A user-transparent system for virtualizing reconfigurable hardware accelerators

A DISSERTATION PRESENTED
BY
CHARALAMPOS VATSOLAKIS
TO
THE ELECTRONIC COMPUTER ENGINEERING DEPARTMENT

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER IN SCIENCE
IN THE SUBJECT OF
COMPUTER ARCHITECTURE

TECHNICAL UNIVERSITY OF CRETE
CHANIA, CRETE

DECEMBER 2015

Thesis advisor: Dionysios Pnevmatikatos Charalampos Vatsolakis

A user-transparent system for virtualizing reconfigurable hardware accelerators

Abstract

Hardware based acceleration is highly efficient, but there are several factors limiting its adoption. The most notable of those, is the lack of a standardized system, capable of providing a transparent interface between software and reconfigurable hardware. The product of this work, is a system capable of loading acceletors and perform I/Os in a completely transparent to the user manner. The user is capable of implementing an accelerator compatible to the system by using a standard set of ports. The access to this accelerator is aided by a given software API.

The system is based on the PCI Express interface (version 1, 4 lanes) for data transactions and the ICAP for reconfiguration. There are three partially reconfigurable regions available, while the systems software is responsible for scheduling the accelerators waiting for execution. There are four scheduling policies implemented; noop, simple, out of order, and forced. The first two, take in account the submission order, while the others, target to reduce the number of reconfigurations.

The total throughput our system reached, equals 618 MB/s for transmissions, 544 MB/s for receptions and 488 MB/s for reconfigurations. The behaviour of our system under heavy load, was evaluated through an edge detection system consisting of four accelerators running sequentially. We sustained a real-time

throughput of 30 FPS in 720p HD datasets, even when the least efficient scheduling policy was selected.

Contents

Listing of figures

Acknowledgments

O give thanks to the LORD; for he is good: for his mercy endures for ever. (Psalm 135:1) First and foremost, I would like to thank God, for being here and attending the MSc program of the Technical University of Crete.

I would like to thank my family and Stella, for their continuous support, patience and encouragement.

I would like to express my sincere gratitude to my supervisor, Professor Dionysios Pnevmatikatos, for the continuous support of my study and research, for his patience, motivation and immense knowledge. I would like to thank Professor Apostolos Dollas and Assistant Professor Ioannis Papaefstathiou, accepting to evaluate my work. I would also like to thank Kyprianos Papademetriou for his guidance and all the fellow MHL colleagues for making the office, a very pleasant place to work.

Finally, I would like to acknowledge Sklavos, my desktop computer. Sklavos is the computer I used during the implementation of the system and its evaluation. Its capability of implementing multiple hardware designs in parallel, significantly reduced the total development time.

Research reported in this thesis was supported by INTERREG EVAGORAS and FP7 FASTER Projects.

People who are more than casually interested in computers should have at least some idea of what the underlying hardware is like. Otherwise the programs they write will be prety weird.

Knuth

1 Introduction

It has been almost 70 years since the first computer was built, but the need for computational efficiency is not yet covered. During the first 25 years, the performance was improved by a factor of 25% per year. In the late 70s the microprocessors were introduced, while the integrated circuit technology gave a greater rate of improvement. In the late 80s Reduced Instruction Set Computer (RISC) architectures were introduced, where the number of assembly instructions was reduced and the performance was aided by Instruction Level Parallelism (ILP) and caches. In order for Intel to exploit the high performance of RISC architectures, x86 internally implemented a RISC-like processor, to which its complex instructions were translated. During the 1990s the performance growth reached a rate of 50%, leading to the dominance of microprocessor based computers. [11]

Since the beginning of the 21st century, the performance growth has fallen to 20%, while the needs for performance were covered through increasing the number of processors per chip. There were several other technologies available to the wide public, since commercial processors became more powerful. Initially, Single Instruction Multiple Data (SIMD) instructions were added and later the Multiple Instruction Multiple Data (MIMD) technique was adopted.[11] The multi-core architectures were introduced to the mainstream computers due to the limitations of increasing the clock frequency in transistor size, power consumption and heat dissipation. Thus, the performance is not defined by ILP, but Thread-level Parallelism (TlP) and Data-Level Parallelism (DLP).[16]

Traditionally, the main processor was responsible for the execution of algorithms. There are several other platforms though competing for the title of the most efficient in terms of computational strength or power consumption. The following sections discuss the current state of computing systems and the capability of accelerating applications through General Purpose Graphics Processor Units (GPGPUs) and Field Programmable Gate Arrays (FPGAs).

1.1 Computing Systems

In this section we discuss the current state of computing systems in terms of computation capabilities, the interconnection of the processor with the main memory and the dominant Input / Output (I/O) interfaces. Accelerating applications in software requires several optimizations to be performed both in the code itself in terms of instructions and to the memory accesses in terms of locality. I/O interfaces are defining the performance of I/O intensive systems, since insufficient data bandwidth increases the overhead.

Initially, the main trend was to increase the number, speed and instructions of the processing elements. Towards this direction, Single Instruction Multiple Data (SIMD) provided a reliable acceleration option. MultiMedia eXtensions were introduced to mainstream computer systems through Intel in the late 90s. It included a set of eight 64 bit registers capable of performing packed 8, 16 or 32

bit operations [22]. Streaming SIMD Extensions (SSE), introducd in Pentium III processor, was capable of performing operations between four packed single precision floating point data elements, doubling the width of the existing MMX registers. In later versions of SSE, more operations and data types were supported. The latest approach of SIMD operations is Advanced Vector eXtensions (AVX), where the total amount and width of registers was doubled (16 x 256-bit wide registers). In the future versions of AVX, the size of the available registers will be increased to 512 or 1024 bits[17].

The potential performance increase by using the SSE instruction set can be up to 16x per core, for single byte operations, but in the most common case the expected speedup can be around 4x. The AVX instruction set is capable of doubling the performance compared to the SSE [17]. As a result, the maximum speedup we may gain in an 4-core chip using fully parallel algorithms, exploiting the AVX capabilities of the system can be up to 32x.

Accelerating real world applications through vectorization is not always that efficient. In [33], the total speedup gained through AVX optimizations, was insignificant compared to the SSE based. The main performance barrier is related to the memory accesses and the data locality.

The amount of on chip memory has been improved over the past decades. Mainstream processors contain a hierarchy of caches consisting of several levels (L2 or/and L3 are shared between cores), with L3 caches reaching 16MB in size. The main memory is based on DDR3 SDRAM DIMMs with a common operation frequency to be equal to 1600MHz, giving a throughput of 12.8 GB/s. The main memory was traditionally managed by the northbridge controller, which also interfaced high bandwidth internal components such as the southbridge and AGP or PCI Express based devices. In modern systems, the northbridge controller is integrated to the processor chip, reducing the communication latency.

The southbridge controller (recently known as platform controller hub) is interconnected to the main processor through a high bandwidth interface. Its main responsibility is to implement I/O interface with internal (SATA, IDE, I2C,

etc.) and external (PS/2, USB, RS232, etc.) peripherals.

1.1.1 PCI Express

PCI Express is a highly efficient serial interface. It is the successor of the parallel PCI/AGP interfaces and there is a number of lanes available per device. The device throughput is related to the number of lanes a device supports. In terms of bandwidth, the initial version of PCIe (v1.0) offered a maximum of 2.5 Giga Transfers per second (GT/s) per lane for each direction. Due to the 8b/10b encoding, the maximum achievable bandwidth per lane was 250 MB/s per direction. A link can be formed by multiple lanes and is denoted as x1, x4, x8 and x16[8].

The PCI Express consists of three discreete layers; the Transaction Layer, the Data Link Layer and the Physical Layer. The communication between components is packet based. The Transaction Layer is responsible for the assembly and disassembly of Transaction Layer Packets (TLPs). There are different types of packets supported depending on the type of transaction, since the PCI Express specification uses Message Space to support all prior sidebar signals such as interrupts. This layer applies a data protection code and a sequence number before submitting it to the Physical Layer. The Data Link Layer is responsible for link management and data integrity. In case of an error, a retransmission of TLPs is requested, until the information is correctly received. The Physical Layer is responsible for converting the information of the Data Link layer into a serialized format and transmitting it across the PCI Express Link. [21]

PCI Express 2.0 doubled the transfer rate to 5.0 GT/s, leading to a maximum throughput of 500 MB/s per lane for each direction. The encoding scheme is the same as the previous version, but there are some other features added such as dynamic link speed management [19]. The latest PCI Express specification (v3.0) doubles the payload to 1 GB/s per lane. This occurs by increasing the transfer rate to 8.0 GT/s and reducing the overhead through a more sophisticated 128b/130b encoding scheme [20]. Each PCIe version doubles the per lane

throughput of its previous, without necessarily doubling the clock frequency. Theoretically a PCI Express x16 device, would provide a maximum throughput per direction of 4GB/s in v1.0, 8GB/s in v2.0 and 16GB/s in v3.0.

The PCI Express interface provides a highly efficient interconnection between the processor and the internal devices. The high throughput provided, allows us to transfer data in real time. Thus it is possible to move the computation from the main processor to a device attached to it and still gain a respectable speedup. The GPGPUs and the FPGAs are types of devices that are able to perform computations efficiently and exploit the high bandwidth of PCI Express.

1.2 GPGPUs

Many-core architectures such as GPGPUs were able to deliver high computational throughput in computation intensive parallelizable applications. Their key feature is their ability to outperform CPUs in floating point operations and memory accesses. The total cost per gigaflop of GPGPUs compared to CPUs is lower, but the power consumption still remains an issue [34]. The difference between them and the CPUs is the amount of logic dedicated to data processing rather than flow control and caching. Each GPGPU contains a set of multiprocessors and each of these is capable of executing a set of threads organized in blocks and grids. Each thread block, executed in a multiprocessor, has access to its memory resources (register file and on-chip shared memory). [7]

Each application executed in the GPUs follows a Single Program Multiple Data programming paradigm, where computation parallelization is strictly defined during the implementation of each accelerator. The data transactions are performed between the main memory of the system and the device DRAM. Each DRAM access performed by a multiprocessor, needs to request a continuous data segment in order for the bus to be saturated. The shared memory per multiprocessor, needs to be managed manually and in a certain manner while its size is another limitation. Generally, implementing accelerators in GPGPUs can be a time consuming procedure and their efficiency is granted only on

parallellizable algorithms.

1.3 FPGAs

Accelerating applications in hardware is the most efficient option in accelerating applications. The flexibility of hardware allows the implementation of a wide variety of applications, thus, even non-parallelizable algorithms may reduce their execution time. There are two options in implementing applications in hardware, FPGAs and ASICs. FPGAs offer reconfiguration capabilities and a more flexible implementation procedure. ASICs on he other hand offer higher speeds and their production in greater quantities can lower their cost. The main difference in terms of performance is the fact that FPGAs require more resources in terms of transistors compared to ASICs and reach lower frequencies[6]. The key feature of FPGAs compared to other hardware solutions, is their ability to alter their behaviour through reconfiguration.

Modern FPGAs are capable of implementing a wide variety of hardware accelerators consisting of components such as DSPs, SRAMs and logic blocks. There is also available a wide variety of precompiled Intellectual Property (IP) cores available to the designer at a relatively low cost. The IP cores available can either implement a set of interfaces between the FPGA and the system resources (onboard peripherals, Input / Output buses), or provide a set of FPGA internal components, such as complex arithmetic operations and buses. These components are either hard core independent circuits inside the chip, or soft core circuits implemented into the reconfigurable logic of the FPGA.

There are multiple FPGA boards available on market offering a wide set of on-board resources. These resources are related both to memory (DRAMs, SRAMs and flash memory) and I/O (PCI-Express, ethernet, USB, HDMI etc). In high end boards [3, 26], there is available a set of QDR SRAMs and DDR3 DRAMS. The combination of the FPGA board resources and the available IP cores, gives us the ability to develop highly efficient hardware accelerators.

Accelerating applications in hardware using FPGAs is a similar procedure

compared to the GPU based acceleration in terms of I/O. The I/O is a common target of research and there are several interfaces available for the interconnection between the CPU and the FPGA. The time consuming part of an application can be executed as a hardware component while the input data need to be transferred to the FPGA and the results to be fetched from it. This procedure requires the FPGA to be loaded with the appropriate hardware component, at initialization time.

1.3.1 Runtime Reconfiguration

In a perfect world, the FPGA would contain the appropriate hardware component the time we need it, so that the system would be able to use it without any delay. In the real world, the reconfiguration needs to take place at runtime, adding a considerable overhead. The roots of this delay lay on the reconfiguration interface, the size of the data required to program the chip and the software responsible for the reconfiguration procedure.

There multiple options available for reconfiguring the FPGA chip, depending on the board. In certain boards there are multiple chips, one of which is responsible for programming the others. In the most common case, the FPGA is programmed through a narrow interface such as JTAG. The appropriate design is sent to the board through a software tool provided by the manufacturer. There is also the ability of keeping a set of bitstreams into the device ROM, attached to the chip. The interface used in this case is much more efficient, but the available memory space allows us to keep a limited set of designs.

The reconfiguration file, contains all the data and commands needed for programming the FPGA and its size depends on the logic available on chip[30]. Its form is strictly device dependent and its production is based on the CAD tools provided by the vendor.

The software involved into the reconfiguration procedure depends on the appropriate system. In several systems, the operating system is not aware of the existence of the FPGA, as the interconnection is based on low bandwidth serial

or parallel interfaces. Thus, there is no need of altering the systems software during the reconfiguration procedure, which occurs through vendor software. In other cases [10], where the interface is managed by a different hardware component (such as a secondary FPGA), the reconfiguration procedure is aided by custom user and kernel level software.

Several issues arise, when the FPGA being reconfigured contains the logic responsible for interfacing with the host. The operating system is may use the interface during the reconfiguration process, which leads to system instability. As a reuslt, before reprogramming the FPGA, we need to disable the driver and disable the port on which is connected. At the completion of the reconfiguration procedure, we need to rescan for new devices added, enable the appropriate device and insert the driver to the system.

The total overhead of the full reconfiguration procedure is prohibitive if performed at runtime. In [18], the reconfiguration time of the netFPGA and XUPv5 boards required several seconds (3.8 and 18.5) for its completion. The reconfiguration procedure was performed through PCI and JTAG respectively. The reconfiguration procedure of the netFPGA was performed through register I/O operations over PCI. The reconfiguration data were transmitted from a user space application to the SelectMAP interface of the FPGA, through a secondary chip (smaller FPGA) controlling the PCI. The reconfiguration in the XUPv5 board, is based on the JTAG interface provided through a vendor USB programmer.

The full reconfiguration of the FPGA comes with a great cost; a large set of data needs to travel through a narrow interface. In order to reduce that cost the data size needs to be reduced and the bandwidth of the available interface needs to be increased. Partial reconfiguration allows the designer to selectively program several regions of the FPGA. In that way, the logic implementing the interface can be placed into the static region of the FPGA and the applications can be loaded into the reconfigurable regions[9]. The bitstream responsible for reconfiguring the partially reconfigurable regions has smaller size compared to the full bitstream.

There are several reconfiguration ports available [30], one of which can be accessed by the static region of the FPGA. The Partially Reconfigurable Regions (PRRs) of the FPGA, can be programmed from the static region through the ICAP port. The ICAP port is documented to work at 100MHz and has 32-bit width, allowing a maximum bandwidth of 400MB/s. The high bandwidth of the ICAP combined with the small size of partial bitstreams (a few MBs), allows us to perform partial reconfiguration in real-time.

1.4 Goals and Contributions

The goal of this work is the implementation of the ideal runtime system for partial reconfiguration. A system easy to work upon, based on standard interfaces both in software and in hardware. The I/O interfaces used, interconnecting the main system memory with the FPGA, would be fast enough to support a wide set of I/O intensive streaming applications. The reconfiguration procedure, manually performed on several systems, would no longer be an issue. It would be performed transparently with no user interference, at high speeds. The selection of the next accelerator to be executed, would be performed by the appropriate scheduler. The number of software applications accessing would be infinite, while the PRRs of the FPGA would be fully saturated. Each software application accessing the system, would have no boundaries on the number of accelerators it submits for execution.

This work targets to the implementation of a system capable of managing the reconfiguration and execution of hardware accelerators through a high throughput interface. Our goal is to deliver an efficient reconfiguration mechanism and a high throughput I/O interface to users. We envision a low-cost heterogeneous environment able to support the execution of multiple accelerators loaded into reconfigurable hardware under the control of host software. Our contributions are:

- The study of a system in which the CPU triggers FPGA reconfiguration over PCI Express.

- The presentation of our micro-architectural choices to achieve high transfer rates for reconfiguration and data transactions.
- The implementation of a kernel based software layer for managing and scheduling the existing accelerators and I/O requests.
- The implementation of a system serving an unlimited number of software applications at a given time, through time based multiplexing.
- The development of a transparent process, i.e. without requiring user involvement, for loading and accessing hardware accelerators.
- The implementation and evaluation of a set of partial reconfiguration scheduling policies.
- The evaluation of our system through a real-time edge detection application.

The report is structured as follows: Chapter 2 contains the related work. Chapter 3 presents our desktop system combining hardware and software resources. First, we discuss the hardware architecture, and then we detail the software side. At the end of Chapter 4 we provide initial results from performance evaluation proving that our approach is efficient. Finally, Chapter 5 concludes the report.

1.5 Publications

K. Papadimitriou, C. Vatsolakis, D. Pnevmatikatos, "Invited paper: Acceleration of computationally-intensive kernels in the reconfigurable era," in Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2012 7th International Workshop.

C. Vatsolakis, K. Papadimitriou, D. Pnevmatikatos, "Enabling Dynamically Reconfigurable Technologies in Mid Range Computers Through PCI Express," in HiPEAC Workshop on Reconfigurable Computing (WRC), Vienna, 2014.

I grew up thinking that a research scientist was a natural thing to be.

Stephen Hawking

2 Related Work

Several works exist in the literature studying the interface with an FPGA and the accesses to existing accelerators from the user aspect. There are works that provide a brief evaluation of the throughput reached through PCI Express transactions between the host and the FPGA.

PCI Express interface was evaluated extensively by Ray Bittner, [5] used it for transferring data between the host and the DRAM memory located into the FPGA board. In that design the device operates as a DMA bus master, capable of performing DMA transactions to the main memory. There is also a set of interrupts issued on DMA completion. The initiation of each transfer is performed by the driver and requires a number of device registers to be written. During the transaction, the data need to be fetched from the device DRAM. The design was implemented in a small Virtex 5 at PCIe v1.0 x1 (single-lane) and the total throughput reached, including the device DRAM latency, was 11-15 MB/s.

The key bottleneck in that work was the high amount of interrupts (7680 per second) and the overhead added for their service. They also measured the performance of the DMA from the hardware aspect and they found it equal to 79.3 MB/s for read and 74.1 MB/s for write requests. The device in our case, supports multiple PCIe lanes, leading to higher bandwidth. We also target to providing a complete system capable of reprogramming the FPGA through PCIe.

A newer evaluation of the PCI Express interface by Ray Bittner, [4] performed a more extensive evaluation to the PCI express interface. In this work, the total number of requests was reduced through improved memory management and register accesses. There is a contiguous buffer of 1MB allocated into the host memory and there is a circular buffer placed into the FPGA holding the addresses of the pending requests. The usage of buffers allows the appropriate data to be prefetched from the DRAM before needed. The total throughput gained from this procedure was around 200MB/s per lane on read operations and the write operations are around 5% slower. The Gen 2.0 configurations yield the 2x performance increase that is expected versus the Gen 1.0 configurations.

Another work, closer to our research is Reusable Integration Framework For FPGA Accelerators (RIFFA) [15], in which is an open source framework for hardware accelerators. Their main contribution is the communication between the hardware accelerators and the user-layer software through PCI Express. Their system can manage multiple accelerators implemented in hardware at a given time. It combines a user layer library, a Linux Device Driver and a set of IP cores. The access to the FPGA from the user space is performed through virtual device files. There is interrupt and DMA support between the workstation and FPGA in both directions. The user access is aided by a high level API provided by a software library. The hardware accelerators are attached to the PLB bus and there is a bride between it and the PCI Express. The total throughput reached is 181 MB/s for upstream DMAs and 25 MB/s for downstream.

RIFFA 2.0 [14], is rewritten and supports a wider range of FPGAs and operating systems. The FPGA support was extended to Spartan 6, Virtex 6 and 7 Series, supporting PCI Express of Gen 1 and Gen 2. The hardware architecture

has replaced the PLB bus with channel multiplexor, so that the data are directly transmitted to each accelerator. There is also support for multiple FPGAs accessed simultaneously (up to 5) from different software threads. The second version of RIFFA has bindings for C/C++, Python and Java. Our work is significantly different since we currently support a smaller set of resource consuming hardware accelerators, scheduled and running at a given time. We also provide interconnection between the user software and the hardware, but our system is capable of partially reconfiguring each FPGA region at a low cost and in a transparent manner.

ReconOS [2], addresses the subject of software and hardware coexistence under the same operating system and libraries. Their approach is based on a custom version of the Linux operating system, supporting several of its mechanisms to be implemented in hardware. Those mechanisms are responsible for synchronization between threads. The system provides the user, an interface for implementing software threads and hardware accelerators, in a manner that both of them are capable of executing in parallel. The user on the other hand, needs to implement their synchronization mechanism in hardware, while a wrapper is provided. The key feature of this work is the ability of software to run in parallel with hardware, resolving synchronization issues. Our system on the other hand, can be executed on almost any desktop computer system running Linux. The data transfers in both systems are transparent to the user and there is a standard interface between software and hardware. As a final comment, it is proper to admit that this work resolves issues we do not address, but there is no information regarding accelerator scheduling.

In another work similar to ours [23], the virtualization of hardware accelerators occurs through the Single-Root I/O Virtualization feature of the PCI Express interface. Their goal, was a system capable of sharing a single FPGA among the host and several virtual machines. Each operating system, was accessing one of the available coprocessors loaded, as a different device into a single chip. The system was built upon the AXI bus and the driver used into physical and virtual operating systems was similar. In [24], they extended this

system by adding partial reconfiguration support. Each coprocessor was located into a PRR, while the partial reconfiguration was performed through a shared resource manager. Both of works, were based on PCI Express gen 2 x8, which supports 4x higher throughput compared to our setup. The reconfiguration procedure, was based on the 8-bit wide ICAP interface placed upon the AXI bus, leading to high reconfiguration overheads. Each of their reconfigurable areas are identified as discrete devices, while in our system, we support one device with multiple PRRs. Generally, they perform a brief presentation of a system based on the throughput it yields, while its behaviour upon real-world applications remains a mystery.

In a previous work, we presented two systems capable of reprogramming the FPGA depending on the user selection [18]. The kernels loaded into the FPGAs were able to communicate with the user application through register I/O. There was a comparison between two platforms on which our systems were implemented; the netFPGA and the XUPv5. The key difference between each platform is related to their primary interface used for communication. The netFPGA uses the standard PCI interface, whereas the XUPv5 supports PCIe gen 1 x1. Another significant difference is the fact that in netFPGA, a secondary FPGA controls the PCI itself, leading to reconfiguration over PCI. The reconfiguration throughput is relatively low since the transaction is performed through register I/O. In XUPv5, the reconfiguration is performed through JTAG giving even lower throughput. In the present work we are extending the second system, by enabling DMA operation over PCIe and providing partial reconfiguration capabilities through ICAP.

If you could utilize the resources of the end users' computers,
you could do things much more efficiently.

Niklas Zennstrom

3

Partially Reconfigurable System

Hardware based acceleration is the most efficient solution for the widest set of applications. This work aims to combine the computation capabilities of hardware systems with the flexibility of reconfigurable logic. In that manner, applications are able to overcome the limitations in functionality of hard-core hardware accelerators. Each application is capable of executing its time consuming components into application specific hardware, leading to a significant speedup.

The development and deployment of reconfigurable hardware accelerators is not a straight-forward procedure. The designer needs to implement both the hardware and the software components required. The hardware components consist of the application logic and several cores required for I/O purposes. Each system contains a subset of software modules that need to be developed. These include the operating system specific driver, required for low-level device access

and the algorithm itself. It is common to insert a user-level library between the algorithm and the driver for ease of access.

Each software application requires hardware acceleration, assumes the accelerator requested is ready for use. There are several issues arising, when software applications require access to hardware accelerators that are either in use by another process, or not even loaded. Thus the designer of each system needs to provide access and functionality controls. In reconfigurable computing, we need to support the case where a set of software applications requires access into a single FPGA chip, at the same time.

There are multiple ways of sharing the FPGA resources among software applications. Each system contains a set of accelerators loaded into the FPGA that software may access. It is common, several applications to require access on accelerators of same logic, while the number of those instantiated is limited. The applications then, need to wait for a certain accelerator to be released in order for them to use it. In that way, the FPGA resources might be underutilized, since it is impossible to predict at design time, the number and type of accelerators commonly needed.

Partial reconfiguration enables us to reprogram certain FPGA regions at runtime, thus we are capable of creating several instances of the same accelerator if needed. The system can load the appropriate hardware accelerators at runtime, increasing the utilization of the FPGA chip. The number and type of accelerators loaded are defined by the computational needs of the applications running. In that manner, the FPGA contains only the necessary set of accelerators, leading to the highest utilization rates.

The contribution of this work is the design and implementation of a system able to communicate and partially reconfigure the FPGA, in a manner transparent to the user. Several hardware accelerators can be developed, depending on the user needs, for a certain piece of software or a software suite. Each of the implemented software packages, may transfer the compute-intensive part of their code into a single or multiple reconfigurable areas of the FPGA. Each hardware component requested, can either be already loaded into the

FPGA, or provided by the user as a bitstream file.

The I/O throughput and the reconfiguration overhead are considered to be the most significant performance barriers in partially reconfigurable systems. In previous chapters, we presented some of the internal and external interfaces available for reconfiguration purposes. During the implementation of our system, we tested several platforms and interfaces. The following subsection presents the initial forms of our system and the control the user had over the reconfiguration procedure.

3.1 Early Approaches

There were several candidates for the implementation of our partially reconfigurable system, providing a set of software and hardware resources ready to be used. Our goal was to create a primitive system capable of reconfiguring an FPGA region upon request. The reconfiguration procedure was performed by the user level software via the driver. The driver was responsible for translating the software requests into register accesses. There were two platforms seleted for implementing the first form of our system; the netFPGA and the XUPv5.

3.1.1 netFPGA

The netFPGA is a PCI based platform, housing two FPGA chips, a Spartan-2 which is responsible for PCI transactions, and a Virtex-II Pro which is available to be programmed with the user application. The main Purpose of the netFPGA board, was the implementation of several network based applications. The Spartan-2 chip housing the Configurable PCI interface (CPCI), contains strictly PCI related hardware. The Virtex-II Pro chip on the other hand, is responsible for executing the network related configurable applications (CNET). Reconfiguration and I/O accesses to CNET, are aided by CPCI and a set of software packages provided by the community. The software packages contain a driver, a user level library, and an utility responsible for reconfiguring the CNET through the CPCI. [10] [1]

The initial form of our reconfigurable system, consisted of several hardware accelerators and a user level application. The hardware accelerators were loaded into the CNET chip, while the software application was responsible for loading and accessing them. The software was capable of identifying the design loaded into CNET, transferring data through register accesses, and then loading the next hardware design needed. The reconfiguration procedure, was performed through an external user level utility taken from the NetFPGA supporting material.

This board operates at 33MHz shared among the available FPGAs. The maximum theoretical throughput reached in PCI equals 133MB/s as the bus is 32-bits wide. The reconfiguration procedure is performed over PCI, through a narrow 8-bits wide SelectMap port connecting the CPCI with the CNET chip. There is an extra overhead added to the overall procedure, since there is no DMA transmission of the bitstream data, but performed through register writes. [18]

The low throughput of the PCI interface in conjunction with the high overhead of the reconfiguration procedure, led us to delve for a more decent platform.

3.1.2 XUPv5

The same system was implemented using the XUPv5 platform, attatched to the PCI Express interface. This platform consists of a single Virtex-5 FPGA, housing both the interface and the application. The interface is based on the first version of the PCI Express and the maximum theoretical throughput is equal to 250MB/s. The applications loaded into the FPGA, are similar to those built on the previous board. The goal of our system remains the same; the user software should be capable of selecting and loading hardware designs into the FPGA on demand.

The integration of our system on this board, required a set of issues to be resolved, since the interface was not placed on a separate chip. The reconfiguration procedure in this platform is performed through the JTAG interface, thus there is a sequence of states that needs to be performed during this procedure. Initially, we need to detach the device from the system by removing

the driver and disabling the appropriate PCI Express port. The FPGA is then reconfigured. Once the procedure is completed, the system needs to detect new devices attached to the PCI Express interfaces. The final step is the device enabling and the driver insertion to kernel. At this point, the device is accessible to the user level software.[18]

The procedure presented above is highly time consuming, since it would be desirable to avoid the overhead of board accesses and driver reloading. The solution for such issue would be the selective reconfiguration of the application related components of the FPGA, as long as the interface dedicated hardware remains intact. Partial reconfiguration enables us selectively reconfiguring FPGA regions. The interface specific components are placed into a static part of the FPGA, while the application specific are located into a single partially reconfigurable region. This leads to an extinguish of the driver related overhead, since there is no need for driver insertion and removal. The partial bitstream is of smaller size compared to the initial, thus the reconfiguration time is also reduced.

Another step for reducing the reconfiguration overhead is the adaption of a more sophisticated reconfiguration mechanism. The idea is to transmit the bitstream data via DMA over PCI Express. Once the data reach the board, they are fed to a 32-bit wide internal configuration port (ICAP). The reconfiguration time is exponentially reduced, due to the high ICAP maximum operating frequency of 100MHz.

The transition from the netFPGA to the XUPv5 platform, led to significant system updates mainly in hardware. The system interface and the reconfiguration mechanism became more efficient, while the system is mainly based on DMA for data exchange. The driver is primitive, providing simple register accesses, thus the overall system control and arbitration are still based on user level software. The reconfiguration procedure is also performed by the user, while the overall system functionality is similar to the netFPGA implementation. Generally, the system upgrade improved the performance of the previous system, but there are several issues, regarding driver functionality leading to user transparency, needed to be resolved.

3.2 Partially Reconfigurable System

In this subsection we present the final form of our implemented system. The main feature of this system is the capability of selectively scheduling hardware accelerators, depending on the reconfiguration policy selected. Each user can submit a hardware accelerator for execution in our system, by providing a set of bitstreams. During the execution of each accelerator, the user needs to provide it with the necessary dataset through a well defined API. The system is then responsible for reprogramming the FPGA reconfigurable region, transferring the appropriate data to it and fetching the results.

In the user aspect, each software application can be accompanied with hardware accelerators built for our system. The hardware interface of each, is well defined and there are very few restrictions during the implementation procedure. The accelerators are loaded and accessed through a simple software API interface. The reconfiguration procedure itself and the low-level I/O operations (such as reconfiguration arbitration, DMAs, register accesses and interrupt handling) are performed completely transparently to the user.

The execution of hardware accelerators in our system consists of several stages. Initially, we need to define which accelerator needs to be executed and provide the appropriate reconfiguration files. Once the reconfiguration bitstreams are loaded into the system, the accelerator is capable of execution. The data transactions between the user software and the accelerator, are performed through transmit and receive functions. Transmission is performed asynchronously, in contrast to data reception which is built as a strictly synchronous procedure. Finally, once the transactions are complete, we need to remove the accelerator from the system.

There were several decisions taken, during the implementation of our system, making it highly optimal. Its key feature is the fact that most of the library calls provided, are non-blocking. The reconfiguration procedure and data transmission require high amount of time for their completion, depending on the state of the system. Thus the algorithm can invest its time in CPU intensive

operations, instead of waiting. The data reception on the other hand, is based on blocking operations in order for the software to acquire the data produced.

The initiation of execution for each hardware component submitted, depends on several parameters. The selection of the next accelerator to be reprogrammed is taken through a scheduling policy. There are several scheduling policies supported in our system. These policies define the strategies followed in the case that multiple copies of the same accelerator need to be loaded. There is a wide range of strategies supported by our scheduler, both sequential and out of order. Sequential strategies reprogram the available PRRs with the next bitstream available in the queue. Out of order strategies, search the queue for accelerators that are already loaded before in order to avoid reprogramming each PRR. The subsection of Systems Software presents the scheduling policies and the inner functionalities of our system in more detail.

The presentation of our system is based on the description of its three main components; the hardware architecture, the kernel level software and the user level software. The hardware architecture illustrates the hardware components of our system and the interfaces used between the CPU and the reconfigurable logic. The kernel level software subsection presents the driver we built for our system, the operating system mechanisms implemented for transaction purposes and the scheduling of accelerators. Finally we present the user level software of our system, the API provided to the users and the translation of each function call to system calls during its completion.

3.2.1 Hardware Architecture

The hardware architecture of our system, as presented in Figure 3.2.1, consists of two discrete components; the desktop computer and the FPGA board. The FPGA board can be attached to any computer supporting the PCI Express interface, which is the only system requirement. The FPGA chip contains all the hardware components required for the system operation and can be divided in two main categories, the static region and the partially reconfigurable regions.

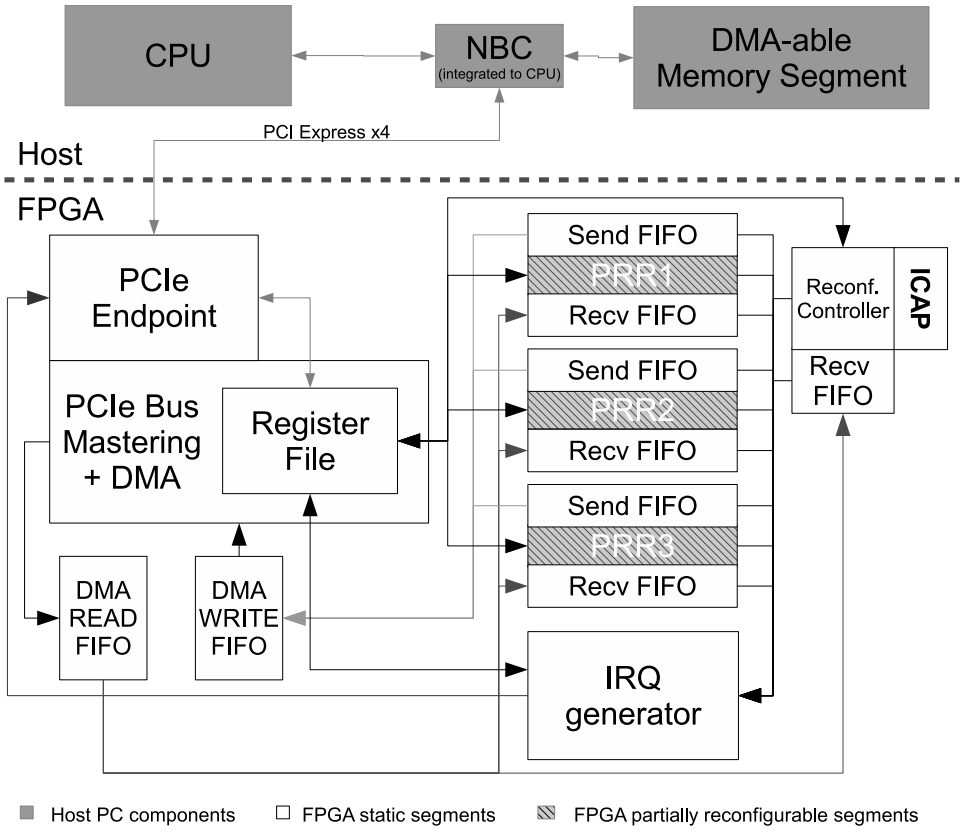

Figure 3.2.1: Hardware Architecture. The hardware components communicate with the host through a PCIe x4 interface. The host operating system initiates DMA transactions between the FPGA and the system memory. The data are transferred from the system memory, to the PRRs and the Reconfiguation Controller. The interrupt generator, informs the CPU regarding the state of the system through interrupts.

The static region holds all the necessary components for loading and accessing the accelerators. The partially reconfigurable regions on the other hand, are dedicated to the hardware accelerators executed.

Static Region

The static Region of our system consists of several components, responsible for reconfiguring and accessing the available PRRs. The interface to the host, is based on the PCI Express endpoint and its DMA mechanism, through which the data travel between the host memory and the FPGA. The data are then transferred through a set of buffers, used for temporary storage. The reconfiguration procedure is aided by a controller which drives the ICAP and provides the necessary data to it. The notification of the host regarding the state of the system is aided by the interrupt generator and the register file.

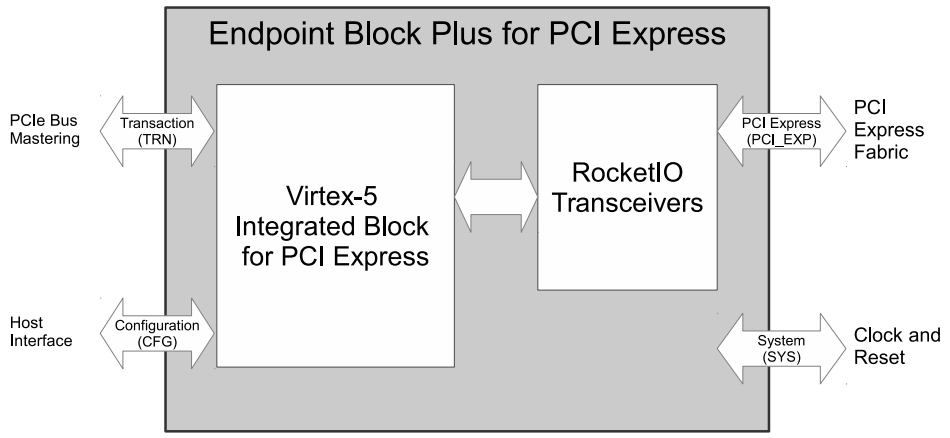

Figure 3.2.2: The Xilinx PCI Express endpoint implementing the Physical, Data Link and transaction layers. The endpoint takes as input Transaction Layer Packets and several control signals for configuration purposes.

The PCI Express interface in our system, consists of the Xilinx PCI Express Block plus endpoint core (presented in Figures 3.2.2 and 3.2.3) and the the DMA bus mastering component. The endpoint supported by our system is based on the first version of the PCI Express interface and is four lanes wide. It consists of the physical, the data link and the transaction layers provided as a hardware core [31]. The application layer provides TLPs to the endpoint, initiating transactions with the main system memory through DMAs. In our system, the application

was partially based on a hardware application implemented by Xilinx [25].

The PCIe interface itself is packet based, thus the data transferred are divided to several Transaction Layer Packets (TLPs). The data payload size transferred is architecture dependent, while a common selected value in DMA transactions is 128 bytes. Each layer adds a set of tags increasing the total size of each packet, leading to higher overhead for smaller packets. In DMA transfers, the user (in our case the driver) needs to set the payload size of each TLP and the total number of TLPs, at initialization time. In our case the throughput was measured to reach its maximum in DMAs of 256 TLPs carrying 128 bytes each (total size 32kB).

In conjunction with the DMAs where the TLPs are issued from the FPGA, in the Register I/O operations the host system issues a certain type of TLPs to the board. The Register File is a sub block located into the Bus Mastering component of the PCI Express interface. There are thirty five registers used by our system, twenty of which are used for DMA purposes. There is a set of fifteen registers used by our system; two dedicated to ICAP and its control, one to interrupts and messages, one for accelerator management and twelve located into the PRRs. There are four 32 bits wide registers, visible to the application for each PRR, three of which are general purpose. The fourth register is holding the ID number of the current bitstream loaded. The other three registers can be used for application specific parameterization and status purposes. Tables 3.2.1 and 3.2.2 present a deeper analysis of the resisters used for system management purposes.

There are two buffers used for temporarily keeping the incoming and outgoing DMA data. Our system works in a clock frequency of 125MHz and the buffers used are 64 bits wide, thus the maximum throughput supported equals the theoretical of the PCIe x4 (1 GB/s). The data are transferred between the DMA buffers and the buffers dedicated to the PRRs/ICAP, through a wider but slower bus (128bit wide operating at 62,5MHz). The data transfers among buffers are initiated during each DMA initiation. The initiation is performed by setting a value into the PRR Options register as presented in Table 3.2.2.

The partial reconfiguration of the FPGA is aided by the internal ICAP port, leading to real time capabilities. The necessary operations are performed by the ICAP controller implemented and the reconfiguration data are transferred through PCI Express. The partial reconfiguration of the FPGA requires the following set of steps to be performed:

1. Setup of the reconfiguration control register, as presented in Table 3.2.1.
2. Transmission of a 32KB bitstream block to the reconfiguration controller through DMA.
3. The controller feeds the ICAP with the appropriate data in a sequence of 32 bit words.
4. When the data are consumed, the reconfiguration control register is updated and a interrupt is issued to the host.
5. The steps 2 to 4 are repeated until all the reconfiguration data are transferred to the ICAP.
6. When the reconfiguration procedure is complete, the controller issues an interrupt.
7. The system checks the ICAP Status Register to acknowledge the integrity of the bitstream loaded.

The interrupt controller, is a minor subsystem responsible for issuing interrupts when an event occurs. The interrupt controller is responsible for updating the contents of the Interrupt Message Register presented in Table 3.2.1. This register contains the signals that cause interrupts, while it needs to be written at initialization time for enabling them. The following events cause interrupts to our system:

- A DMA transaction is completed.
- An accelerator consumed the given data block and requests the following.

- An accelerator produced a data block of results and the system needs to dispatch them.
- The execution of an accelerator is complete, another accelerator can be scheduled.
- The reconfiguration controller consumed its data block and requests the following.
- The reconfiguration procedure is complete.

The static logic contains the all necessary mechanisms in order to manage the available PRRs, which house the accelerators executed.

Partially Reconfigurable Regions

The reconfigurable space of our system consists of three equally sized PRRs. Each application is located into a PRR and has a set of dedicated memory segments and registers The total area allocated by the PRRs occupies the 70% of the FPGA in terms of slices, DSPs and routing resources. The accelerator itself is the second main component of our system. It resides into one of the three available PRRs, while our system can be extended to support up to seven PRRs.

Our system needs to support a wide set off accelerators, thus we need to provide a flexible interface to the hardware designers. The interface needs to be simple and robust, easing the development process. Thus each accelerator implements a set of simple ports for communicating with the system. These ports are presented in the Table 3.2.3. These ports include clock and reset signals, the unique accelerator ID, the general purpose registers available to the accelerator, the appropriate control and data signals for accessing the data buffers and the signal of the execution completed.

To address the case when multiple I/O intensive accelerators are executed, each designer needs to implement a stall state. In this state, the accelerator waits

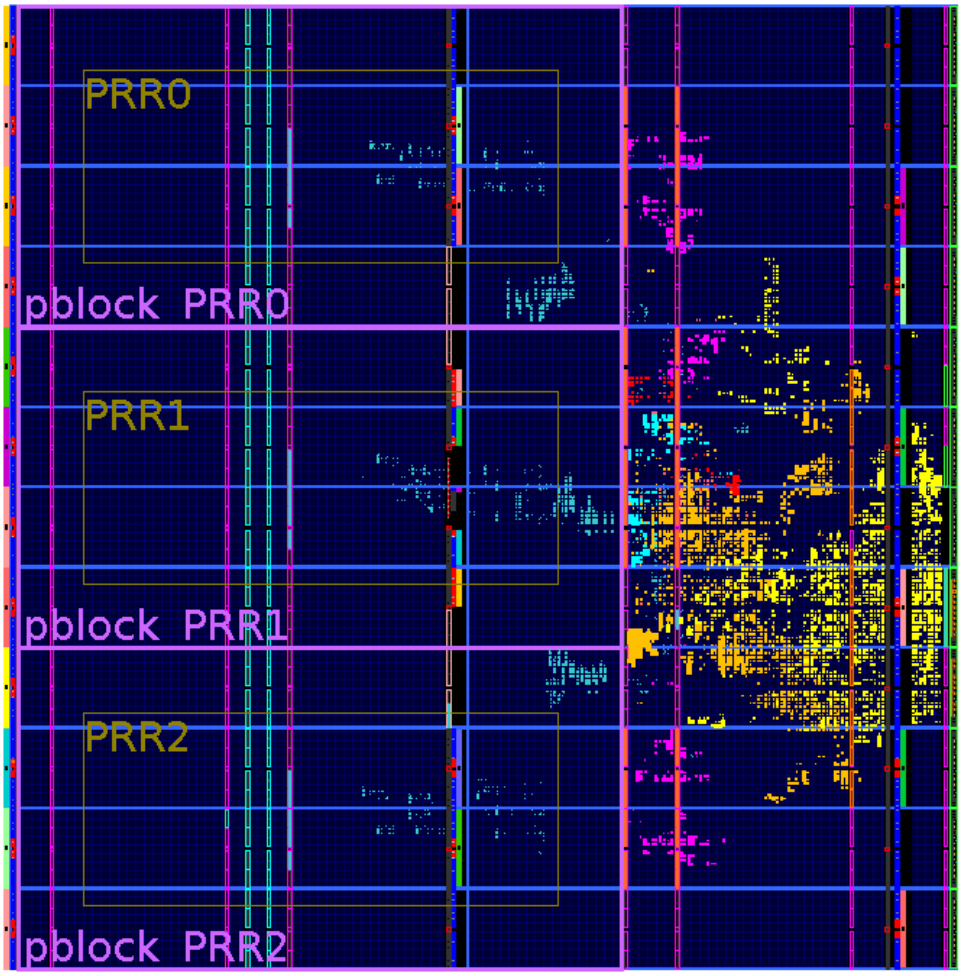

Figure 3.2.3: The hardware architecture as the components are placed in the FPGA. The PCIe endpoint and the bus mastering application are marked with yellow and orange dots. The ICAP Controller is marked with red dots and the PRRs are placed on the left side. The incoming and outgoing PRR buffers are attached to each PRR and are visible as orange rectangles. The accelerators loaded in this design are built for evaluation purposes, thus underutilizing the PRR resources.

for the data to arrive or the results to be sent. When these issues are resolved by our system, the accelerator can continue its execution. The designer also needs to provide to the system a signal regarding the completion of the accelerator

execution. In that way, the software will schedule another accelerator, depending on the policy selected.

Hardware Architecture Version 1.1

The system we built, based upon the first generation of the PCI Express may be considered depricated, thus we chose to rebuild our system upon a better platform. The resulting system (version 1.1) is based on the second generation of the PCI Express interface (four lanes wide). The PCI Express core and bus mastering application are upgraded, doubling the total theoretical throughput. This required the use of a faster clock, operating on 250 MHz. The increased clock frequency drives the PCI Express related components, while all the remaining components are operating at 125 MHz.

3.2.2 Software Architecture

The software components of our system, as presented in Figure 3.2.4, make it operational, functional and efficient. These consist of the user level library and the kernel level driver. Each application compatible to our system, consists of three copies of the same hardware design, one implemented per PRR. It is not known where each accelerator will be executed since the selection of the PRR that will be placed is performed at runtime.

The applications implemented, access their hardware accelerator(s) in a standard manner, through a user-space library. The library contains a set of simple function calls, translated to system calls and passed to the driver. The driver is capable of scheduling the FPGA reconfiguration, partially reprogramming the FPGA and communicating with it.

There is a standard path, the data blocks need to follow in order to reach the hardware accelerator. As presented in figure 3.2.5, the data transfers between the user and kernel software buffers are performed asynchronously before the data are requested. The system only needs to pay the cost of the data copy to and from

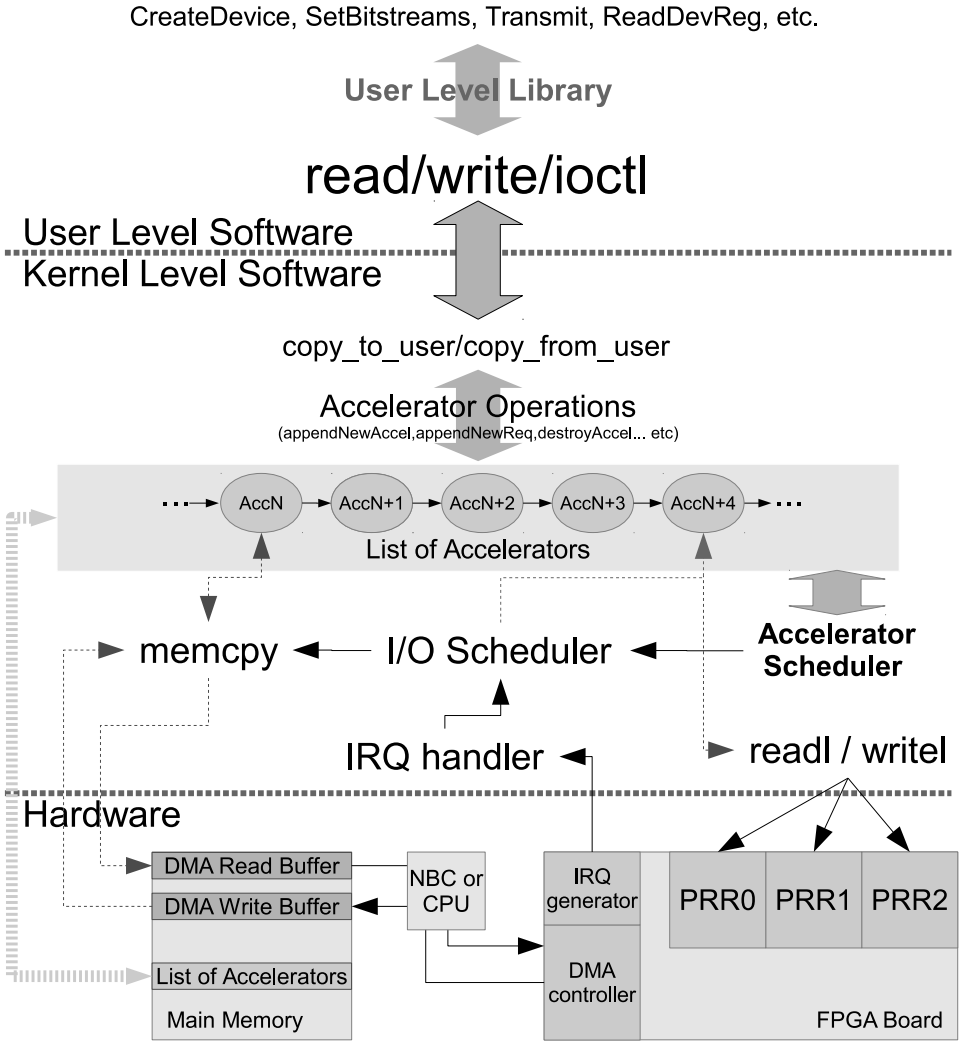

Figure 3.2.4: The software architecture of our system.
Initially, the user level software makes a library call translated to system calls. The driver then performs the appropriate operations to the list of accelerators. Once an interrupt is issued, the handler calls the I/O scheduler to dispatch the messages produced by the hardware. Depending on the message, the system may initiate a DMA or perform register I/O. The accelerator scheduling is policy based.

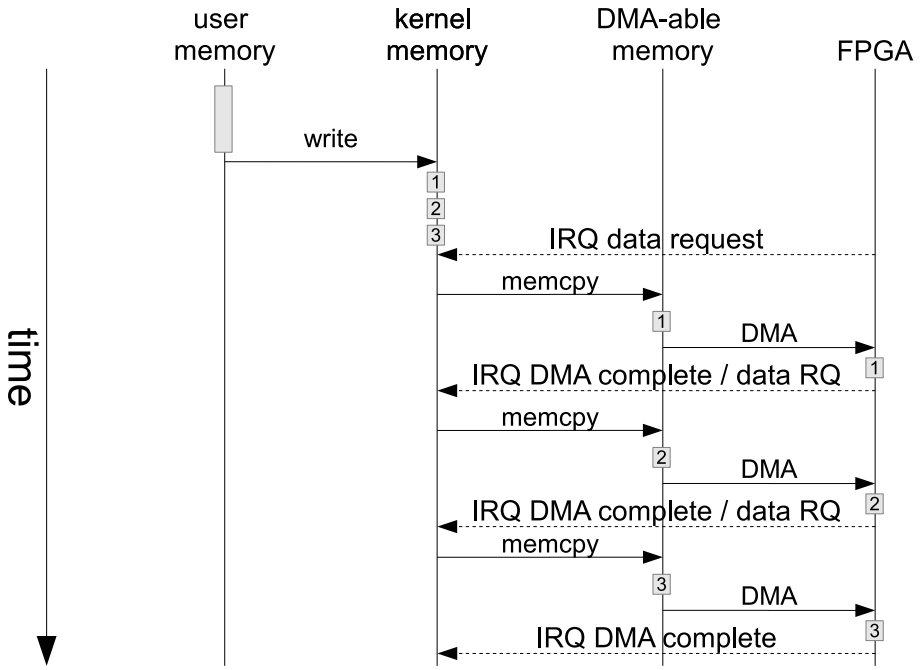

Figure 3.2.5: The operations performed during data transmission. The first step during data transmission is the data copy into the accelerator dedicated buffers in kernel space. The data buffers are segmented into 32KB blocks, as the DMA transaction size. Once the accelerator is ready to execute and request data, the system copies a block into the DMA-able memory segment and begins the transmission. The data reception is performed in an opposite manner.

the DMA-able buffers.

The virtualization of hardware accelerators is performed through the task scheduling. There are multiple applications waiting to be executed, containing several tasks each. Each task is submitted for execution in order, while multiple different applications may submit the same task. The system we built, accesses the tasks and chooses which of them will be executed in which PRR. The scheduling is policy dependent; in certain policies the tasks are scheduled in order, while in others the execution depends on the tasks that are already loaded. The following subsections present the two main software components of our system.

User Level Software

The main component of the user level software, is a library built for ease of access. The communication between the user level software and the driver is performed through this library. It purpose is to translate an easy and memorizable user level API, to complex system calls implemented by the Driver. Table 3.2.4, presents the prototypes of the functions provided to the user and defines an adaptive manner of communication between the software and the hardware.

Each software suite, adapted to our system needs to perform a set of steps in a certain order during its operation.

1. The user application creates a new accelerator instance.
2. It provides to it the appropriate bitstream files.
3. The application at this point, may need to write a set of registers (placed into the accelerator) for initialization purposes.
4. The user level software transmits to the accelerator the appropriate data sets and receives the results.
5. Once the execution of the accelerator is complete, its instance is deleted.

Our system implements non-blocking send and blocking receive functions. Each function is translated to a system call and informs the driver regarding the pending requests for the accelerator. When the software requires to write data to a hardware accelerator that is not yet loaded, the system may keep that data inside its structures and the software will not block. The data will be sent to the accelerator once it is loaded. Dispatching results on the other hand, requires a set of data to be calculated and fetched from the FPGA. As a result, the program needs to block until the system returns the results (which requires loading the accelerator and giving data to it).

The device driver, is accessed through a complex set of open, read, write and close system calls. Each function presented in Table 3.2.4, is capable of opening

the device, sending data to it and closing it in the end. The following subsection present the operations of the driver implemented in more detail.

Kernel Level Software

The purpose of our driver is to hold, schedule and serve the tasks submitted for execution. It is character device based and its accesses are performed through file operations. The device descriptor is located in the device folder and the user library can open it, write to it and then close it. The open and close system calls lead to the creation and destruction of new accelerator instances. Our system uses only the write system call for performing operations, through which requests are passed between the driver and the software.

Each request is transferred through a standard structure, holding information regarding its type and arguments. The request and its data are transferred from user to kernel space through the copy_from_user system call. It is then dispatched depending on its type and the driver performs the appropriate operations on the accelerator. The results are then written to the user application buffers if necessary (through copy_to_user system call), before the system access returns. There are the following types of requests supported by our system; set bitstreams, write device registers, read device registers and send/receive data.

The accelerator instances are kept into a list presented in Figure 3.2.6, until their execution is complete. Each instance, contains a set of data required for the reconfiguration and execution of every hardware accelerator. These include the bitstreams, the list of requests and the list of responses. The bitstreams are copied to the accelerator instance through the set bitstreams request. The write device registers, read device register and send data requests populate the list of requests. Each of these requests is added as an entry to that list sequentially, until data are requested by the software. Reading device registers and data blocks requires the user software to block until all the requests are dispatched by the accelerator.

The list of responses holds the data coming from the accelerator through DMA transactions. The data on this list are used in order for the system to serve the

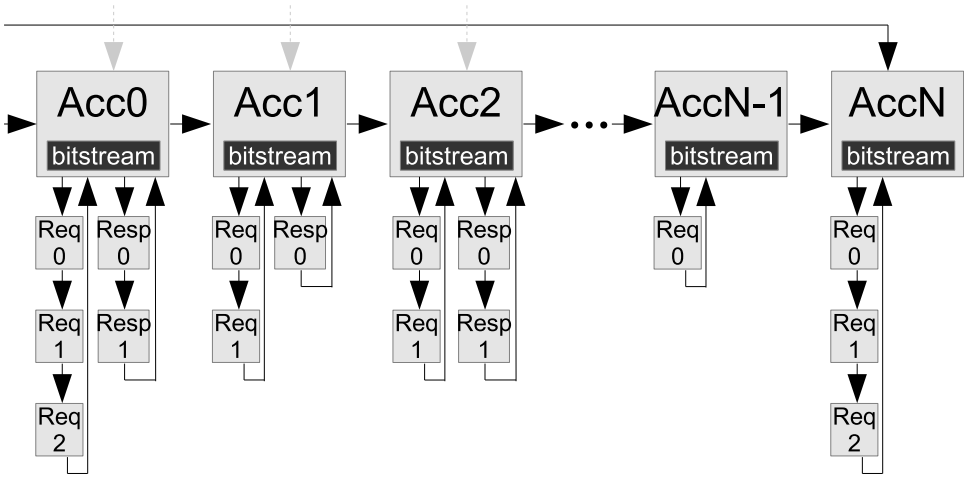

Figure 3.2.6: The list of accelerators as kept in Driver.
The accelerators are kept into a list until their execution is complete. Each accelerator holds its bitstream, a list of requests and a list of responses. Upon execution these requests will be dispatched in order, leading to DMA initiations or register accesses. The incoming data are placed into the list of responses.

"receive data" request. Each request issued, locks and seeks for data stored into that list. The data blocks found, are copied to the user level buffers. Only when the requested amount of data is sent back to the user, the software is unlocked and its execution to continues.

The list of accelerators, presented in Figure 3.2.6, is the most fundamental component of the driver, since it is the main data structure of our system. It holds all the necessary data for the configuration and execution of the accelerators submitted. This list is populated by the user level software calls and the data are consumed by the event handler of our system. The data consumption is linked to the incoming events from the hardware subsystem. The hardware collects one or more events and then issues an interrupt for informing the system.

PCIe supports both Message Signal Interrupts (MSI) and legacy interrupts. MSIs are issued by performing memory write transactions and their key feature is that the total amount of available interrupts is increased to 32. The legacy interrupt emulation is also performed by certain messages. Each legacy interrupt

pin (INTA, INTB, INTC and INTD) is represented by two certain messages (Assert and De-assert) sent to the system interrupt controller [31].

The interrupt handler dispatches the events written on the interrupt message register (table 3.2.1) and calls the I/O scheduler. The hardware components set the appropriate bit fields on the register and then an interrupt is triggered. The handler reads the value of this register and passes it as a parameter to the I/O scheduler function. The interrupt is acknowledged by resetting the messages dispatched through the register bit fields. Our system does not support nested interrupts, thus the hardware does not issue a new interrupt even though an event may occur during the interrupt service.

The I/O scheduler is responsible for translating incoming events, scheduling I/O operations and reprogramming the PRRs. The I/O operations performed per accelerator are related to the state of the PRR on which is loaded, and the incoming messages. The source of these messages can be either the accelerator scheduler, or the hardware, and certain types may lead to a state transition. There is a set of five states for each PRR, on which the I/O scheduling is based:

- **Starting State.** In the beginning of execution, all the available PRRs are in this state. They are not occupied by any accelerator and they are not accessed by user level software. They contain the last accelerator loaded and remain inactive until there is a software event issued. These events, generated by the accelerators scheduler will activate the PRR, either by loading a new accelerator, or by reusing the existing one.

- **Reconfiguration State.** When a reconfiguration software event is issued, the I/O scheduler sets the PRR in reconfiguration state. The reconfiguration of a PRR is set to be in the highest priority, thus the I/O scheduler exclusively transmits data to the reconfiguration controller. During this procedure, the requests for retrieving data from PRRs are served. The requests for transmitting data, on the other hand, are kept in a queue until the completion of the reconfiguration. The hardware then informs the system with an interrupt accompanied with the appropriate

message. The PRR is reset and switched to running state.

- **Reset and Reuse.** Certain scheduling policies (presented later in this section), allow the system to avoid the reconfiguration procedure. In that case, after the appropriate software event is issued, the PRR enters this state. In this state, all the necessary operations are performed for the PRR to become active and to be reset, before switching to running state.

- **Running State.** At this point, the PRR is loaded with the proper accelerator and its execution begins. The execution is strictly based on the data and configuration provided by the user, thus the requests regarding register I/O operations are immediately dispatched. The incoming hardware messages inform the system regarding the status of the buffers. The system then setups the DMAs required for sending/receiving data, and the execution continues until the "Execution Complete" hardware signal arrives. This signal leads to a transition to the dispatch data state.

- **Dispatch Data State.** The execution completion of an accelerator requires the acquisition of the remaining data. All the incoming messages from the PRR are ignored and once the expected data arrive or the application releases the accelerator, the PRR returns to the initial state (Starting State).

The state transition and the messages leading to it, are briefly presented in figure 3.2.7. The PRRs are the core of our system and the I/O scheduling is strictly based on their state. An accelerator is capable of issuing DMA requests even if its execution is complete. The system accepts the incoming requests only if the PRR holds an accelerator in running state. The DMAs are executed sequentially, while the highest priority is given to the DMAs targeting the ICAP. The I/O scheduling as a procedure, consists of five individual steps:

1. **Dispatch Messages.** Initially the system needs to process the incoming software and hardware messages. The software messages are generated by

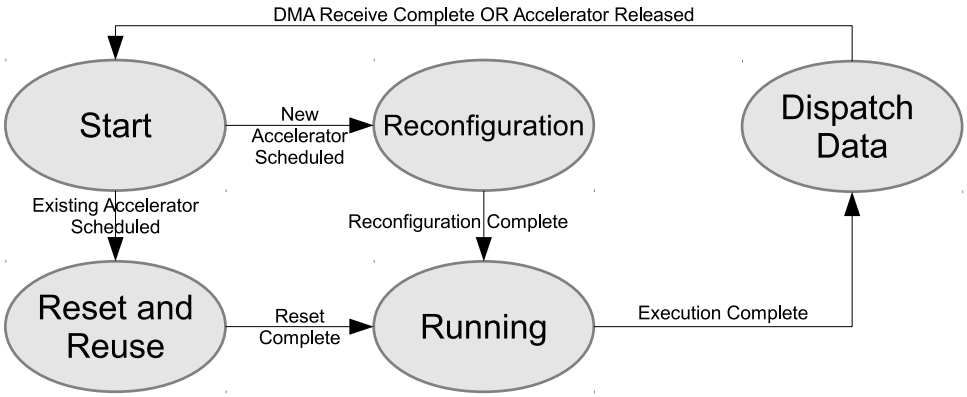

Figure 3.2.7: The state transition of a PRR.
Each PRR initially holds a random accelerator (start state). The scheduler issues a software event for either resetting the existing (reset and reuse state), or to program a new accelerator (reconfiguration state). The accelerator is executed (running state) until a completion event is issued. The execution is now complete and the system awaits for the last data packet to arrive (dispatch data state). The PRR is then returned to its starting state.

the accelerators scheduler, regarding the reconfiguration and reuse of a PRR, while the hardware messages are dispatched from the interrupt message register (table 3.2.1). The hardware messages are filtered depending on the state of the PRR they target, while the software messages are directly passed to the following step.

2. **State Transition.** At this point, the system dispatches the messages responsible for altering the state of a PRR. This is the phase where all the software and several hardware messages are fully dispatched. The state transition may lead to the initiation of several DMAs, performed in the following steps.

3. **Enqueue DMAs.** The messages regarding data transfers for active accelerators, are dispatched at this point. There are two queues available for keeping the pending requests, based on their type (one for sending and another for receiving data). Each message dispatched, leads to the addition

of a new request in the end of each queue. As mentioned earlier, there are operations performed in higher priority, such as the reconfiguration procedure, or the activation of an accelerator (switching from ”Reconfiguration” or ”Reset and Rerun” to ”Running” state). In those cases, there are several requests added in the beginning of the transmission queue. The requests are dispatched sequentially in the final phase.

4. **Perform PIO.** There is also the capability of transmitting words to the PRRs in the form of register I/O. Each accelerator contains a list of pending register reads and writes, provided by the software. In the most common case, the software writes the appropriate registers with the configurations provided, at initialization time. At this step, the system performs those register read/write requests sequentially.

5. **Initialize DMAs.** The final step of the I/O scheduler is responsible for initiating DMA transactions. The main requirement for this step, is the completion of the pending DMA operations. The DMAs are initiated based on the first elements kept in the appropriate data queues, which are then removed and consumed. Our system is capable of initiating full duplex DMAs, when both queues are populated.

The I/O scheduling procedure, as presented above, constitutes the key functionality of the interrupt handler. The implementation of our system is a slightly optimized version of the steps presented. In the real world scheduler, we had to reduce the execution time, by merging several operations into nested code blocks. As a result, the operations presented are executed sequentially, but each step is not restrained into its single code block.

There are two operations triggering the I/O scheduler, the hardware message generation, and the software message generation. The hardware message generation has been presented in previous subsections, while the software message generation is performed by the accelerator scheduler. There are two software messages produced, regarding the loading of new and the reuse of

existing accelerators. Each of these messages are strictly based on the accelerator scheduling policy selected.

Accelerator scheduling is a complex procedure, defining the next accelerator to be executed. The accelerators executed, are kept into the accelerators list (grey arrows in figure 3.2.6). There are three pointers, indicating the accelerators loaded into the PRRs of our system. Once an accelerator is released by the user level software, meaning the completion of its execution, the scheduler is called for selecting a new accelerator to be executed. The selection of the new accelerator to be executed, is based on the current scheduling policy. We implemented a set of scheduling policies; noop scheduling, simple scheduling, out of order scheduling and forced scheduling. Each of these policies are presented below, both in terms of code and in terms of functionality.

Noop Scheduling Policy

```
acc = firstPendingAccelerator();
loadToFreePRR(acc);
```

The noop scheduling is the simplest scheduling policy implemented. The accelerators are executed sequentially and in order. There are only reprogram messages generated by this policy, thus each PRR is always reprogrammed before use. There is no re-usage supported even though the required accelerator might already be loaded. As a result, the reconfiguration costs always burden the system leading to lower throughputs. Generally, this is a baseline scheduling policy targeting to define the minimum performance boundary.

Simple Scheduling Policy

```
acc = firstPendingAccelerator();
tmp = acceleratorOfFreePRR();
/* The accelerator is reset and used,
   not reprogrammed.*/
if(acc->type == tmp->type) resetFreePRR();
else loadToFreePRR(acc);
```

The implementation of the simple scheduling policy, is similar to the Noop Scheduling. The pending accelerators list, is accessed sequentially and each time, the first three accelerators of the list are executed (as presented in figure 3.2.6). The only difference to the previous policy, is the system check for the case where the accelerator loaded is the same with the one we wish to load. If a PRR is already programmed with the suitable accelerator, the accelerator is reset and no reconfiguration is performed. In that manner, the order of execution is strictly sequential, but the reconfiguration procedure is possible (but not probable) to be avoided.

Out of order Scheduling Policy

```
acc = firstPendingAccelerator();
tmp = acceleratorOfFreePRR();
/* We seek for an accelerator that is already
  loaded to the free PRR */
while(acc->type != tmp->type && acc->next != NULL)
        acc = acc->next;
if(acc->type == tmp->type) resetFreePRR();
else loadToFreePRR(firstPendingAccelerator());
```

The out of order scheduling policy, is a more sophisticated version of the simple scheduling policy. Their main difference is the fact that the accelerators kept on the list are not executed sequentially, but there is an out of order selection of the next accelerator to be executed. Once the execution is complete, the scheduler seeks the list for any existing accelerator of the same type to the one loaded. If such accelerator is found, the PRR is reset and the reconfiguration procedure is avoided. It is performed only if there is no pending accelerator same as the one loaded. The execution order is different to the submission order, while it is possible for several PRRs to hold the same accelerator if needed.

Forced Scheduling Policy

```
acc = firstPendingAccelerator();
tmp = acceleratorOfFreePRR();
while(acc->type != tmp->type && acc->next != NULL)
    acc = acc->next;
if(acc->type == tmp->type) resetFreePRR();
else {
    acc = firstPendingAccelerator();
    /* search for the accelerator that is not
       loaded to any of the PRRs */
    while( isRunningInPRRs(acc->type)
                              && acc->next != NULL )
        acc = acc->next;
    /* if unique accelerator is not found,
       load the first pending */
    if(isRunningInPRRs(acc->type))
        loadToFreePRR(firstPendingAccelerator());
    else
        loadToFreePRR(acc);
}
```

Forced scheduling policy extends the out of order policy, aiming to reduce even more the reconfigurations performed. The goal of this policy is selecting a different accelerator to be loaded in each PRR. The scheduler seeks for an accelerator that is either similar to the one loaded, or an accelerator that is not loaded into any of the available PRRs. In the first case, the accelerator is reset and reused, while in the second, a unique accelerator is loaded. Even though multiple instances of the same accelerator are avoided, they are preferred compared to inactivity. Thus, if the scheduler does not find neither the proper accelerator, nor a new accelerator, the first pending accelerator is loaded on the PRR. Generally, this policy is considered to be the most efficient, but in certain cases, the

reconfiguration overhead is not the only factor defining the system throughput.

Each one of the scheduling policies implemented has its own strengths and weaknesses. The first two policies follow the in order scheduling paradigm, which leads to fairness regarding the time each accelerator was submitted. Since our system supports multiple users, these policies ensure that the user that submitted his work first, will also be the first to be served. The other two policies on the other hand, are more performance oriented. The execution order is considered less important, usually bypassed if possible. Thus the performance is improved but the service time among users may be unequal. Each scheduling policy is further analyzed in the following section, dedicated to the evaluation of our system.

Kernel Level Software Version 1.1

Before discussing the evaluation of our system, it is proper to present a set of performance oriented upgrades performed during the transition to a newer platform. The upgrades performed increased the overall system stability and the total throughput gained. There were two strategies followed during the system upgrade; the reduction of both the critical sections in the schedulers and the total amount of memory copies performed.

Interrupt based systems, need to predict situations where nested interrupts may occur. In such cases, each interrupt can be serviced by a different processor, leading to the creation of multiple critical sections within the interrupt handling. Accesses to those segments require locking mechanisms, spinlocks in our case, increasing the overall system complexity and interrupt service overheads. The extinction of spinlocks, would require a mechanism that can guarantee exclusive sequential accesses on critical segments.

Event polling, allows a single handler to access the critical section at a given time. The event handler is called through a high resolution timer, producing software interrupts every 2us. Each event handler, dispatched the events produced and calls the existing handler (used upon interupts). There is no

possibility of collision between handlers, thus the locking mechanisms are removed. The 2us interval is long enough for new events to produce, while a DMA of 32KB in PCIe g2 x4 requires at least 16us for its completion.

There is a standard path the data need to follow until they reach the hardware. Initially, they are copied from the user to the Kernel level memory segments. The data are kept in memory, until the scheduler copies them to the DMA-able memory segment and the DMA is performed. The data transmitted, follow a similar path; the FPGA receives data packets on DMA-able buffers, copies them to the accelerator memory and then asynchronously transfers them back in user space. This version tends to reduce the number of data copies performed.

This extension enables DMAs to be performed by any kernel memory segment; all the memory buffers used, are allocated into the DMA-able memory segments of our system. In that manner, there is only a signle copy performed between user and Kernel space, before the hardware transaction is performed. The only limitation of this approach, is the total amount of memory allocated by the system, which must not exceed 4GB.

Bit(s)	RW/RO	Usage
Registers 0-19 Reserved for DMA operations		
Register 20 Reconfiguration Control Register		
19:0	RW	Bitstream Size (in 4-byte words).
23:20	Not Available.	
24	RW	Reprogram Start.
25	RO	Reconfiguration is in progress.
26	Not Available.	
27	RO	Reconfiguration is paused.
28	RO	Reconfiguration is complete.
31:29	Not Available.	
Register 21 ICAP Status Register		
31:0	RO	ICAP output.
Register 22 Interrupt Message Register		
0	RO	DMA Read Complete.
1	RO	DMA Write Complete.
3:2	Not Available.	
4	RO	PRR0 Requests Data.
5	RO	PRR1 Requests Data.
6	RO	PRR2 Requests Data.
7	Not Available.	
8	RO	PRR0 Ready to Transmit Results.
9	RO	PRR1 Ready to Transmit Results.
10	RO	PRR2 Ready to Transmit Results.
11	Not Available.	
12	RO	PRR0 Execution Complete.
13	RO	PRR1 Execution Complete.
14	RO	PRR2 Execution Complete.
15	Not Available.	
16	RO	Reconfiguation Procedure is Complete.
17	RO	ICAP FIFO is Empty.
30:18	Not Available.	
31	RW	Interrupts Enabled
Register 23-27 Not Available.		

Table 3.2.1: Register File (1/2)

Bit(s)	RW/RO	Usage
Register 28 PRR Options		
1:0	RW	DMA Receive Destination Buffer: 00 PRR0, 01 PRR1, 10 PRR2, 11 ICAP.
3:2	Not Available.	
5:4	RW	DMA Send Source Buffer: 00 PRR0, 01 PRR1, 10 PRR2.
7:6	Not Available.	
8	RW	DMA Receive copy to Buffer.
9	RW	DMA Send copy from Buffer.
27:10	Not Available.	
28	RW	PRR0 reset.
29	RW	PRR1 reset.
30	RW	PRR2 reset.
31	Not Available.	
Register 29 PRR0 Accelerator ID.		
31:0	RO	Unique ID for the accelerator loaded.
Registers 30 and 31 PRR0 Argument Registers.		
31:0	RW	General Purpose Registers available to the Application.
Register 32 PRR0 Result Register.		
31:0	RO	General Purpose Register available to the Application.
Register 33 PRR1 Accelerator ID.		
31:0	RO	Unique ID for the accelerator loaded.
Register 34 and 35 PRR1 Argument Registers.		
31:0	RW	General Purpose Registers available to the Application.
Register 36 PRR1 Result Register.		
31:0	RO	General Purpose Register available to the Application.
Register 37 PRR2 Accelerator ID.		
31:0	RO	Unique ID for the accelerator loaded.
Register 38 and 39 PRR2 Argument Registers.		
31:0	RW	General Purpose Registers available to the Application.
Register 40 PRR2 Result Register.		
31:0	RO	General Purpose Register available to the Application.

Table 3.2.2: Register File (2/2)

clk	input clock operating at 125 MHz
resetN	input active low reset signal
incomingData	input 64 bit incoming data
incomingRen	output read enable for incoming data
incomingEmpty	input incoming data are consumed
outgoingData	output 64 bit outgoing data
outgoingWen	output write enable for outgoing data
outgoingFull	input outgoing data not transmitted
reg1,2	input 32 bit general purpose registers
ressreg	output 32 bit general purpose register
complete	output execution complete
bitstreamID	output 32 bit identification for certain bitstream

Table 3.2.3: Accelerator Port Map

Accel* CreateDevice ();	Creates an accelerator entry. Returns reference to the accelerator or NULL in failure.
int SetBitsetreams(Accel*, char*, char*, char*);	Assigns the bitstream files for all PRRs. Returns non-zero value in case of an error.
int Transmit(Accel*, void*,int);	Non blocking transmission of data. Takes as parameters the pointer to data and their size.
int Receive(Accel*, void*,int);	Blocking reception of data. Takes as parameters the pointer to data and the size of the data expected.
int ReadDevReg(Accel*);	Blocking read of a certain device register.
void WriteDevReg#(Accel*,int);	Non-blocking write to a certain device register. It can be either WriteDevReg1 or WriteDevReg2.
void DeleteDevice(Accel*);	Destroys an accelerator entry.

Table 3.2.4: Software Prototypes

Let the future tell the truth, and evaluate each one according to his work and accomplishments. The present is theirs; the future, for which I have really worked, is mine.

Nikola Tesla

4 Evaluation

This chapter is dedicated to the evaluation of our system in terms of throughput and performance. The throughput measurements are taken both regarding the data transfers and the reconfiguration procedure. The goal is to evaluate a set of different platforms and interfaces involved, as our system evolves. The performance evaluation is aided by a complex edge detection system. This system consists of four accelerators, presented in the appropriate section. In that manner, we tend to evaluate not only the behaviour of our system, but its scheduling capabilities in real world applications.

4.1 Throughput Based Evaluation

The subject of this section is the presentation of the total throughput reached, during the implementation of our system. The main goal is the comparison of

reconfiguration and transfer overhead amongst various platforms used, and the creation of a better understanding for the decisions taken at design time. There are two computer systems used during our measurements, presented on table 4.1.1.

System	**G965 Based**	**X58 Based**
Processor	Pentium D @2.8GHz	Core i7 950 @3.0GHz
Number of cores	2	4
Chipset	Intel G965 Express	Intel X58 Express
PCIe Support	Version 1.1 [12]	Version 2.0 [13]
RAM	2GB DDR2 @ 667	6GB DDR3 @ 1600
Operating System	CentOs 6.4 @ 32bit	CentOs 6.4 @ 64bit
KERNEL Version	2.6.32	

Table 4.1.1: Desktop System Specifications

The G965 based desktop was used during the early implementations of our system, based on the netFPGA and XUPv5 platforms. The X58 based desktop is mainly used during the development of the final version of our system (LX330T FGPA). There are also several measurements performed, regarding the total reconfiguration and data throughput reached by the XUPv5 platform. The results of those measurements are presented later in this section, while the specifications of each FPGA board are presented on table 4.1.2.

The amount of reconfigurable resources provided to hardware designers, in all of our system versions, is high enough for implementing non-trivial accelerators. Despite the high amount of resources available, the throughput evaluation was aided by a set of trivial accelerators, performing simple arithmetic operations among device registers. The throughput based evaluation is divided in two main subsections, dedicated to data and reconfiguration throughput respectively.

	netFPGA	**XUPv5**	**LX330T**
FPGA	Virtex-II Pro 50	Virtex 5 LX110T	Virtex 5 LX330T
Clock Frequency(MHz)	33.3	62.5	125
Data Interface	PCI 32bit @ 33MHz	PCI Express v1 x1	PCI Express v1 x4
Reconfiguration Port	8-bit SelectMap	JTAG & ICAP 32bit	ICAP 32bit
FPGA Resources [29], [28], [27], [30]			
Slices	23616[1]	17280[2]	51840[2]
Max Distributed RAM (Kb)	738	1120	3420
Max Block RAM (Kb)	4176	5328	11664
Full Bitstream Size (MB)	2,27	3,71	9,86
Partial Reconfigurable Regions			
Number of PRRs (size)	1 (100% of FPGA)	1(50% of FPGA)	3(23% of FPGA each)
Slices	23616[1]	8640[2]	11520[2]
Max Distributed RAM (Kb)	738	560	780
Max Block RAM (Kb)	4176	2736	1728
Partial Bitstream Size (MB)	2,27	1,73	1,95

[1] Virtex 2 Slice = 2 x (4-input LUT + Flip Flop)
[2] Virtex 5 Slice = 4 x (5-input LUT + Flip Flop)

Table 4.1.2: FPGA Board Specifications
The clock is doubled on each implementation compared to its previous one. The total amount of LUTs of each PRR is increased, while the amount of distributed RAM is almost kept the same, but the available block RAM size is decreased.

4.1.1 Data Throughput Evaluation

This subsection is dedicated to a brief presentation of the total data throughput reached, during the evolution of our system. Its purpose, is the creation of a better understanding regarding the maximum reachable throughput. Initially, we had to perform a baseline measurement regarding the lowest reachable data throughput, using the netFPGA platform. The following version of our system targets to evaluate the maximum throughput reached, through increasing the DMA transaction size. This leads to the selection of the appropriate DMA data size for our system, where the PCIe interface is fully saturated. Finally, based on the previous selection, we present the throughput reached by the latest version of

System	netFPGA Based	XUPv5 Based
Register Read (KB/s)	755	702
Register Write (KB/s)	889	1051

Table 4.1.3: Register I/O Measurements.
The measurements were performed on the G965 based system, comparing the throughput between the initial versions of our system. Their most notable difference is on the Register Writes, where the XUPv5 based system is 15% more efficient.

our system.

All the measurements presented in this subsection, regard the total real throughput reached in the user aspect, including user and systems software overheads. Each result, was calculated as a mean value of ten sub-measurements, consisted of either 100 register I/O operations (register throughput), or data transfers of 10GB (DMA throughput). The overall measurement procedure was aided by real-time timers, calculating the total time required for all of the transactions to complete.

The total data rate reached by our system through register data transfers, is presented on Table 4.1.3. The bandwidth of register accesses is extremely low, compared to the theoretical, even on the PCIe based system. Higher data rates can only be achieved through DMA, where the board is the bus master. The results of those measurements are presented on 4.1.1 and 4.1.2.

There are several facts, verified by the measurements of figures 4.1.1 and 4.1.2. First of all, that newer chipsets yield better performance; the X58 chipset is backwards compatible to PCI Express v1.0, but still its performance is higher (Table 4.1.1). Secondly, the minimum amount of data transmitted over PCIe, is a single TLP and its size is standard (on figure 4.1.1, doubling the TLP size, doubles the throughput as the latency is stable). Finally, as presented on figure 4.1.2, the interface seems to saturate for requests of size greater than 16KB (128

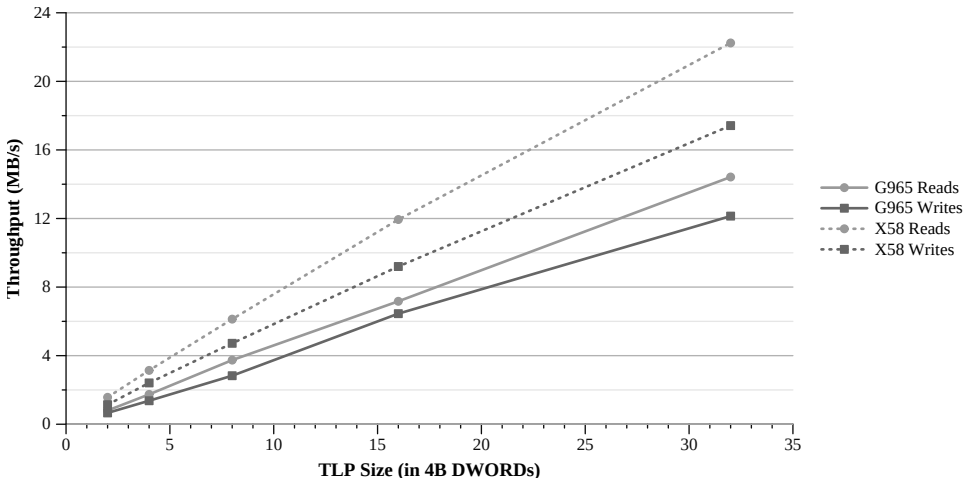

Figure 4.1.1: Throughput of DMA transactions on XUPv5 (single TLP). The read and the write throughout can be considered equal for the G965 based system, while the X58 based system doubles the Read throughput and gets 55% better write throughput.

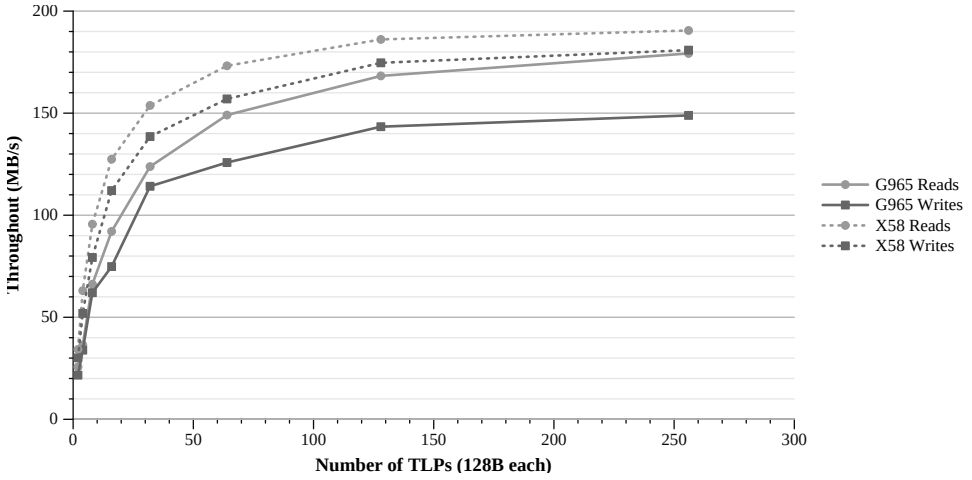

Figure 4.1.2: Throughput of DMA transactions on XUPv5 (multiple TLPs). The curves of the two systems increase in a similar manner and the total throughput reached, equals 149 MB/s (Writes) and 179 MB/s (Reads) for the G965 based system, while for the X58 based system, the results are 181 MB/s (Writes) and 191MB/s (Reads).

System	XUPv5 Based	LX330T Based
PCIe interface	Version 1 x1	Version 1 x4
Maximum Reachable	200 MB/s	800 MB/s
DMA Read (System Transmit)	191 MB/s (96% of max)	618 MB/s (77% of max)
DMA Write (System Receive)	181 MB/s (91% of max)	544 MB/s (68% of max)

Table 4.1.4: DMA Throughput Summary.
Increasing the number of lanes by 4x, gives us a DMA read and write gain of 3,2x and 3x respectively.

TLPs of 128 bytes).

During the evolution of our system, we had to balance the need for a decent throughput with the amount of resources available. As a result, we considered the most efficient DMA request size to be 32KB. Each DMA is temporarily stored into a Block RAM segment of 32KB and then it is transmitted to its destination. This size remains the same, even in the final version of our system, implemented over a 4-lane wide interface. The total throughput reached there, equals 618 MB/s for DMA Reads (system writes) and 544 MB/s for DMA Writes (system reads). It also supports full duplex transactions, which yield a combined throughput of around 1GB/s.

As mentioned earlier, the maximum throughput reached for the PCIe interface, is equal to 2.5 GTs per lane. Due to the 10b/8b encoding scheme, it is translated to 250 MB/s per lane. The throughput reached may be considered significantly lower than the theoretical maximum of 250 MB/s per lane. That is the rate of byte transfers performed in the physical layer, while there are several bytes added to each TLP among the transaction, the data link and the physical layer of the PCIe. The total amount of extra bytes added, varies between 20-28 depending on the system and device settings, in that way an overhead of 15-21% is added to 128 B TLP transactions [21]. As a result the maximum reachable throughput, taking into account the packet overheads, is around 200 MB/s per lane (in case there is no packet loss and no acknowledgement costs).

Table 4.1.4 summarizes the total throughput reached by our system. Generally,

System	netFPGA Based	XUPv5 Based		LX330T Based
Reconfiguration Port	Select Map over PCI	JTAG	ICAP over PCIe x1	ICAP over PCIe x4
Port Width	8-bit	1-bit	32-bit	32-bit
Port Frequency	33,3 MHz	6 MHz	62,5 MHz	125 MHz (OC)
Port Maximum Throughput	33,3 MB/s	0,75MB/s	250 MB/s	500 MB/s
Throughput Reached	0,6 MB/s	0,2 MB/s	82,4 MB/s	488 MB/s
Port Utilization	2 %	27 %	33 %	98%

Table 4.1.5: Reconfiguration Throughput Summary
The last two columns, based on the ICAP interface, are capable of performing reconfigurations in realtime.

the total increase in lanes did not give us a linear increase in throughput, due to possible packet loss and arbitration costs. The total throughput reached though, is high enough for executing real-time applications, consisting of multiple accelerators.

4.1.2 Reconfiguration Throughput Evaluation

This subsection is dedicated to the reconfiguration interfaces used during the system evolution and the throughput they yield. Our goal, is to perform partial reconfigurations in real-time, by increasing the total reconfiguration throughput. There was a set of 20 measurements taken, during the calculation of the total reconfiguration throughput reached by our system, amongst its various versions. The results are briefly presented on Table 4.1.5.

As table 4.1.5 suggests, during the evolution of our system, the total throughput gain was higher when internal reconfiguration ports were used. This strategy requires the FPGA to be already programmed with the initial design, which contains the ICAP controller. The design is located into the device ROM memory and the initial configuration is performed during the system startup.

The reconfiguration data targeting the PRRs, are sent through DMA, reaching the highest throughput possible. The main bottleneck in this procedure, is the width of the reconfiguration controller; the incoming data packets are 64-bit

wide, while the ICAP is half the size (32-bit). In the final version of our system (LX330T based), we managed to resolve this issue through double buffering, leading to even higher utilization rates. In that case, we use partial bitstreams of 1,95 MB (Table 4.1.2), leading to a rate of 250 partial reconfigurations per second.

The following section is dediacted to the combination of data and reconfiguration throughput reached, through a real-world application. We present an edge detection application built over our system, the total data transferred between our system and the board, and the total frame rate reached.

4.2 Edge Detection System

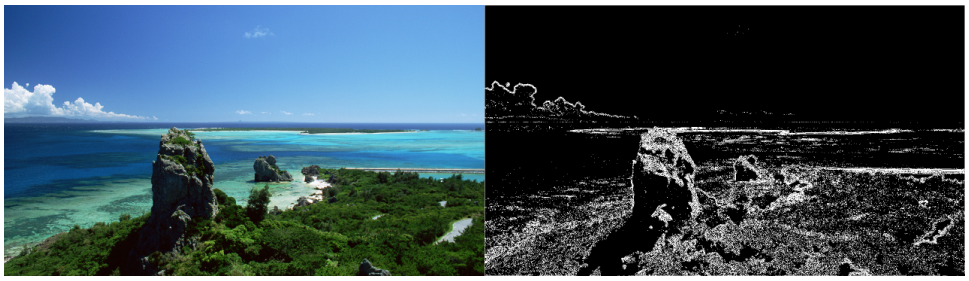

Figure 4.2.1: The Edge Detection System.
The input (left) and the output (right) of the edge detection application.

There is an edge detection application, built for evaluation purposes in the final version of our system. The application, consists of four discrete phases; the Greyscale conversion, the Gaussian Blur and Edge Laplace transformations, and the Threshold phase. Each of these phases need to execute sequentially, taking as input the results of the phases prior to them. The input is a bitmap based image in the RGB color space, while the output is a black and white image with all the edges marked (as presented on figure 4.2.1).

The edge detection application implemented in hardware, consists of four accelerators, one per each algorithmic phase. The hardware accelerators, need to

be loaded and executed sequentially in order for the system to produce the desired results. Every image produced, needs to pass through the user level software and then transmitted back to the following accelerator when needed. Thus during the execution for our system, the user transfers a total of eight images, four images transmitted and four received (only the first transmission is in RGB while all the other transactions are in Greyscale color space).

Before moving to the deeper presentation of each phase in hardware, it is proper to state, several functions that are similar among implementations. These functions are related to the architecture of our system, as presented on subsection 3.2.1 ("Hardware Architecture").

First of all, the accelerators are clocked at 125 MHz, which is the base system clock. Regarding the input and output data ports, their width is 64-bit, and the pixel placement on memory is little endian based. All the hardware accelerators support a waiting state when the input buffer is empty or the output buffer is full. Finally the completion signal is activated, when the counter of the pixels processed, reaches the size of the image in pixels (the image width and height are written on the registers at initialization time). The following subsections present the implementation of each algorithmic phase.

4.2.1 Greyscale

The image conversion to Greyscale, is generally a simple procedure; in the most common case, it is the mid value of all color channels of a pixel. Since each pixel in RGB consists of one byte per channel, the software requires three 8-bit additions and one division to be performed per pixel.

The calculation of each pixel value in hardware, consists of two additions and one division operation. The additions are performed in one clock cycle (the clock of 125 MHz is low enough to support nested adders). The division by three, is aided by a Block RAM containing all the appropriate values calculated; dividing a 10-bit value by 3, is equal to accessing a memory of 1024 x 8, holding the results of the division. The hardware implementation of the accelerator

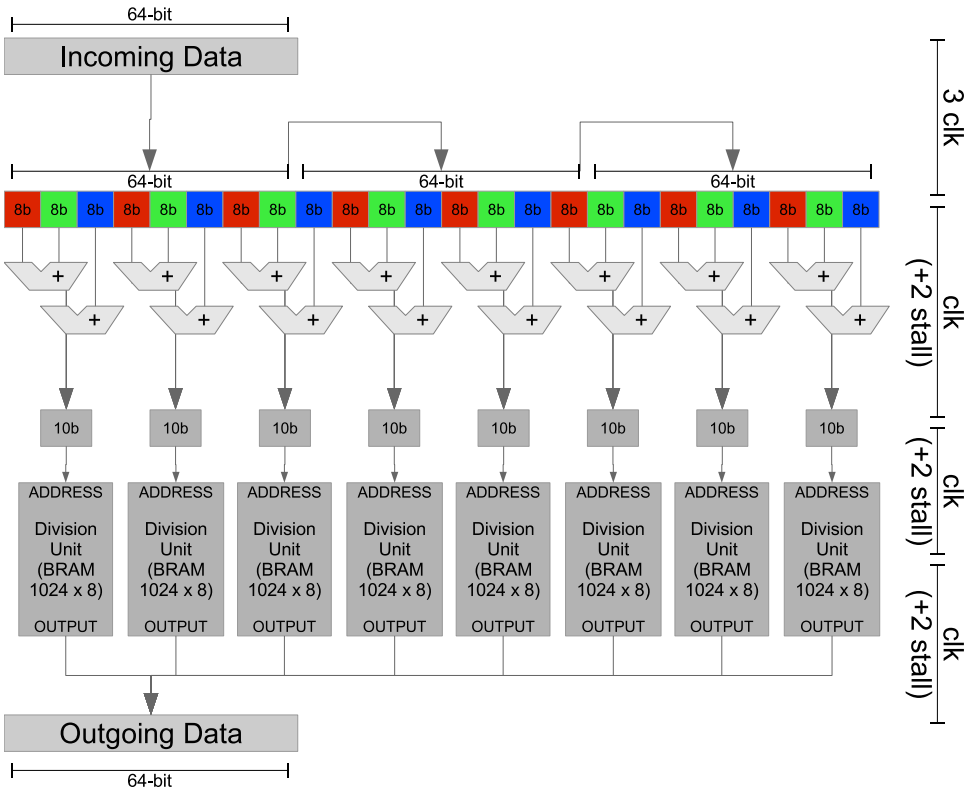

Figure 4.2.2: Hardware Architecture of the Greyscale Accelerator. The hardware is pipelined and produces a result, once every three clock cycles (the boxes in the figure represent registers and memories). It takes 12 clock cycles in order for the first result to reach the output.

performing this step, is presented on figure 4.2.2.

In the ideal case where there is a continuous data flow, the accelerator produces a result every three clock cycles; it requires 192-bit data (dispatched from input port as 64-bit words) for producing 64-bit results. The maximum theoretical throughput in this case, is 333 MPixels/sec.

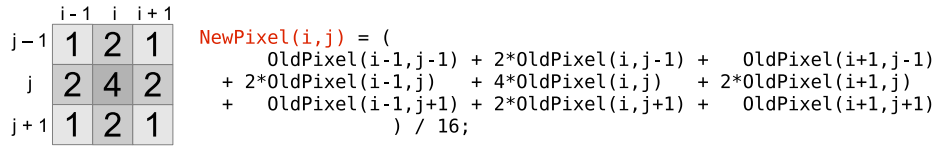

Figure 4.2.3: Algorithm used for Gaussian Blur transformation.

4.2.2 Gaussian Blur

The Gaussian Blur image transform, on its simplest form, adds information to a pixel based on its neighbour pixels; the current pixel value is equal to the old pixel value plus the pixel values surrounding it (each value is multiplied by the proper weight). Figure 4.2.3 presents the algorithm and the weight used per pixel, during this step. Implementing the Gaussian Blur in software, we slightly optimized the procedure by replacing the multiplications/divisions with bit shifting operations.

The hardware implementation of this step, had two main challenges we had to face; the memory arrangement of the incoming data and the computation of the results. The image pixels are transmitted sequentially, from left to right and top to bottom, while they reach the hardware as words of 8 pixels wide (64-bit data port). Our goal was to implement a fully pipelined accelerator, capable of producing 8 pixels each cycle.

As figure 4.2.3 suggests the computation of a single image line, requires three lines of data to be processed. In our design, the incoming data are the pixels of the "j+1" line, while the "j" and "j-1" lines are kept into FIFOs. Each 8-pixel word produced, requires the processing of 10 pixels wide lines. Our design supports 8-bytes wide data, as a result, we store the two LS bytes of the previous row and use them as the two MS bytes of the following. The data memories used and their connection, are presented on figure 4.2.4.

Once the data are available for processing (three lines of ten pixels every cycle), the system performs the appropriate calculations. Each pixel value is calculated by its computational block, thus our system contains eight block

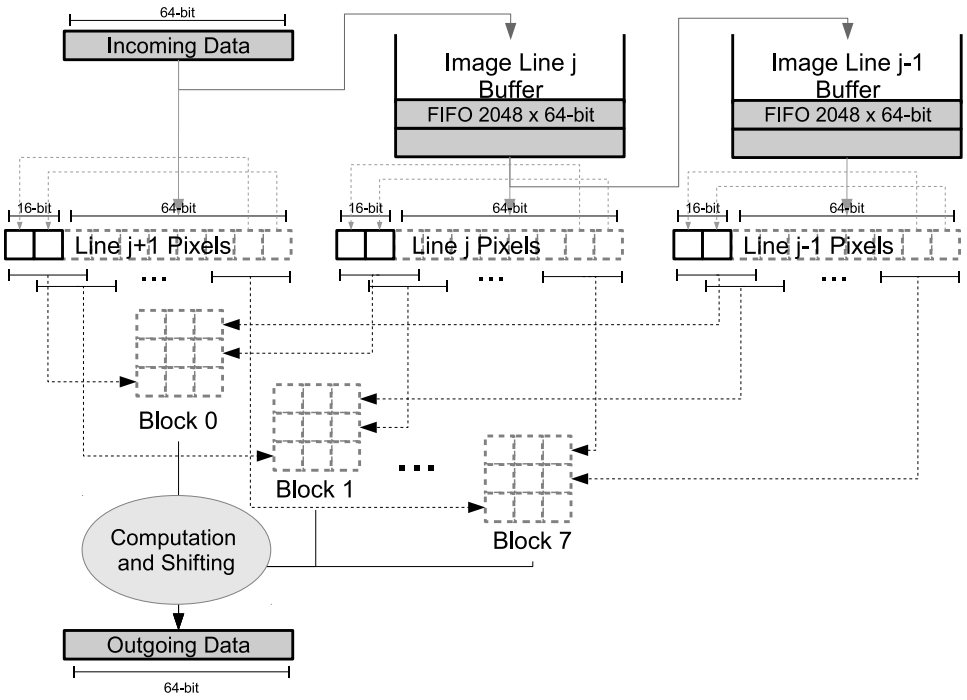

Figure 4.2.4: Data arrangement in memories.
We temporarily keep the image lines into FIFOs. The FIFO structure adds the restriction of the image width not to exceed 16K pixels and to be multiple of 8.

instances. The implementation of each block is presented on figure 4.2.5. The data are then gathered, shifted if necessary, and transmitted to the host.

The total time required for this accelerator is six cycles; one for data reading, three for computations, one for pixel shifting, and one for writing the output. The production of results starts when the first image line is loaded into the "Image Line j Buffer" (of figure 4.2.4). The results are then produced on every cycle at a rate of 1 GPixel/s.

4.2.3 Edge Laplace

The Edge Laplace transformation, is the opposite operation to Gaussian Blur; it subtracts the neighbour related information from a pixel, while the resulting pixel

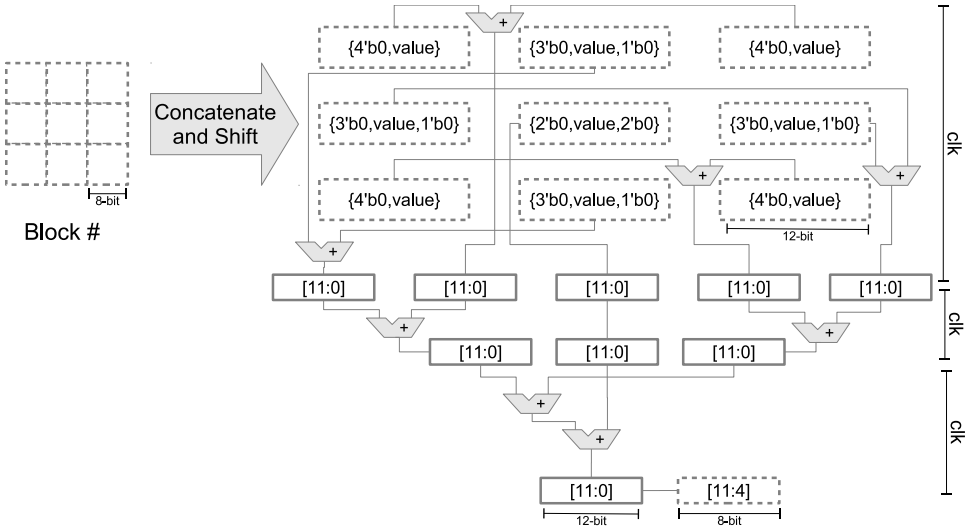

Figure 4.2.5: Computations performed per Block in Gaussian Blur. The multiplications required, are performed through bitwise concatenations, while the intermediate data are kept in 12-bit buffers. The 8-bit wide result is calculated by dividing the total sum by 16 (through discarting the 4 LS bits). The computation lasts 3 clock cycles.

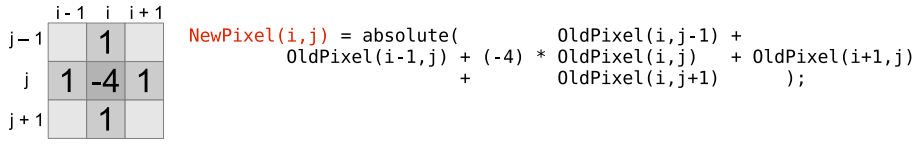

Figure 4.2.6: Algorithm used for Edge Laplace transformation.

is the absolute value of this subtraction. Figure 4.2.6, presents the algorithm and the pixels involved to the calculation.

The hardware implementation of this step, is similar to the Gaussian Blur; the calculation is performed in blocks of 8 pixels. The pixel formation is kept the same, as presented in figure 4.2.4, while the computational block is slightly more complicated due to the use of absolute value. This complication adds an extra delay of two cycles to the design, which is hidden through pipeline. The

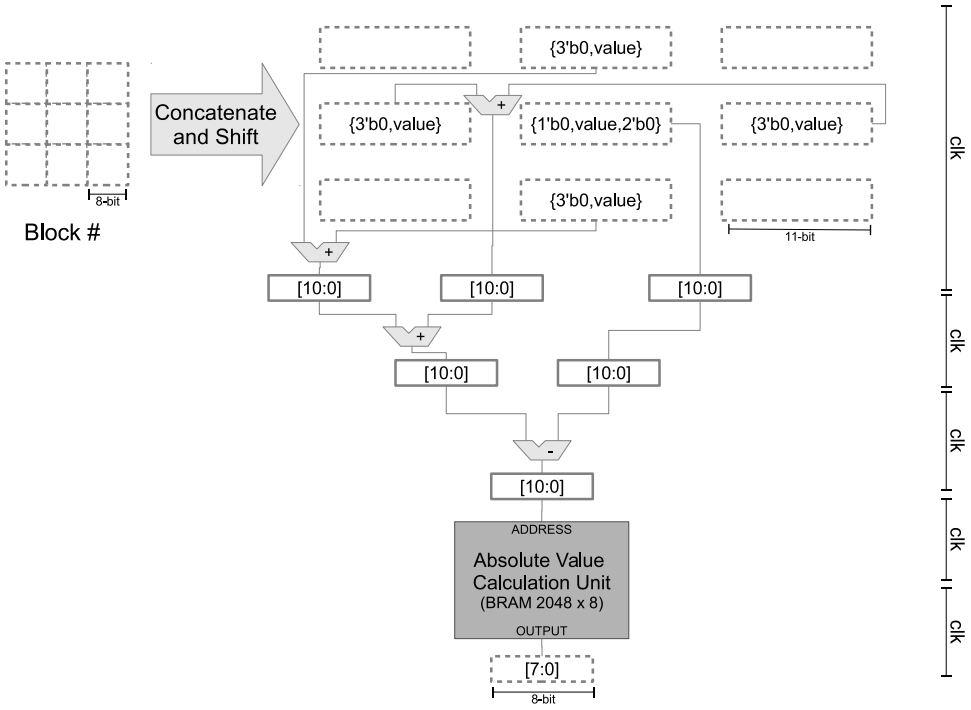

Figure 4.2.7: Computations performed per Block in Edge Laplace. The idea is similar to the Gaussian Blur block. The only challenge we had to face was the calculation of the absolut value, aided by a preloaded Read-Only Block RAM. The total procedure, requires 5 cycles for its completion.

architecture of the Edge Laplace calculation block, is further explained in figure 4.2.7. This accelerator also yields a theoretical throughput of 1 GPixel/s.

4.2.4 Threshold

The threshold is the final algorithmic step we implemented, through which comparing a standard value (in our case 10) to each pixel value, the later is set either to white or to black. The hardware implementation of this step had no special challenge, since the calculation of a pixel value simply requires a comparator and a multiplexer.

In our case however, we decided to reduce the total logic required, by using a

Dataset	720p	1080p
Resolution	1280 x 720	1920 x 1080
Data Sent / Received (KB)	5400 / 3600	12150 / 8100
Theoretical Framerate[1] (fps)	75,5	28,9
Estimated Framerate[2] (fps)	31,9	19,8
Software Implementation Framerate[3] (fps)	13	5,6

[1] The maximum framerate the system can yield, based on its data throughput of half-duplex transactions without reconfiguration and computation costs.

[2] The theoretical framerate, including the reconfiguration costs in the worst case scenario (Noop Scheduling Policy).

[3] It is a single-threaded implementation with a few optimizations, but without SIMD.

Table 4.2.1: The datasets used and the expected framerate. The above calculations, are based on the throughput our system yields. The extra execution overheads can be balanced with the full-duplex capabilities of our system.

subtractor and inverting the most significant bit of the result. The resulting bit is then replicated to the eight bits of a pixel value. Even though the result can be calculated upon a single cycle, we added a single level of registers holding the result of the subtraction. This lead to a higher clock frequency, eliminating the delays occurred due to long wires (routing delays), resulting to a throughput of 1 GPixel/s.

4.2.5 Real World Measurements

The theoretical capabilities of our system combined with its scheduling features, are put into test in this subsection. The accelerators presented earlier in this section, could possibly fit into the resources of a single PRR. The reason for implementing them as four discrete accelerators, is the need for performing at least one partial reconfiguration per frame. In that manner, we are capable of effectively evaluating the scheduling policies available.

The experiments on our system, were based on 720p and 1080p HD images,

processed 100 times by each execution. Table 4.2.1, presents the datasets used, the throughput we expect to gain, and the results of the software implementation. The only factor reducing the framerate is the image size, thus an image is loaded once and processed repeatedly. We also study the behaviour of our system, when multiple executions are performed in parallel. The scheduler is then responsible for selecting the appropriate software process to be served, depending on the contents of each PRR.

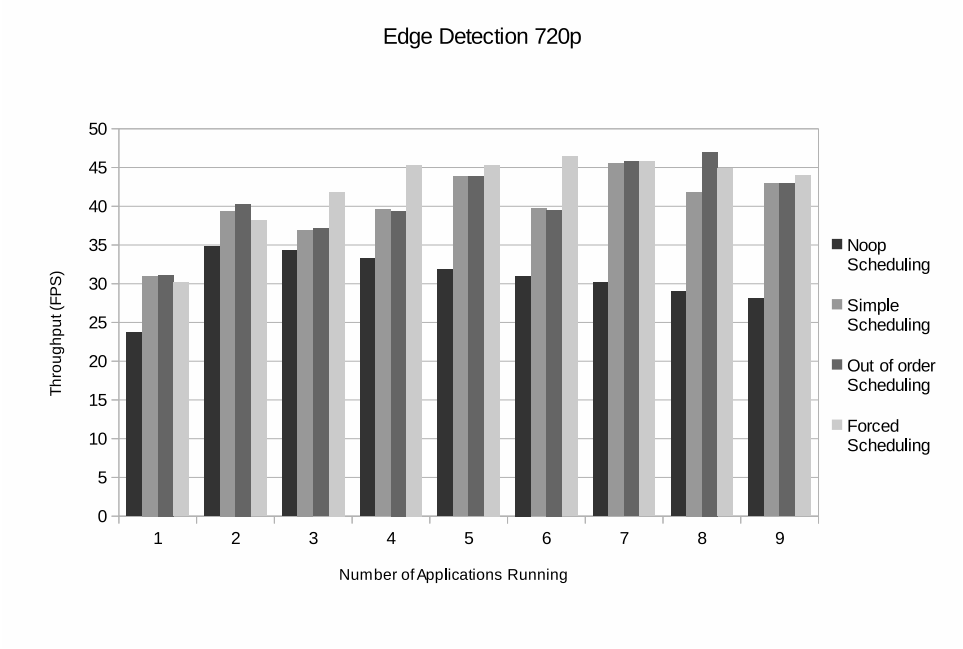

Figure 4.2.8: Cumulative FPS per instances of Edge Detection System for 720p datasets.
The speedup gain of our system, compared to software in the worst case, equals 1.8x. The performance of the "simple" scheduler is close to the performance of the "out of order" scheduler. The "Forced" scheduling policy is the most efficient for this dataset.

In Figures 4.2.8 and 4.2.9 we present the cumulative framerate reached, per number of user software instances at a given time. The first figure is related to the

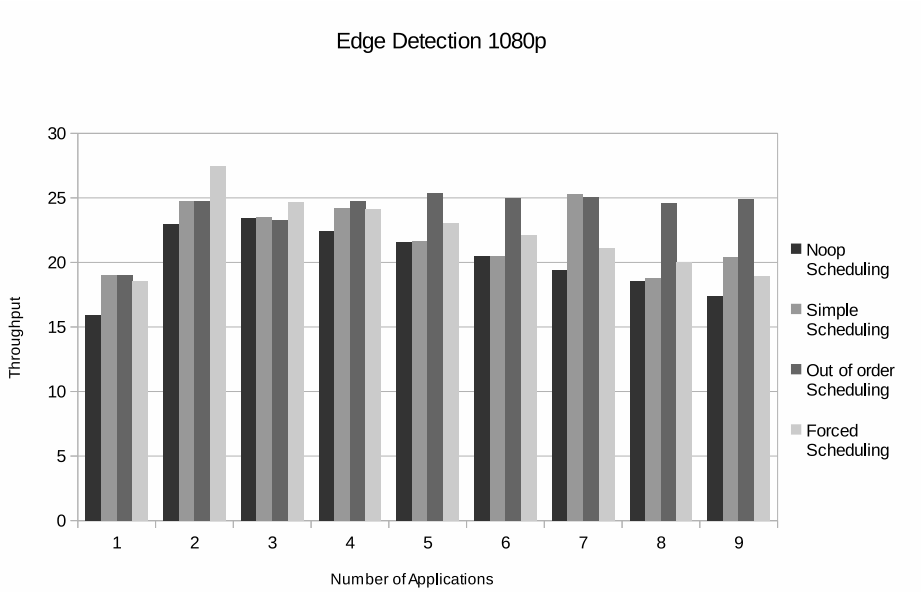

Figure 4.2.9: Cumulative FPS per instances of Edge Detection System for 1080p datasets.
The speedup gain of our system, compared to software in the worst case, equals 3x. The "out of order" scheduling policy is the most efficient and in certain cases the "noop" is very close to the "simple" scheduling. The efficiency of the "forced" scheduling policy is reduced as the number of executions increases.

720p dataset, where the lowest framerate equals 23,7 FPS and the highest 47 FPS. The second figure, is related to the 1080p dataset while the FPS range in this case, is between 15,9 and 27,5 FPS.

In both cases, the lowest throughput is recorded when the system is used by a single application and the "noop" scheduling policy is selected. The experiment results deviate by 25% and 20% (720p and 1080p respectively), compared to the lowest expected framerate (Table 4.2.1). The system framerate on its worst case scenario, is significantly lower than the estimated, due to DMA initiation costs and low device utilization (only one PRR is used).

Scheduler	Noop	Simple	Out of Order	Forced
720p	30,68	40,09	40,79	42,43
1080p	20,24	22,02	24,09	22,22

Table 4.2.2: Average Values per Policy.
The "Forced" policy is the most efficient on 720p datasets, since it reduces the number of reconfigurations to the lowest. The "Out of Order" scheduling is the most efficient on 1080p datasets since it allows multiple instances of the same accelerator.

The highest throughput reachable on the other hand, differs among datasets; in the 720p dataset can be found at the ”out of order” scheduling policy upon eight executions, whereas, two executions of the ”forced” scheduler are most efficient in the 1080p dataset. The reason for such behaviour, is the fact that 720p dataset requires almost equal time to be spent for data transactions as for reconfigurations. This limits the maximum measured troughput, to be 37,7% lower than the theoretical (Table 4.2.1). The 1080p dataset on the other hand, requires a significantly higher amount of time during data transactions rather than reconfigurations. Thus, the maximum framerate reached in this case is very close to the theoretical.

Figures 4.2.8 and 4.2.9, allow us to draw several conclusions regarding the scheduling policies implemented. First of all, all the policies seem to increase performance, until the number of applications reaches a certain point. From that point on, the throughput may either be stabilized, or it may decrease progressively. Secondly, the ”noop” scheduler defines a bottom line of performance, since the reconfiguration procedure is never avoided. The ”simple” scheduling is a better policy, but its higher performance is based on random factors. Finally, the most efficient policies are the ”out of order” and ”forced” scheduling.

There are two factors, defining the selection of the most suitable scheduling policy; equality and performance. The ”simple” scheduling policy, is the best

option for preserving the execution order of the accelerators. There is no priority to existing accelerators, thus all accelerators are equal. Its performance varies randomly, in a range between the ”Noop” and the ”Out of Order” scheduling policies. Performance oriented systems, need to take in account the amount of user data transferred. As table 4.2.2 suggests, the ”forced” policy is preferred when the reconfiguration bitstream size, is smaller than the size of data processed. If the user data are of a significantly higher size, multiple instances of the same accelerator are desirable. As a result, the ”Out of Order” scheduling policy is the most efficient.

Generally, the I/O and the reconfiguration procedure are considerable performance barriers. The total throughput reached by our system, is more than 70% of the maximum theoretical. The partial reconfiguration procedure on the other hand, can be preformed in realtime, or even avoided if possible. The evaluation was aided by a complex application consisting of four accelerators. The system managed to reach real-time performance with 720p HD dataset, even using the less efficient scheduling policy.

The final comment regarding the evaluation of our system, is the fact that the edge detection application, was also tested upon data acquired by mulitple cameras (two web cameras working at 720p). The framerates reached, were slightly lower than those of static data, but the schedulers had similar behaviour. The following section, is dedicated to conclusions and the possible evolution of this work.

4.2.6 Real World Measurements v1.1

At this point, it is proper to present some final measurements taken upon the latest version of our system. The thoeretical maximum throughput is doubled as we migrated from the first to the second generation of PCI Express. The new FPGA board, based on the Virtex 6 chipset, increases the total amount of logic resources available to the user. The following table 4.2.3, is dedicated to the comparison between the LX330T based system and the new system (LX550T).

	LX330T	LX550T
FPGA	Virtex 5 LX330T	Virtex 6 LX550T
Clock Frequency(MHz)	125	250 (PCI Express), 125 (System)
Data Interface	PCI Express v1 x4	PCI Express v2 x4
Reconfiguration Port	ICAP 32bit	ICAP 32bit
FPGA Resources [28], [32]		
Slices	51840[1]	85920[2]
Max Distributed RAM (Kb)	3420	6200
Max Block RAM (Kb)	11664	22752
Full Bitstream Size (MB)	9,86	18,01
Partial Reconfigurable Regions		
Number of PRRs (size)	3	3
Slices	11520[1]	15360[2]
Max Block RAM (Kb)	1728	3456
Partial Bitstream Size (MB)	1,95	2,7

[1] Virtex 5 Slice = 4 x (5-input LUT + Flip Flop)
[2] Virtex 6 Slice = 4 x 6-input LUT + 8 x Flip Flop

Table 4.2.3: FPGA Board Specifications
The I/O throughput is doubled, while the clock remains the same. The total amount of resources available is increased.

The main feature of this system, is the increase of the total I/O throughput (2x), while keeping the interfaces intact. The PCI Express related components operate at 250MHz, while the remaining system components operate at 125 MHz. The partial reconfiguration procedure is still performed through the 32-bit wide ICAP interface (overclocked at 125MHz), while the total amount of reconfiguration data is increased. Generally, the new system is capable of performing faster I/O operations, yet the increase in partial bitstream size, leads to a significant increase in reconfiguration latency. The following figure (4.2.10), presents the performance of our new system as measured form the prespective of the edge detection application.

As figure 4.2.10 suggests, the maximum throughput reached equals 64 frames

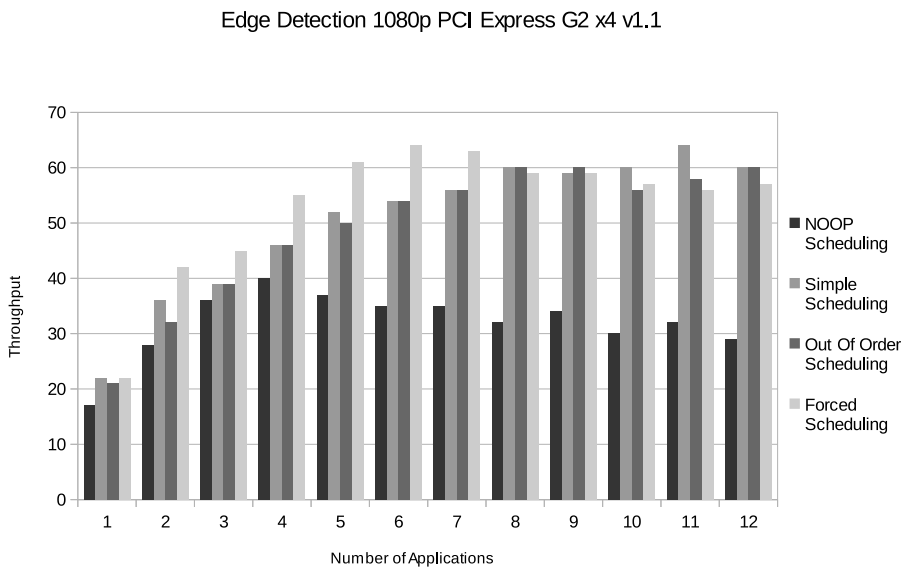

Figure 4.2.10: Cumulative FPS per instances of Edge Detection System for 1080p datasets.
The speedup gain of our system, compared to the previous implementation equals 150%. When a single application accesses the FPGA, the total framerate is just slightly higher compared to the previous system.

per second. This requires multiple applications to access the FPGA simultaneously, in a manner that I/O requests are saturated. The high cost of the reconfiguration procedure, needs to be balanced with multiple applications accessing the same accelerator at a given time. In that way, the total number of reconfigurations performed is reduced and the total throughput is increased.

I am turned into a sort of machine for observing facts and grinding out conclusions.

Charles Darwin

5

Conclusions and Lessons

We developed an efficient system, capable of loading and accessing accelerators, in a completely transparent manner. The I/O and accelerator scheduling does not require the user's involvement, since it is performed in kernel level. Our work allows the execution of multiple reconfigurable hardware accelerators, accessed by either a single, or by multiple users. In that manner, we exploit the hardware resources available, both in terms of I/O, and of reconfigurable logic.

The system is capable of serving an infinite number of software applications in parallel through time division multiplexing; the actual number of applications served at a given time can be equal to the total number of PRRs available. All the other user software applications requesting an execution slot, remain blocking until the driver loads the appropriate accelerator and the results are calculated.

We studied several platforms during the implementation of our system, analyzing the I/O and reconfiguration capabilities of each. Initially, we attempted

to work on the netFPGA platform, where both the I/O and reconfiguration was performed over PCI. The lack of efficiency in I/Os, led us to the selection of the PCI Express interface and the XUPv5 platform. The reconfiguration procedure, performed over JTAG, was the last performance barrier we had to overcome. The usage of internal reconfiguration ports, combined with partial reconfiguration, allowed us to reconfigure the FPGA in real time.

The system on its final version, is based on the PCI Express interface(four lanes wide) and the Virtex 5 LX330T chip. The incoming data, are either redirected to one of the three available reconfigurable regions, or to the ICAP interface. The ICAP is controlled by the systems software, thus the reconfiguration procedure occurs in a completely transparent manner.

The reconfigurable resources available, need to be known to the systems software subsystem at compile time. The driver is flexible enough, for supporting a wide set of hardware platforms, by performing minor changes to it. Even though the number of the available reconfigurable areas needs to be defined, the user may submit a large number of accelerator execution requests. The service order of those requests, is based on the scheduling policy currently selected.

The scheduling policies implemented, allow the user to select between efficiency and fairness, regarding the following accelerator to be executed. It is common to be multiple accelerators of the same type, awaiting for execution at a given time. Sequential scheduling policies ensure fairness, by selecting the following accelerator to be executed, based on the submission order. Out of order scheduling policies implemented, target to reduce the number of reconfiguration operations. Each scheduling policy is selected by the system administrator at driver initialization time, as the user has no knowledge regarding the internal system operation.

We provide a simple, yet well defined set of interfaces between hardware and software, hiding the existence of intermediate layers. Those layers provide execution support for multiple accelerators, as multiple applications consider having exclusive hardware access. The user is not aware regarding the execution state of an accelerator submitted; the software application blocks until the

desired results are fetched. There are applications, transmitting data without expecting results to be retrieved. Those applications are automatically detected and removed from our system, before the accelerator is loaded.

The evaluation of our system, was aided by an edge-detection system, consisting of four dependent accelerators. The output produced by each accelerator, was given as input to the following. The overall procedure required multiple reconfiguration to be performed. The system managed to maintain real time performance, on high definition image datasets, even when the least efficient scheduling policy was selected.

Generally, we implemented a system that could be used in any type of computer system; the developers would be capable of implementing hardware accelerators, hidden in dynamic libraries. The time consuming part of each algorithm, located into software dynamic libraries, can be implemented in hardware. The software library, could check if our system is present and then use the accelerator instead of an optimized, yet software, function. The computation intensive part of the application would then execute in hardware, while the developer would not have to be aware neither of the reconfiguration procedure, nor of the individual I/O operations performed.

5.1 Lessons Learnt

The first and most important lesson derived from this work, was the difficulty of building a custom system based on custom interfaces. During the implementation of this work, we chose not to use a given bus (such as PLB or AXI) and standard devices attached to it. This decision was made for two main reasons; performance and portability. The existence of a standard bus, with multiple pre-built cores placed on it (such as AXI based PCI Express interface, microBlaze controlled ICAP etc.) would definitely increase the overall system overhead. As technology advances, newer interconnections emerge, thus designs built upon a standard bus might become deprecated over time. The system we built, can easily be ported with minor changes to both newer and older FPGA

platforms.

Another important piece of knowledge gained from this work, was related to the placement characteristics of reconfigurable hardware. CAD tools still require manual placement to be performed in order for the system to reach the highest frequency possible. In this work, the lack of a standard bus and its placement attributes, led us to manually place several hardware components. Reaching high frequencies on signals that travel across the FPGA, requires a set of pipeline registers to be added, reducing the total net delay. The overall effort required for resolving such issues through pipeline registers and manual placement, was higher than expected.

The systems software in our system is strictly based on interrupts, which in certain cases are lost. A good idea would be the software to perform polling based event acquisition. A system workqueue could be called every 1 or 2 milliseconds and read a device register. This would definitely reduce the overall interrupt service costs. This deferrable function would reduce the messages transferred between the software and the hardware. We could also read the appropriate registers when the bus is not busy performing DMAs, instead of mixing messages that leads to an overhead increase.

Finally, the final lesson we learned during the implementation procedure, was the importance of a generic driver. The driver we built, is based on standard interfaces, without requiring any special kernel functions. The driver can be compiled and executed on any linux based system, with minor or any changes.

It is a mistake to try to look too far ahead. The chain of destiny can only be grasped one link at a time.

Winston S. Churchill

6 Future Work

In this section, we present possible extensions of this work, both in terms of functionality and performance. There are several possible upgrades, extending the functionality of this system, by increasing the amount of applications an accelerator may serve. The performance oriented upgrades, are strictly related to the technology of the FPGA platform used. The following subsections are dedicated to the presentation of each approach.

6.1 Functionality Oriented Extensions

The system on its current form, supports a single accelerator to be accessed by a single software application. The idea behind those upgrades, is the adoption of multi-threaded hardware accelerators, simultaneously accessed by multiple software applications. This may lead to the reduction of partial reconfiguration

operations, as the available reconfigurable resources of each PRR would be highly utilized.

The system needs to be as similar to its current version as possible, performing only minor changes to the interfaces implemented. The user software interface will be kept as presented in subsection 3.2.2, while the kernel level software will be responsible for accessing the appropriate hardware thread. The hardware interface needs to be backwards compatible to single threaded accelerators, thus the number of available I/O ports will be kept the same.

There are several updates that need to be performed in software. The user level software API can be kept intact, as long as the driver manages the new system functionalities. We need to add several blocks of code to detect and manage the available thread slots on each accelerator. Thus the scheduling policies need to be redefined; the total amount of applications served per accelerator, will be increased. The control over the execution completion and the PRR reconfiguration, needs to consider the number of applications running.

There are minor upgrades to be performed on the static hardware components of our system, mainly altering the interface to the hardware accelerators. As presented in table 6.1.1, the hardware architecture of our system, requires minor register file upgrades; the new 8-bit value, can be taken as the least significant byte of the bitstreamID register. This value is not visible to the user, thus the driver is responsible for scheduling the threads within the accelerator.

The implementation of multithreaded hardware accelerators, requires a notably high amount of effort compared to the appropriate single threaded. Increasing the amount of reconfigurable logic used, leads to several issues related to the available routing resources. More saturated PRRs in terms of logic tend to work in lower clock frequencies. Thus, it would be proper to allow the designer to select between full or half clock frequency.

During the implementation of multi-threaded hardware accelerators, the designer needs to consider the temporary storage of results, until a full DMA block is gathered. The system, contains a single DMA buffer for keeping the

clk	input clock operating at 125 MHz (or optionally at 62,5MHz)
halfClk	(optional) when active, the half clock frequency is selected
resetN	input active low reset signal
incomingData	input 64 bit incoming data
incomingRen	output read enable for incoming data
incomingEmpty	input incoming data are consumed
outgoingData	output 64 bit outgoing data
outgoingWen	output write enable for outgoing data
outgoingFull	input outgoing data not transmitted
reg1,2	input 32 bit general purpose registers
ressreg	output 32 bit general purpose register
complete	output execution complete
bitstreamID	
[31]	when active, the accelerator is multi-threaded
[30:16]	output 15-bit bitstream unique ID
[15:8]	output 8-bit max number of parallel applications accessing the accelerator
[7:0]	output 8-bit current application index number
nxtApp	input 8-bit index of next thread to execute

Table 6.1.1: Updated Accelerator Interface
The new / updated ports are marked in bold, adding an 8-bit value to the existing hardware interface, while extending the 32-bit bitstream ID value.

results produced by a single-threaded accelerator. The contents of this buffer are transmitted to software through DMA. In time multiplexed multi-threaded hardware accelerators, calculating a block of results requires several blocks of data to be processed by each thread. Thus, the designer needs to store the intermediate results of each thread on its own buffer. Once a data block is calculated, the designer can copy it to the outgoing data buffer and then transmit it.

There are two strategies a designer may follow during the implementation of a multi-threaded hardware accelerator; the duplication based and the resource sharing based. The duplication based, requires a copy of the computational core per thread. The resource sharing based, requires the implementation of a mechanism capable of storing and loading the contents of registers. The

following subsections present those strategies in more detail.

6.1.1 Duplication Strategy

Duplication based strategy is designed to ease the implementation procedure. The idea is simple; for each hardware thread supported, there is a clone of the computational core. Each thread has its own dedicated registers, while switching between contexts, can be simply performed by driving the appropriate register load signals. Figure 6.1.1 presents the multi-threaded version of the Greyscale accelerator (subsection 4.2.1) designed upon this strategy.

The main drawback of this approach is the large amount of resources required by each thread. Increasing the number of supported threads, linearly increases the amount of logic required. Several resources are left unused during the execution; only one thread executes at a time, the other simply stall. The following strategy seems to manage the resources used in a more adaptive manner.

6.1.2 Resource Sharing Strategy

The idea in this case is similar to context switching in CPU; the register values can be stored into a Block RAM cells during the stall state. In that way, we are capable of saving the state of the accelerator executed. Once that state is stored, we need to backup the intermediate data located into memory segments (including the results). Once all the intermediate data are transferred into BlockRAM segments, the system may assign the same accelerator to another software application.

All the register outputs, are driven into the context switch Block RAM. Upon stall state we enable the write-enable signal, leading to state recording. When in stall state, the system may perform context switch by writing the output of the Block RAM on the system registers. Figure 6.1.2, presents the Greyscale accelerator designed upon this strategy. It supports up to eight software applications to be executed in parallel. The implementation cost is high, but the amount of resources used is lower compared the previous strategy.

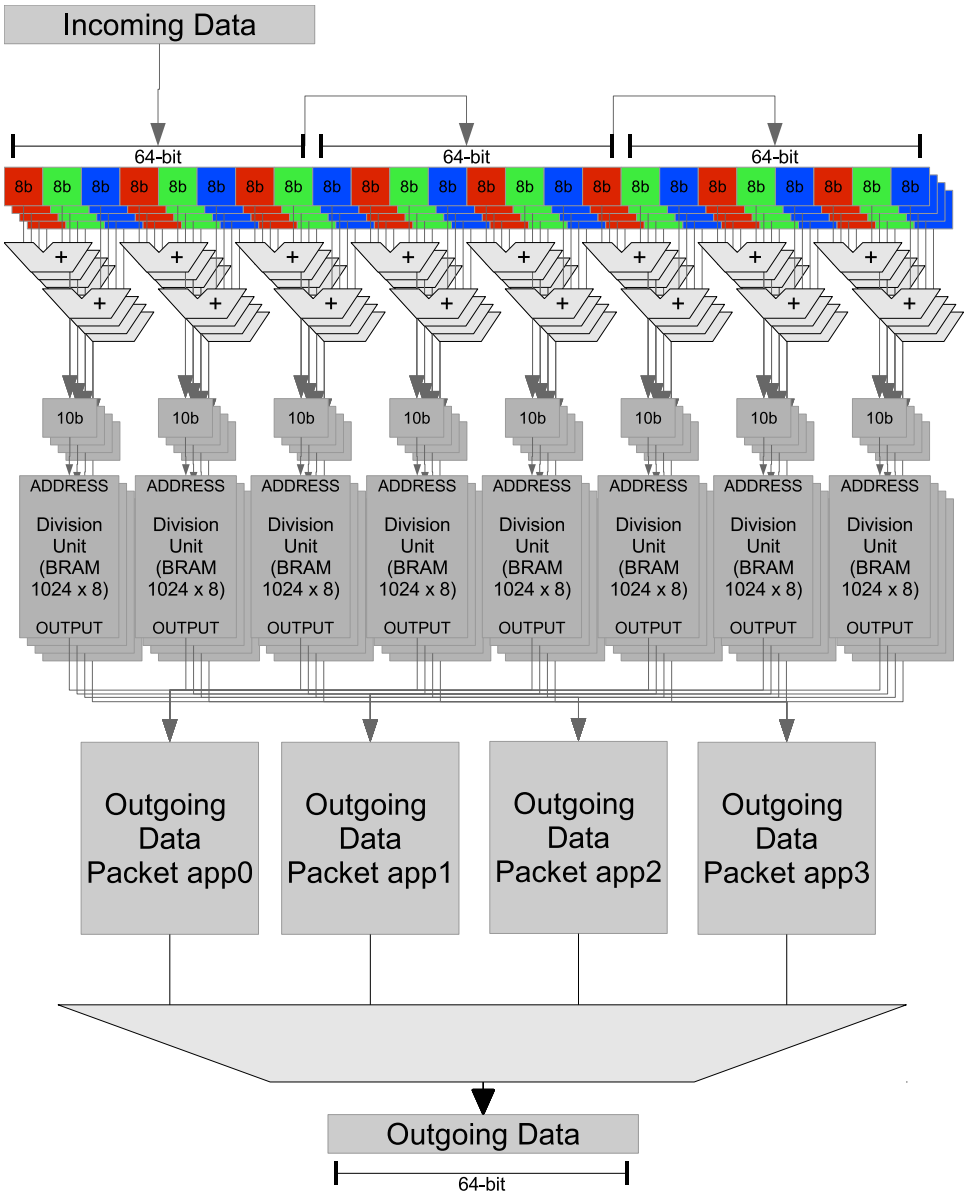

Figure 6.1.1: Multi-threaded duplication based Greyscale Accelerator. There is one core dedicated to each thread, supporting a total of four threads.

Generally the functionality expansions in our system, are not targeting to a performance increase. Performance oriented system upgrades are presented into

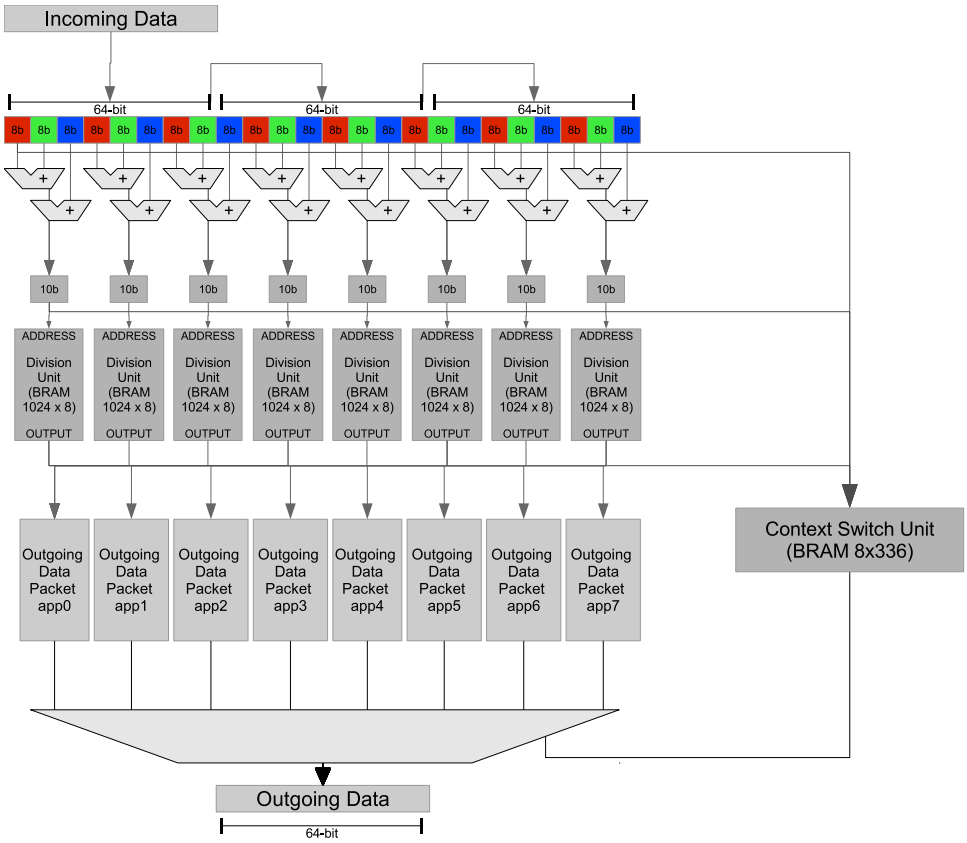

Figure 6.1.2: Multi-threaded resource sharing based Greyscale Accelerator. We chose to exclude from this figure the outputs of the Context Switch Unit (CSU). They are given as input to registers through a multiplexor.

the following section.

6.2 Performance Oriented Extensions

The performance of this work, is strictly deteriorated by the capabilities of the selected platform and I/O interface. Newer patforms, would allow us to create a greater number of larger PRRs, increasing the number of the accelerators running simultaneously. Even in the current platform, it would be a reasonable expectation to increase the number of the PRRs to eight, if the board could

provide us with a higher quality of routing resources.

The PCI Express interface, on its latest (third) version, yields a theoretical throughput of 1 GB/s per lane, which is four times higher than the theoretical of our system. The maximum I/O throughput is the main performance barrier in our case, since a high amount of applications relies on streaming data. Thus either by moving to the second version to the PCI Express interface, or by doubling the number of lanes, the performance is doubled.

In the existing system, using double buffering techniques might lead to a performance increase. The incoming and outgoing data buffers, are large enough to hold a single 32KB packet of data. Increasing their size to double, would give a performance boost, when a single PRR is used. The data transfers could be initiated before either the input is consumed, or the result buffer is full. The accelerator in that manner, would never enter in stall state, and the I/O interface would be saturated. In our case, during the execution of three accelerators, the I/O interface is always saturated, thus the performance increase would be less significant.

Despite the restrictions added by the hardware subsystem, there are a few upgrades that can be performed in systems software. These upgrades regard mostly the implementation of two other scheduling policies; the external scheduling and the type based scheduling. External scheduling, would allow a scheduler running in user level, to define the selection of the following accelerator to be executed. Type based scheduling, would combine computation intensive with I/O intensive accelerators, at a given time in a ratio of 2:1. In that way, stall times would be reduced to minimum.

The implementation of a more intelligent scheduling policy would also be a good idea; the highest performance was gained in out of order and forced policies. The system could combine those policies in order to achieve the highest performance possible. The new policy supported would normally allow different accelerators on different PRRs. In certain cases, where there is a high amount of requests for a certain accelerator type, this policy would allow two instances of the same accelerator to be loaded.

The Driver is implemented as a character device; there is an amount of data copied byte by byte between the user and kernel level. Those copies are performed parallel to execution, thus the overhead is merely hidden when multiple applications access our system. This issue arises when the system serves a single software application, where the total amount of copy-related overhead is paid. Altering the inner structure of the driver by converting it to block based, might yield a performance increase. The driver could map pages into the user memory and then transmit those same pages, reducing the total number of data copies. Altering the driver might lead to a more complex software interface.

As operating system virtualization is a reality in modern systems, the integration of hardware accelerators in virtual machines would be a fruitful idea. This could be achieved in two ways; either by creating a communication channel between the system driver and the virtual machine(VM), or by assigning the PCI Express device directly to the VM. The first case, would allow the access to the device from both the host and the guest operating systems, while a set of specific drivers would be required. The second case, where the PCI Express device would be assigned to the VM, would allow our system to be running directly onto the guest operating system.

References

[1] Netfpga. URL http://netfpga.org.

[2] A. Agne, M. Happe, A. Keller, E. Lubbers, B. Plattner, M. Platzner, and C. Plessl. Reconos: An operating system approach for reconfigurable computing. *Micro, IEEE*, 34(1):60–71, Jan 2014. ISSN 0272-1732. doi: 10.1109/MM.2013.110.

[3] Altera. Arria v gx fpga development board. URL http://www.altera.com/literature/manual/rm_avgx_fpga_dev_board.pdf.

[4] R. Bittner. Speedy bus mastering pci express. In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pages 523–526, Aug 2012. doi: 10.1109/FPL.2012.6339270.

[5] Ray Bittner. Bus mastering pci express in an fpga. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '09, pages 273–276, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-410-2. doi: 10.1145/1508128.1508176. URL http://doi.acm.org/10.1145/1508128.1508176.

[6] J. Bucek, P. Kubalik, R. Lorencz, and T. Zahradnicky. Comparison of fpga and asic implementation of a linear congruence solver. In *Digital System Design (DSD), 2013 Euromicro Conference on*, pages 284–287, Sept 2013. doi: 10.1109/DSD.2013.125.

[7] NVIDIA Corporation. *NVIDIA CUDA C Programming Guide*, April 2012.

[8] S.K. Dhawan. Introduction to pci express-a new high speed serial data bus. In *Nuclear Science Symposium Conference Record, 2005 IEEE*, volume 2, pages 687–691, Oct 2005. doi: 10.1109/NSSMIC.2005.1596352.

[9] David Dye. Partial reconfiguration of xilinx fpgas using ise design suite. May 2012. URL http://www.xilinx.com/support/documentation/white_papers/wp374_Partial_Reconfig_Xilinx_FPGAs.pdf.

[10] G. Gibb, J.W. Lockwood, J. Naous, P. Hartke, and N. McKeown. Netfpga - an open platform for teaching how to build gigabit-rate network switches and routers. *Education, IEEE Transactions on*, 51(3):364–369, Aug 2008. ISSN 0018-9359. doi: 10.1109/TE.2008.919664.

[11] John L. Hennessy and David A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006. ISBN 0123704901.

[12] Intel. *Intel® G965 Express Chipset Family Datasheet*, July 2006. URL http://www.intel.com/Assets/PDF/datasheet/313053.pdf.

[13] Intel. *Intel® X58 Express Chipset Datasheet*, November 2009. URL http://www.intel.com/content/dam/doc/datasheet/x58-express-chipset-datasheet.pdf.

[14] M. Jacobsen and R. Kastner. Riffa 2.0: A reusable integration framework for fpga accelerators. In *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, pages 1–8, Sept 2013. doi: 10.1109/FPL.2013.6645504.

[15] M. Jacobsen, Y. Freund, and R. Kastner. Riffa: A reusable integration framework for fpga accelerators. In *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, pages 216–219, April 2012. doi: 10.1109/FCCM.2012.44.

[16] Henry Kasim, Verdi March, Rita Zhang, and Simon See. Survey on parallel programming model. In Jian Cao, Minglu Li, Min-You Wu, and Jinjun Chen, editors, *Network and Parallel Computing*, volume 5245 of *Lecture Notes in Computer Science*, pages 266–275. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-88139-1. doi: 10.1007/978-3-540-88140-7_24. URL http://dx.doi.org/10.1007/978-3-540-88140-7_24.

[17] Chris Lomont. *Introduction to Intel® Advanced Vector Extensions*. URL https://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions.

[18] K. Papadimitriou, C. Vatsolakis, and D. Pnevmatikatos. Invited paper: Acceleration of computationally-intensive kernels in the reconfigurable era. In *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2012 7th International Workshop on*, pages 1–5, July 2012. doi: 10.1109/ReCoSoC.2012.6322874.

[19] PCI-SIG. *PCI Express Base 2.0 Specification*, . URL http://www.pcisig.com/specifications/pciexpress/base2.

[20] PCI-SIG. *PCI Express 3.0 Frequently Asked Questions*, . URL http://www.pcisig.com/specifications/pciexpress/resources/PCIe_3.0_External_FAQ_Nereus_9.20.pdf.

[21] PCI-SIG. *PCI Express Base Specification*, April 2012.

[22] Ariel Ortiz Ramirez. An overview of intel's mmx technology. *Linux J.*, 1999 (61es), may 1999. ISSN 1075-3583. URL http://www.linuxjournal.com/article/3244.

[23] O. Sander, S. Baehr, E. Luebbers, T. Sandmann, Viet Vu Duy, and J. Becker. A flexible interface architecture for reconfigurable coprocessors in embedded multicore systems using pcie single-root i/o virtualization. In *Field-Programmable Technology (FPT), 2014 International Conference on*, pages 223–226, Dec 2014. doi: 10.1109/FPT.2014.7082780.

[24] Duy Viet Vu, O. Sander, T. Sandmann, S. Baehr, J. Heidelberger, and J. Becker. Enabling partial reconfiguration for coprocessors in mixed criticality multicore systems using pci express single-root i/o virtualization. In *ReConFigurable Computing and FPGAs (ReConFig), 2014 International Conference on*, pages 1–6, Dec 2014. doi: 10.1109/ReConFig.2014.7032516.

[25] J. Wiltgen and Ayer J. Xapp1052, bus master dma performance demonstration reference design for the xilinx endpoint pci express solutions, September 2011. URL http://www.xilinx.com/support/documentation/application_notes/xapp1052.pdf.

[26] Xilinx. Vc707 evaluation board for the virtex-7 fpga. URL http://www.xilinx.com/support/documentation/boards_and_kits/vc707/ug885_VC707_Eval_Bd.pdf.

[27] Xilinx. *Virtex-II Pro and Virtex-II Pro X FPGA User Guide*, November 2007. URL http://www.xilinx.com/support/documentation/user_guides/ug012.pdf.

[28] Xilinx. *Virtex-5 Family Overview*, February 2009. URL http://www.xilinx.com/support/documentation/data_sheets/ds100.pdf.

[29] Xilinx. *Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet*, June 2011. URL http://www.xilinx.com/support/documentation/data_sheets/ds083.pdf.

[30] Xilinx. *Virtex-5 FPGA Configuration User Guide*, October 2012. URL http://www.xilinx.com/support/documentation/user_guides/ug191.pdf.

[31] Xilinx. *LogiCORE IP Endpoint Block Plus v1.15 for PCI Express User Guide*, October 2012. URL http://www.xilinx.com/support/documentation/ip_documentation/pcie_blk_plus/v1_15/pcie_blk_plus_ug341.pdf.

[32] Xilinx. *Virtex-6 Family Overview*, August 2015. URL http://www.xilinx.com/support/documentation/data_sheets/ds150.pdf.

[33] Lihong Zhang, Xiaoling Yang, and Wenhua Yu. Acceleration study for the fdtd method using sse and avx instructions. In *Consumer Electronics, Communications and Networks (CECNet), 2012 2nd International Conference on*, pages 2342–2344, April 2012. doi: 10.1109/CECNet.2012.6201608.

[34] G. Zumbusch. Tuning a finite difference computation for parallel vector processors. In *Parallel and Distributed Computing (ISPDC), 2012 11th International Symposium on*, pages 63–70, June 2012. doi: 10.1109/ISPDC.2012.17.