy the request. The first fit algorithm performs reasonably well, as it ensures that allocations are quick.

5.4.2 Best Fit

In the best fit algorithm, the allocator keeps a list of free blocks (known as the free list) and, on receiving a request for memory, scans along the list for the first block that is large enough to satisfy the request. If the chosen block is significantly larger than that requested, then it is usually split, and the remainder added to the list as another free block. The best fit algorithm performs reasonably well, as it ensures that allocations are quick.

5.4.3 Buddy Allocation

In a buddy system, the allocator will only allocate blocks of certain sizes, and has many free lists, one for each permitted size. The permitted sizes are usually either powers of two, or form a Fibonacci sequence (see below for example), such that any block except the smallest can be divided into two smaller blocks of permitted sizes. The main advantage of the buddy system is that coalescence is cheap because the “buddy” of any free block can be calculated from its address.

5.5 Memory Deallocation

Memory deallocation is the process of freeing blocks of memory on request. Typically the deallocator receives a request to deallocate a part or all the memory for a specific application. In order to do this we have to update the page table, and simply make the translations as invalid. We also have to change the control registers with the new limits.

Chapter 6

Application Manager

Our Application Manager runs on the microblaze, and is responsible for managing and sharing the resources between multiple applications. On the Manager we also run the scheduler for all the applications and the kernels of the applications.

6.1 Manager API

We implemented an API for the Host-CPU in order to communicate efficiently with the FPGA-Manager.

Listing 6.1: Host-CPU / FPGA-Manger Communication API

```
int app_req(int num_pr, int priority, int mem_size);
void app_snd_bit(int * bitstreams, int app_id);
void app_snd_sch(int * scheduling, int app_id, int last);
void app_snd_data(int * data_buf, int app_id, int last);
int app_rcv_data(int * data_buf, int app_id, int last);
```

First of all, we are able to send an application request from the Host-CPU to the FPGA. In the application request, we are requesting for a specific amount of memory, a number of minimum partial reconfigurable regions and the priority of the application. The return of this function is the answer of the manager, if the manager has enough space for the application then he replies with the application id, otherwise he replies negative.
Then if the reply of the manager is the application id, the application send the partial bitstreams, the scheduling micro-code and starting data. When the application finishes sending the bitstream, the scheduling information, and the starting data (if the application has) we indicating to the PCIe controller to start the application.

When the application finishes the execution on the FPGA, the application that runs on the Host-CPU in able to read data back. When the application is ready to read data back indicates to the PCIe controller that has read all the data in order the Garbage Collector that runs on the FPGA to free all the allocated memory of this application.

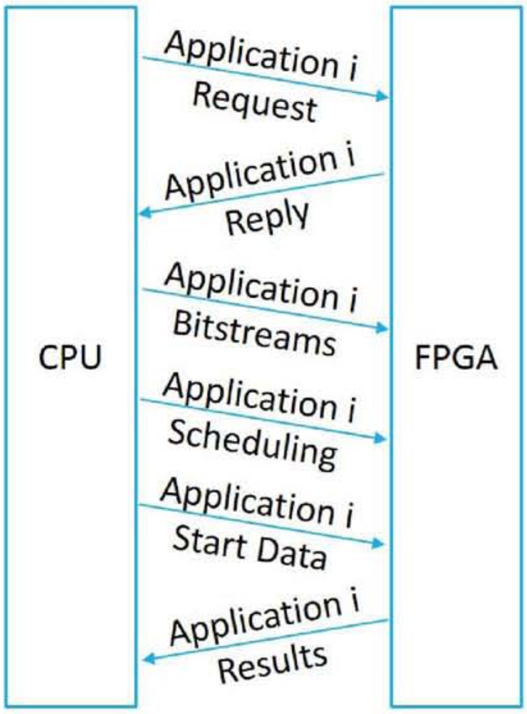

Figure 6.1: CPU-FPGA Application API

6.2 Application Scheduling

We have developed an application scheduler that has a predefined number of maximum applications, and maximum number of kernels per application. For each of this applications we have a queue with the scheduling instructions. The application scheduler checks the application queues if there are instructions that can be executed. If there are instructions of applications with high priority we execute them first. The scheduling is based on this simple instruction set:

Listing 6.2: Application Kernel Scheduling

```
void Execute_Kernel_and_Wait(int kernel_id);
void Execute_Kernel_and_Continue(int kernel_id);
void Allocate_Memory_Block(int block_size);
void Free_Memory_Block(int block_ize);
```

This minimal set of instructions we support allows the application developer to create his scheduling program and run his application. However, the complexity of controlling the data dependencies between kernels in on the application developer. We intend to change the way we do the scheduling, so that the application developer give us his control/data flow-graph, which is more simple than having to control all the data dependencies. With this approach we will add one more layer of abstraction for the application developer, and we will increase the performance gain, because we will able to start a kernel of an application when the data are ready.

6.3 Partial Reconfiguration

Partial Reconfiguration is the ability to dynamically modify blocks of logic by downloading partial bit files while the remaining logic continues to operate without interruption. Xilinx Partial Reconfiguration technology allows designers to change functionality on the fly, eliminating the need to fully reconfigure and re-establish links, dramatically enhancing the flexibility that FPGAs offer. The use of Partial Reconfiguration can allow designers to move to fewer or smaller devices, reduce power, and improve system upgradability. Make more efficient use of the silicon by only loading in functionality that is needed at any point in time.

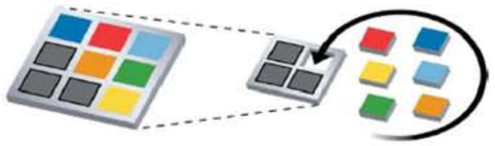

Figure 6.2: Partial Reconfiguation of an Application with Multiple Kernels on one Partial Reconfigurable Region

6.3.1 Xilinx ICAP

The AXI Hardware ICAP enables an embedded microprocessor, such as MicroBlaze, to read and write the FPGA configuration memory through the Internal Configuration Access Port (ICAP). This enables a user to write software programs that can modify FPGA circuit structure and functionality during the operation of the circuit.

The ICAP state machine (ISM) constantly monitors the read port of the FIFO for bitstream data. The FIFO read port is 32 bits wide, clocked at 100 MHz – the maximum clock frequency supported by the ICAP. As soon as the FIFO empty signal is de-asserted, the ISM fetches data from the FIFO and writes it to the ICAP. Since the FIFO depth is double the maximum PCIe read request size, the bitstream read from host memory can overlap with ICAP transactions, maximising reconfiguration throughput.
We have created an interrupt based DMA that transfers the partial bitstreams from the DRAM to the FIFO port of the the ICAP. As a result the microblaze does not have to wait for the partial reconfiguration to finish. Moreover, by using the dynamic reconfiguration port (DRP) we can dynamically overclock the ICAP in order to achieve better performance.

Chapter 7

Conclusion and Future Work

We proposed an infrastructure that intends to tact the problem of integrating the FPGAs inside a high-performance CPU-FPGA system, and particularly inside a cloud computing system. We do this by designing a system-level architecture capable of executing multiple applications simultaneously, virtualizing and sharing the resources between the applications.

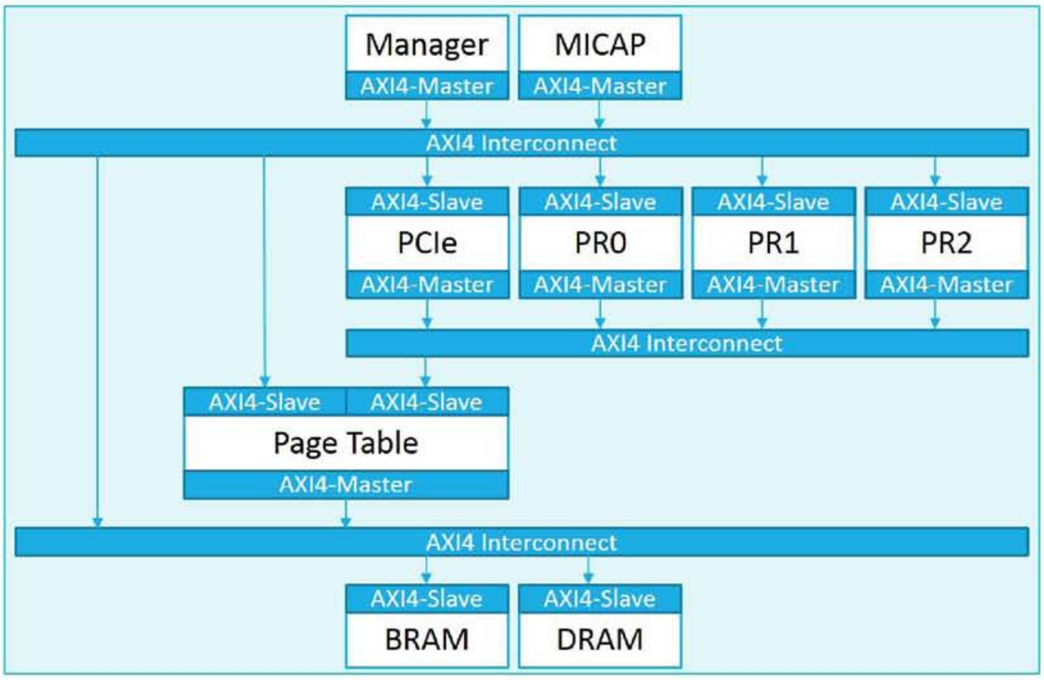

Figure 7.1: Our System-Level Architecture

The first part of my thesis we exploited many system-level architectures. The result of this work was published on the paraFPGA symposium on Septem-

ber,2016 with the title "Exploring Automatically Generated Platforms in High Performance FPGAs".
At first we took a set of applications in order to exploit multiple system-level architectures, and found the baseline architecture for our system. Then based on this architecture we developed the features we needed for virtualizing and using efficiently the FPGA resources. Like memory management units, scheduler of applications and scheduler of application's kernels. We support unique features like dynamic memory allocation, and dynamic memory free. We designed a efficient page table in terms of area and performance, which is the most critical component of our architecture. We also support partial reconfiguration which is a must for real-time systems, in order to be able to reconfigure the FPGA on the fly and change the application on real-time.
Reaching target performance does not have a trivial solution. Customizing the accelerator configuration while using a fixed system architecture is not enough to compete with state-of-the-art CPUs. It is essential to customize the system architecture to reach this goal, especially in applications where data movement overhead dominates overall performance. In this effort we studied few directions in system-level architectural exploration to orchestrate an approach for customizing system-level components, such as number and type of bus interfaces, memory hierarchies, and number of accelerators. We considered different data transfer protocols as a way to minimize data movement overhead. We intend to study other types of applications to extract more efficient ways of system customization. Last but not least, we intend to explore more complex architectures, smarter scheduling techniques and virtualization of multiple FPGAs.

Bibliography

1. P. Skrimponis, G. Zindros, I. Parnassos, M. Owaida, N. Bellas, P. Ienna. Exploring Automatically Generated Platforms in High Performance FPGAs. Parallel Computing with FPGAs (ParaFPGA). Edinburgh, Scotland, September 1-4, 2015

2. K. Vipin, S. Fahmy. DyRACT: A Partial Reconfiguration Enabled Accelerator and Test Platform. Field-programmable Logic and Applications (FPL). Munich, Germany, September 2 - 4, 2014

3. K. Vipin, S. Shreejith, D. Gunasekera, S. A. Fahmy, N. Kapre. System-Level FPGA Device Driver with High-Level Synthesis Support. International Conference on Field Programmable Technology (FPT), Kyoto, Japan, December 9-11, 2013

4. K. Fleming, H. J. Yang, M. Adler, J. Emer. The LEAP FPGA Operating System. 24th International Conference on Field Programmable Logic and Applications (FPL). Munich, Germany, September 2-4, 2014

5. H. J. Yang, K. Fleming, M. Adler, J. Emer. LEAP Shared Memories: Automating the Construction of FPGA Coherent Memories. 22nd International Symposium on Field Programmable Custom Computing Machines (FCCM). Boston, USA, May 11-13, 2014

6. Vivado Design Suite User Guide: High Level Synthesis. Online at www.xilinx.com

7. Altera OpenCL SDK Programming Guide. Online at www.altera.com

8. M. Jacobsen, R. Kastner. RIFFA 2.0: A reusable integration framework for FPGA accelerators. 23rd International Conference on Field programmable Logic and Applications (FPL). Porto, Portugal, September 2-4, 2013.

9. A. Agne, M. Happe, A. Keller, E. Lubbers, B. Plattner, M. Platzner, C. Plessl. ReconOS: An Operating System Approach for Reconfigurable Computing. Micro, IEEE (Volume:34 , Issue: 1)

10. C. Plessl, M. Platzner. Virtualization of Hardware – Introduction and Survey. European Regional Science Association(ERSA). Las Vegas, Nevada, USA, 2004