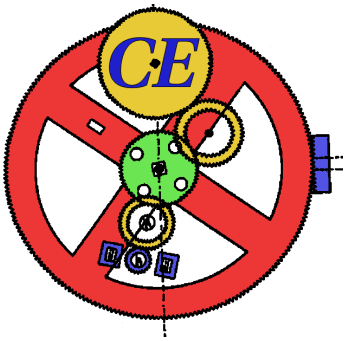


Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2010

MSc THESIS

General Purpose Computing with Reconfigurable Acceleration

Anthony Arthur Courtenay Brandon

CE-MS-2010-28

Abstract

In this thesis we describe a new generic approach for accelerating software functions using a reconfigurable device connected through a high-speed link to a general purpose system. In order for our solution to be generic, as opposed to related ISA extension approaches, we insert system calls into the original program to control the reconfigurable accelerator using a compiler plug-in. We define specific mechanisms for the communication between the reconfigurable device, the host general purpose processor and the main memory. The reconfigurable device is controlled by the host through system calls provided by the device driver, and initiates communication by raising interrupts; it further has direct accesses to the main memory (DMA) operating in the virtual address space. To do so, the reconfigurable device supports address translation, while the driver serves the device interrupts, ensures that shared data in the host-cache remain coherent, and handles memory protection and paging. The system is implemented using a machine which provides a HyperTransport bus connecting a Xilinx Virtex4-100 FPGA to the host. We evaluate alternative design choices of our proposal using an AES application and accelerating its most computationally intensive function. Our experimental results show that our solution is up to 5× faster than software and achieves a processing throughput equal to the theoretical maximum.

TU Delft

Faculty of Electrical Engineering, Mathematics and Computer Science

General Purpose Computing with Reconfigurable Acceleration

generic reconfigurable computing platform

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Anthony Arthur Courtenay Brandon
born in Paramaribo, Suriname

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

General Purpose Computing with Reconfigurable Acceleration

by Anthony Arthur Courtenay Brandon

Abstract

In this thesis we describe a new generic approach for accelerating software functions using a reconfigurable device connected through a high-speed link to a general purpose system. In order for our solution to be generic, as opposed to related ISA extension approaches, we insert system calls into the original program to control the reconfigurable accelerator using a compiler plug-in. We define specific mechanisms for the communication between the reconfigurable device, the host general purpose processor and the main memory. The reconfigurable device is controlled by the host through system calls provided by the device driver, and initiates communication by raising interrupts; it further has direct accesses to the main memory (DMA) operating in the virtual address space. To do so, the reconfigurable device supports address translation, while the driver serves the device interrupts, ensures that shared data in the host-cache remain coherent, and handles memory protection and paging. The system is implemented using a machine which provides a HyperTransport bus connecting a Xilinx Virtex4-100 FPGA to the host. We evaluate alternative design choices of our proposal using an AES application and accelerating its most computationally intensive function. Our experimental results show that our solution is up to 5× faster than software and achieves a processing throughput equal to the theoretical maximum.

Laboratory : Computer Engineering
Codenumber : CE-MS-2010-28

Committee Members :

Advisor:	Stephan Wong, CE, TU Delft
Advisor:	Ioannis Sourdis, CE, TU Delft
Chairperson:	Koen Bertels, CE, TU Delft
Member:	Georgi Gaydadjiev, CE, TU Delft
Member:	Rene van Leuken, CAS, TU Delft

To my family, friends, and my parents in particular, for always supporting me.

Contents

List of Figures

List of Tables

Acknowledgements

I want to thank Stephan Wong and Georgi Gaydadjiev for allowing me to work on this project. Furthermore I want to show my gratitude towards Ioannis Sourdis for helping me publish a paper, and taking the time to read and comment on my thesis. Lastly I want to thank Google for sponsoring this project.

Anthony Arthur Courtenay Brandon
Delft, The Netherlands
December 13, 2010

Achievements

Conference Publications

- Anthony A.C. Brandon, Ioannis Sourdis, Georgi N. Gaydadjiev, **General Purpose Computing with Reconfigurable Acceleration,** *20th Int. Conf. on Field Programmable Logic and Applications (FPL), Milano, Italy, August 2010.*

1 Introduction

Generally there are two different ways to solve computational problems, either in hardware or in software. A hardware solution such as an application-specific integrated circuit (ASIC) has the advantage of being fast and optimized for a particular task, however the design and fabrication is very expensive. Software on the other hand is slow but flexible in that it can be changed over time, and is also cheaper to develop. Reconfigurable computing proposes a compromise between these two by allowing the hardware to be reconfigured to allow different applications, while still providing large performance benefits over software. By combining reconfigurable computing with general purpose computing, we can implement critical parts of an application in hardware, while implementing non critical parts in software, removing the need to implement the entire application on a reconfigurable device.

Integrating Reconfigurable computing with traditional Computing systems has been a challenge since the early days of FPGAs; several machines have been designed to this direction, such as PAM[1], Splash[2], and DISC[3]. The primary drawback of all these attempts was the limited bandwidth and increased communication latency between the host general purpose processor and the reconfigurable device. In order to overcome these bottlenecks, other solutions proposed a more tightly coupled reconfigurable unit integration, e.g. Garp[4], Chimaera[5], PRISC[6], and OneChip[7]. Recently, however, several components have been released which support standard high-speed communication between general purpose processor(s), memory and other peripheral devices. These solutions use high-speed on-board links such as the Intel QuickPath[8] and the AMD HyperTransport bus[9] and provide multi-GByte/sec low latency communication. The above developments offer a new opportunity for integrating reconfigurable computing in general purpose systems.

Rather than building a machine from scratch or suggesting fundamental architectural changes, it is more performance- and certainly cost-efficient to propose a generic solution that uses existing off-the-shelf components with only software (and configuration) modifications. As opposed to approaches that require ISA extensions to the host processor, such as Molen[10], Garp[4] and others, we describe a generic solution that requires only an FPGA device driver, few compiler extensions and a reconfigurable wrapper in the FPGA.

Significant work has been done in automatic software to Hardware-Description-Language translation in order to simplify the use of Reconfigurable Acceleration by the programmer (e.g. DWARV[11] and ROCCC[12]). However, there are remaining unsolved issues towards a smooth integration of Reconfigurable acceleration in a General Purpose machine. Several commercial systems have been released offering reconfigurable acceleration combined with traditional computers, however, they are in general hard to program. Examples of such machines are the Altix SGI[13], and the Convey[14], which

however target High-Performance computing rather than the General Purpose domain.

The remainder or the introduction is organized in four sections. Section 1.1 describes the problems with current platforms for combining General Purpose computing with reconfigurable computing. It also names three requirements for creating a generic, general purpose reconfigurable computing platform. In Section 1.2 we state several features which the proposed platform should support in order to facilitate the integration of reconfigurable computing with a general purpose computer. Finally, Section 1.3 gives an overview of the remaining chapters of this thesis.

1.1 Problem Statement

All the various reconfigurable platforms mentioned above still suffer from various drawbacks, some of which are solved in some but not in others. The first drawback is that of data bandwidth. Because applications which are good candidates for reconfigurable acceleration are usually also applications which process large amounts of data, it is very important that the accelerators have a low latency and high bandwidth connection to the source of the data. Some architectures such as GARP[4], Chameleon[15], Chimaera[5], OneChip[7] and PRISC[6] attempt to solve this by integrating the reconfigurable accelerator in the same chip as the general purpose processor; however, this requires redesign of the chip, and a different fabrication process making it expensive. As mentioned above, standard high speed buses such as QuickPath and HyperTransport, which are specifically developed for high speed connections between processors, main memory and peripherals with latencies of less than 200ns[16], help reduce the bottleneck presented by data access.

Another drawback of many platforms is that applications developed for a certain platform are entirely dependent on the specific hardware or development tool-chain used in that platform. This means that significant work is needed to port either the application or accelerator (or both) to a different implementation of the same platform. Some platforms such as Molen[10], Convey[14], and Altix[13] address this by specifying a specific interface to the accelerator and software. However even these platforms still have some drawbacks.

The next drawback is that most of these systems are not suitable for general purpose computing. Convey and Altix are aimed at the high performance computing market, meaning supercomputers and clusters used for computationally intensive tasks such as simulations in scientific research, instead of general purpose computing. Even a platform such as Molen, which is not aimed at high performance computing, nor uses any custom hardware, is not ideal for general purpose computing. Because Molen is implemented entirely on an FPGA board, there is no readily available operating system that will run on it while giving the same performance and features as an operating on a general purpose machine. Additionally, the lack of support for address translation for memory accesses from the accelerator, makes it necessary to either modify the operating system so it no longer requires virtual memory, or to use a separate memory at a fixed physical address, which is used to transfer data between software and the accelerator.

The final drawback of reconfigurable computing systems in general is the programmability. The common solutions to programming for these systems are either through some new or extended programming language which is then transformed by the compiler into

some form which uses hardware, or through function calls which manage the reconfigurable device. The ideal situation for the programmer is of course to use an existing programming language without any modifications, and having the compiler automatically use hardware acceleration where possible. This requires extensive changes to the entire development tool-chain, from the compiler, to the assembler, and linker. The other extreme is to have the programmer manage the hardware entirely by hand through a library or other software interface.

These problems with the existing reconfigurable computing architectures can be summarized in the following requirements for our platform:

- The reconfigurable accelerator should be connected to the host system using an existing low latency/high bandwidth link.
- Applications should be easily ported to a different implementation of the same platform.
- The platform should be implemented using readily available, off-the-shelf components to keep the machine costs low.
- The platform should add support for reconfigurable computing to a machine without extensive modifications to the operating system.
- The platform should be programmable in a simple way, while requiring only small changes to the tool-chain for a working prototype.

1.2 Objectives

In the previous section we describe the requirements for a generic, platform-independent solution for reconfigurable acceleration in a general purpose system. In order to meet these requirements we propose several features which allow us to more efficiently integrate a reconfigurable device in a traditional computer system, addressing issues related to the memory accesses, programming interface, and system-level support. The proposed system must provide the following:

- **Virtual shared memory:** The reconfigurable accelerator must have direct access to the main memory, which is shared with the host, using virtual addresses. Allowing the accelerator to access main memory through virtual addresses accomplishes three things. The first is that because the accelerator has access to main memory, there is no need to explicitly transfer data from main memory to local memory of the FPGA. Secondly, by using virtual addresses and address translation we can ensure that the reconfigurable accelerator cannot access memory regions it is not supposed to. Lastly, allowing the accelerator to use virtual addresses also facilitates the use of pointers in the accelerator. Otherwise all pointers to data would have to be detected, and translated before being sent to the accelerator.

- **Memory protection and paging:** All memory accesses of the reconfigurable accelerator must be checked for permissions, and page segmentation must be considered. Modern operating systems offer memory protection by letting each application execute in its own address space. This means that applications can only access data which belongs to them, and this must also be enforced for any reconfigurable accelerator attached to the system. Furthermore, the operating system can swap certain pages in memory to disk, to increase the total amount of usable memory. Before the accelerator can access a page in memory we must ensure that it is restored to memory.

- **Interrupt support:** The reconfigurable device must use Interrupts to indicate TLB misses and execution completion. Using interrupts allows the reconfigurable device to notify the operating system of these situations without the need for the OS continuously polling a register, which wastes computing cycles with waiting instead of performing useful computations.

- **Controlling the Reconfigurable device:** We will use system calls to lock and to release the device, execute a function, and read/write from/to the device. Using system calls for these actions instead of ISA extension, allows us to avoid any modifications to the hardware, and also the extensive changes needed to the tool-chain to support new instructions.

- **Compiler extension:** The above system calls will be inserted in the original code using a modified compiler after annotating the function to be accelerated. This extension allows the programmer to mark functions which will be executed in hardware, without needing to explicitly call the system calls from his code, and without requiring major modification to the development tool-chain.

1.3 Overview

This thesis consists of a further 4 chapters: The next chapter, Chapter 2 discusses some of the background on reconfigurable computing. We explain the difference between course grained and fine grained reconfigurable fabrics, and how a they can be integrated into a traditional computer as a co-processor, or as part of the General Purpose Processor. We then look at the application of reconfigurable computing in the High-Performance Computing domain, and how it is used in the Convey and SGI Altix architectures. Finally we also look at how the Molen architecture solves problems such as programmability, the need for many opcodes to execute different hardware accelerators, and portability of applications.

In Chapter 3 the proposed architecture and implementation are described. The architecture consists of a compiler plug-in, a device driver and a hardware wrapper. We explain how the compiler plug-in replaces calls to functions with calls to hardware and how the driver is used to manage the reconfigurable device. The details of implementing virtual shared memory, memory protection, interrupt support and communication between the host and accelerator are also explained.

Chapter 4 describes the final implementation platform and the various components that were used. These include the HyperTransport bus used to connect the FPGA to the host, the HyperTransport core used by the FPGA to communicate over the link, and the FPGA board with HyperTransport connector.

Chapter 5 provides an overview of the results of testing the implementation. First the platform used for implementation is described, followed by a description of the encryption application used for testing. This is followed by an evaluation of the performance based on a comparison between accelerated and non-accelerated application.

Finally, in Chapter 6 we draw our conclusions and discuss future improvements to the platform.

2 Background

In chapter 1 we explained how reconfigurable computing can be used to speed up software applications by performing some of the computations in hardware, without requiring expensive ASICs. Reconfigurable computing is becoming more and more popular as FPGAs continue to improve in size and performance, and as performance requirements, both at home and for scientific computing, continue to increase.

When designing a reconfigurable computing architecture combined with a general purpose computer, there are two important design questions. Most important may be weather the architecture will be fully reconfigurable, or a hybrid, using both reconfigurable and general purpose computing. Both approaches have advantages and drawbacks, however because the goal of this project is to create a general purpose machine which supports reconfigurable computing, we will ignore the fully reconfigurable architectures, since these would not support the use of an operating system. The next is what kind of reconfigurable fabric should be used. The fabric can be either course-grain, fine-grain, or some combination of both. FPGAs such as the Virtex-4 combine elements of both course and fine-grain architectures. The second question is how to integrate the reconfigurable fabric with the general purpose computer. The designer can choose to either implement the reconfigurable device on the same chip as the processor, or to it as a separate co-processor.

Over the years there have been several attempts at commercial machines using reconfigurable computing. Most of these are aimed at high performance computing (XD1[17], Altix[13], Convey[14], SRC[18]). These systems are sold as supercomputers with added support for reconfigurable computing. Because these systems are somewhat similar to our proposed design we will take a close look at the Convey and Altix machines later on in the chapter.

The architecture we propose is based on the Molen Polymorphic Processor. Molen has several features which make it ideal to use as a starting point for our design. Molen uses a PPC processor with a reconfigurable co-processor to accelerate functions in an application. It also extends the instruction set with 8 instructions, of which only 4 are needed for basic functionality. The reason that this is a good starting point for a new architecture is that communication between the host and reconfigurable device is already defined. It allows us to use the Molen programming paradigm which allows the programmer to mark functions which will be executed in hardware. The 4 instructions can easily be replaced with system calls. Basing our design on Molen also gives us access to existing applications with accelerators for testing.

This chapter is organized as follows. Section 2.1 describes what a reconfigurable fabric is and how they are used. In Section 2.2 we discuss how a reconfigurable fabric can be integrated into a traditional computing system. Subsequently, in Section 2.3 we discuss high performance computing with reconfigurable acceleration, and take a more

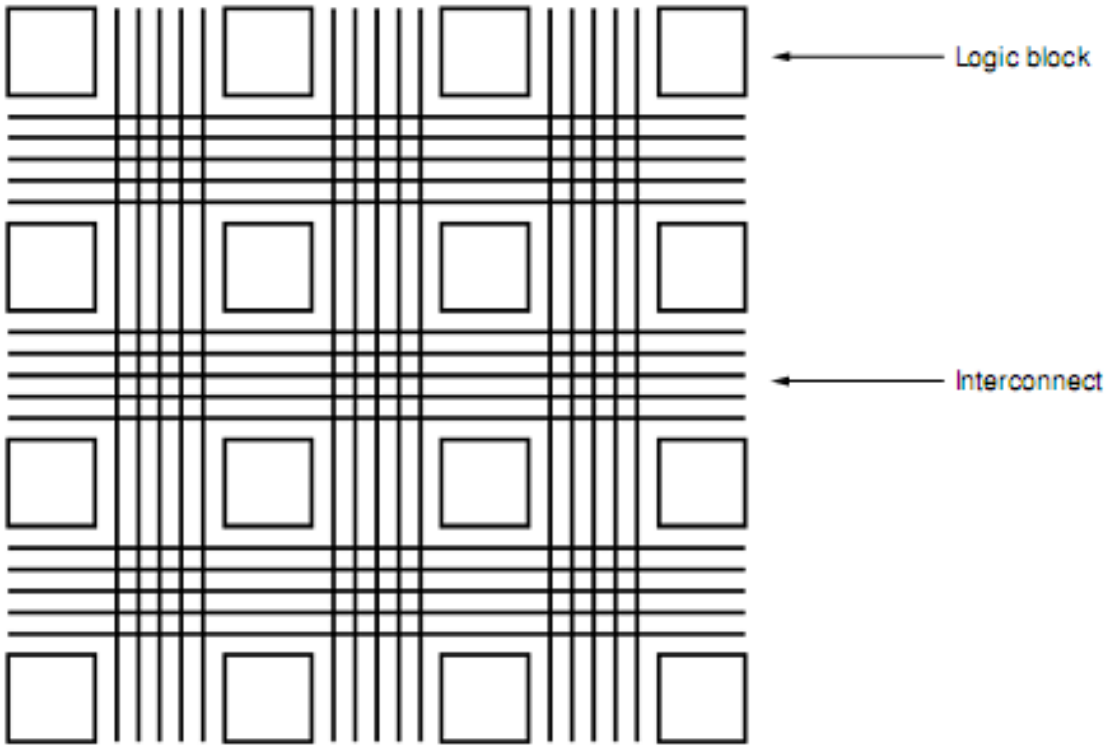


Figure 2.1: An example of how Computational blocks can be connected in a reconfigurable fabric. Source: Reconfigurable Computing[19].

detailed look at the commercial architectures Altix[13] and Convey[14]. In Section 2.4 we present the MOLEN polymorphic architecture[10]. Finally, in Section 2.5 we give a short summary of the topics discussed in this chapter.

2.1 Reconfigurable Fabric

The reconfigurable fabric is one of the basic requirements for reconfigurable computing. The reconfigurable fabric is a specially designed chip consisting of computing elements and an interconnect such as in Figure 2.1. The grid formation shown in the figure is not necessarily the way these devices are implemented but the underlying principles are the same. The connections between different elements can be switched on or off to determine the behavior of the chip. There are two styles of reconfigurable fabrics: Fine-grain, and course-grain. In fine-grain reconfigurable fabrics, the computational elements used to create functionality in the device operate on a single bit, whereas in course-grain architectures the logic blocks operate on multiple bits. The result is that functionality which requires single-bit manipulation can be efficiently implemented in a fine-grain architecture. The downside is that the number of logic blocks required to recreate the same functionality as in a course-grain architecture will be higher, resulting in lower clock frequencies.

Because of the large number of computational elements, it is possible to perform

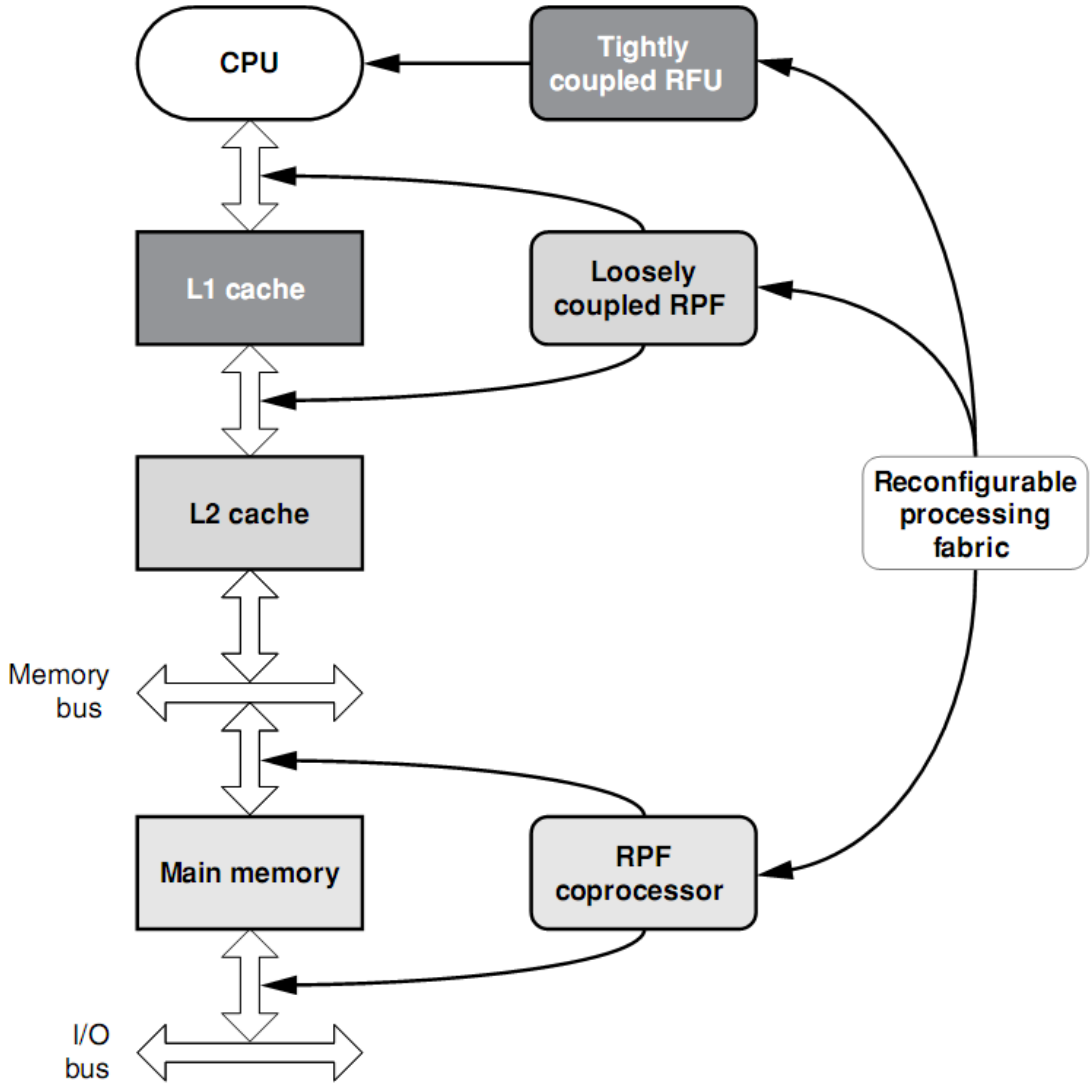


Figure 2.2: A block diagram showing the memory hierarchy of a computer. The RF can be loosely coupled, tightly coupled or attached as co-processor. Source: Reconfigurable Computing[19].

many calculations in parallel. This parallelism is what reconfigurable computing exploits in order to speed up applications. When building a reconfigurable architecture, the designer can choose between using an existing reconfigurable device, or to design one for maximum performance. This is beneficial when the architecture targets a specific application domain, which can benefit from a custom reconfigurable fabric. The downside is the cost involved in designing and fabricating a new device.

The most commonly used reconfigurable fabrics are Field Programmable Gate Arrays (FPGAs). FPGAs consist of a fine-grain reconfigurable fabric with additional components, such as adders, multiplier, and memories in order to ensure good performance in

common cases such as adders, multipliers and memory, but also to minimize the number of used resources for custom logic. Early FPGAs were used to implement programmable logic on circuit boards; however, due to the increasing circuit complexity of FPGA devices it is now possible to use them for implementing complex functionality. Modern FPGAs are large enough to implement general purpose processors and prototype ASIC designs.

2.2 Memory Hierarchies for Reconfigurable Computing

In general purpose computing with reconfigurable acceleration, the reconfigurable fabric must somehow be connected to an existing host processor. Figure 2.2 shows the different levels at which the fabric can be connected in the memory hierarchy. The reconfigurable fabric is on the same chip as the GPP in both the tightly and loosely coupled case, while the co-processor is on a separate chip. When tightly coupled, the reconfigurable fabric is used as one of the functional units of the GPP as seen in Chimaera[5]. A loosely coupled fabric is a separate component which is connected to the data-bus in order to communicate with memory and the processor. The co-processor is a separate chip housing the RF, which is then connected to a peripheral bus, allowing it to communicate with main memory and optionally the host processor.

Each of these approaches has different advantages and disadvantages. The more tightly coupled the RPF is, the better the memory access performance. In the case of a tightly coupled RPF, it has the same memory access times as any other functional unit in the processor. Whereas in the case of a co-processor there will be a higher latency and lower bandwidth. However integrating a tightly coupled RFU in a system requires altering the design of the CPU, and even in the case of a loosely coupled RFU, it must still be placed on the chip. Whereas a co-processor can be attached to the bus much like any other peripheral.

2.3 Reconfigurable High-Performance Computing

High-Performance Computing (HPC) refers to the use of supercomputers and computer clusters to perform scientific and financial computations requiring high performance. Examples of such applications are: simulation of weather models, protein folding, DNA matching, and constructing three dimensional models in searching for oil and gas reserves. High-Performance computers deliver the required performance by exploiting parallelism in the applications, and by using the most powerful processors available.

Because Reconfigurable computing is inherently efficient in performing many parallel computations, there has been a move towards the use of Reconfigurable High-Performance Computing. Two currently available HPC systems with reconfigurable computing are SGI Altix[13] with RASC[20] and Convey Computer's HC-1[14]. These two platforms both use FPGAs and GPPs to allow for HPC with reconfigurable acceleration, although the two approaches are different. In Altix the programmer must know more about the platform, and must explicitly use the reconfigurable accelerator through a library. In the Convey, the compiler does automatically detects when the accelerator

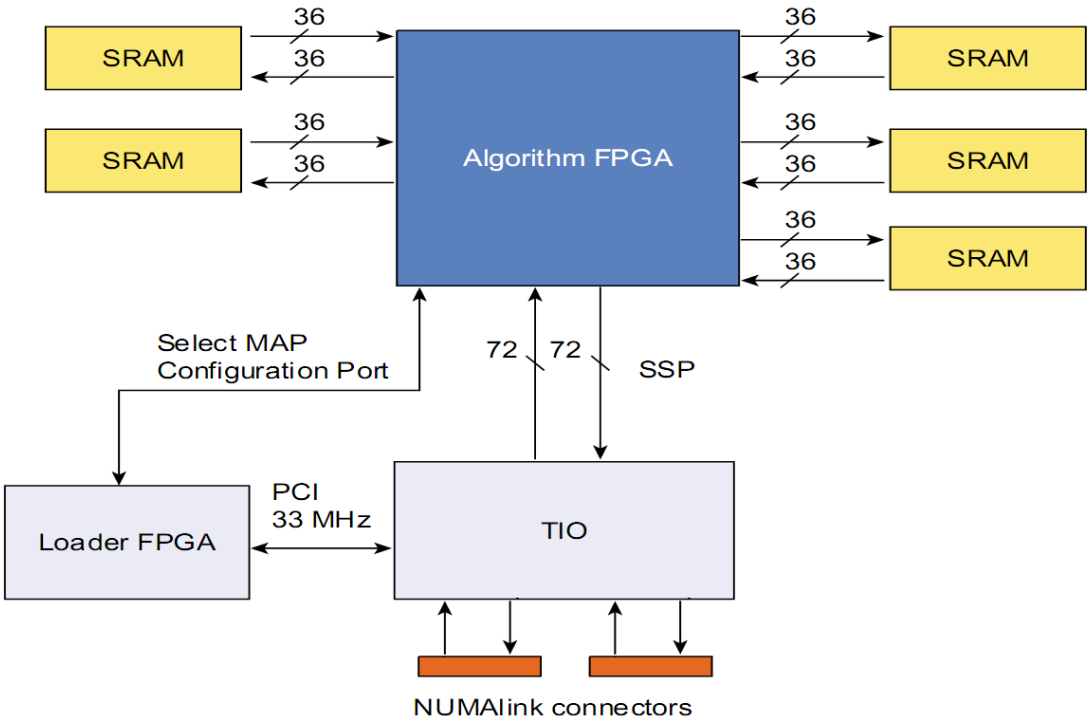


Figure 2.3: Block diagram showing how the FPGA is connected to the NUMAlink interconnect and several banks of SRAMs. Source: Reconfigurable Application-Specific Computing User's Guide[20].

can be used, allowing the programmer to program without learning any details about the system.

2.3.1 ALTIX

SGI Altix provides support for reconfigurable computing in the form of their reconfigurable Application Specific Computing (RASC) program[20]. In RASC, an FPGA (Figure 2.3) is connected to the SGI NUMAlink[20] interconnect as a co-processor. NUMAlink is a high bandwidth and low latency interconnect which is used to connect processors, memory, and other components in Altix machines. The FPGA is also connected to several banks of SRAMs, which it uses to store data locally. The FPGA is divided into the Core Services block and the re-programmable Algorithmic block. The Algorithmic block implements in hardware an application specific algorithm which is used to speed up the execution of an application. The Core Services block provides the interface between the Algorithmic block and the host system, and to do so implements the following features: It Implements the Scalable System Port (SSP), which allows communication over the NUMAlink; Provides read and write access to the SRAMs from both the host system and the algorithmic block; Allows single and multi step debugging of the algorithmic block; And provides access to the algorithms debug port and registers.

Figure 2.4 shows that a device driver provides access to the FPGAs core services from

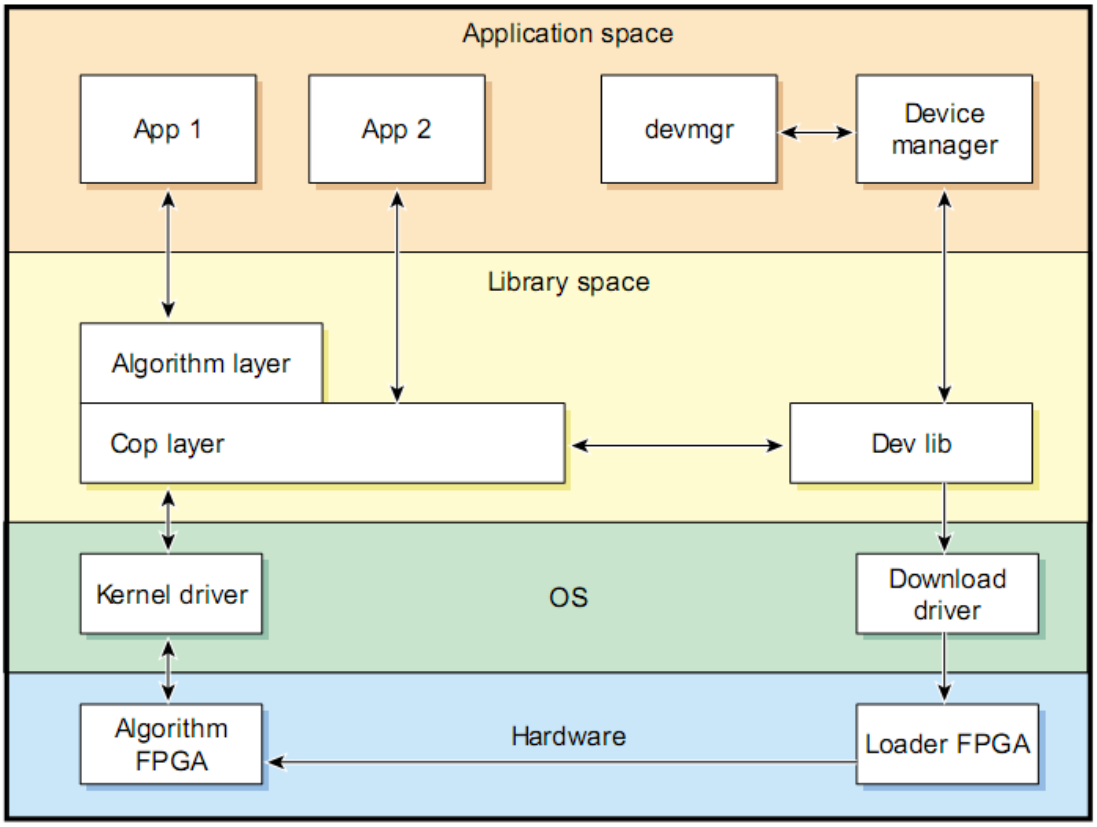


Figure 2.4: Block diagram showing the different levels of abstraction of the Algorithm FPGA. Source: Reconfigurable Application-Specific Computing User's Guide[20].

software through system calls. In order to avoid programming using low level system calls, the FPGA can be accessed through a library called RASC Abstraction Layer (RASCAL) which abstracts the system calls and device management. The programmer uses library calls to reserve and use devices; the programmer must also explicitly transfer data to and from the SRAMs using library calls. As seen in the figure, the library provides 2 levels of abstraction which cannot be mixed in a single application. The Cop layer deals with individual devices, while the Algorithm layer presents multiple devices as a single device to the programmer.

2.3.2 CONVEY

The Convey architecture (Figure 2.5) consists of off-the-shelf Intel processors in combination with a reconfigurable co-processor. The Convey Instruction Set Architecture[21] extends the x86_64 instruction set with instructions executed by the co-processor. The co-processor can be reconfigured at run-time to execute different instructions, depending on the application. The instruction executed by the co-processor are grouped into "personalities" which can be programmed into the co-processor at run-time. A personality is essentially an instruction set tailored to a specific application, and defines all

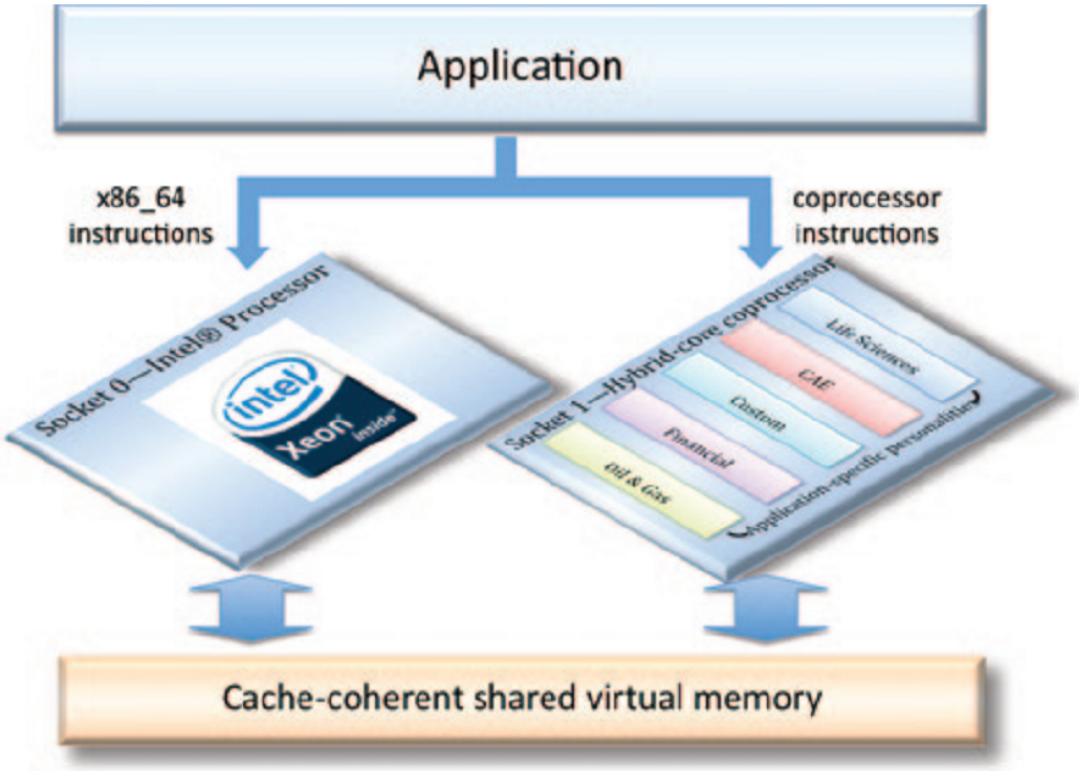


Figure 2.5: The Convey Architecture with Intel host processor, co-processor and shared virtual memory. Source: Convey HC-1 Architecture White Paper[21].

instructions that can be executed by the co-processor. Each personality has a minimal set of instructions which are implemented in all personalities. While the system can have multiple personalities, only a single personality can be loaded into the co-processor at any time. Furthermore the co-processor shares a cache-coherent view of the global virtual memory with the host processor.

The co-processor consists of three components (Figure 2.6): the Application Engine Hub (AEH), Memory Controllers (MCs), and the Application Engines (AEs). The AEH is responsible for the interface to the host processor and I/O chipset, and fetching instructions. The Memory Controllers perform address translation for the AEs and perform the DMA requests to deliver data to the AEs. This allows the co-processor to work on the same virtual address space as the host processor. The AEH and MCs implement the instructions which are shared among all personalities to ensure that basic functionality such as memory protection, access to co-processor memory and communication with the host processor is always possible. The AEs are the units that execute the remainder of the instructions in the personality.

Programming for the Convey architecture is done in standard C/C++ or FORTRAN. The compiler then emits code for both the Intel host processor and the co-processor, in the same executable. During compilation, the compiler uses a state description of the instructions in a personality to determine what code can be executed on the co-processor.

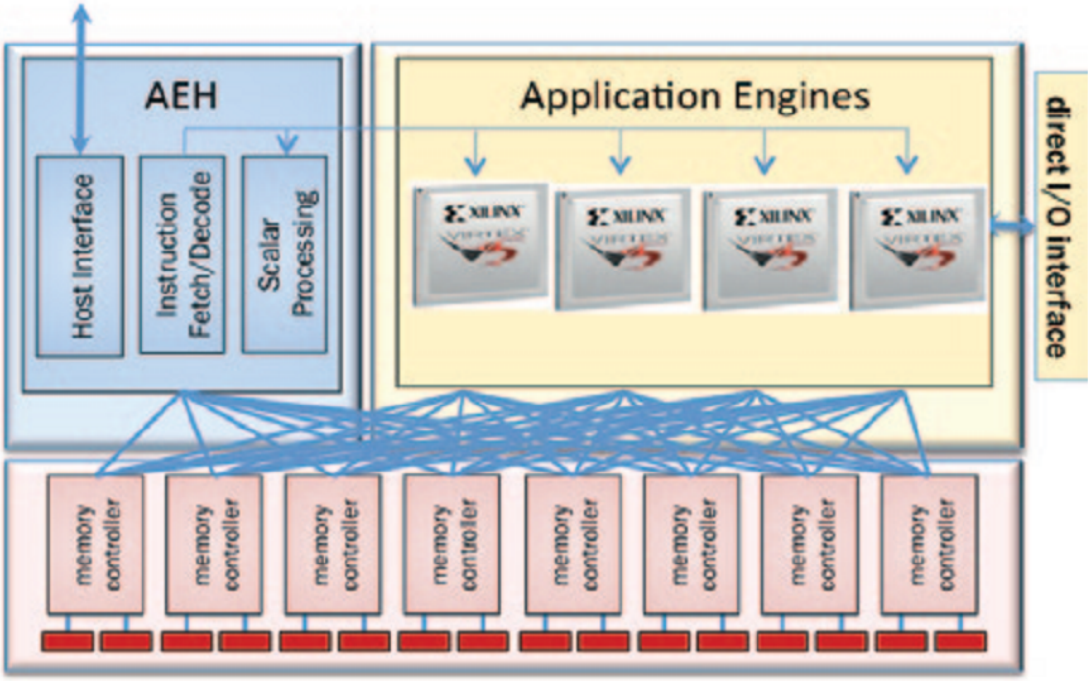


Figure 2.6: The coprocessor is divided into the Application Hub, Memory Controllers, and Application Engines. Source: Convey HC-1 Architecture White Paper[21].

The compiler also emits instructions which start the co-processor by writing to a memory location. This approach allows existing software to be recompiled without changes in order to make use of the advantages of the architecture.

2.4 MOLEN

The MOLEN polymorphic processor[10] is a reconfigurable architecture designed to solve several of the issues with many reconfigurable architectures. Namely, how to make an architecture where applications can be easily ported to different hardware using the same platform without major redesign. The MOLEN architecture (Figure 2.7) consists of a GPP and a reconfigurable accelerator which can communicate through a set of registers called Exchange Registers (XREG). A program is executed on the GPP, with certain computationally intensive functions implemented as accelerators. While the program is executing, whenever one of these functions is needed, the appropriate accelerator is run in order to compute the result. Parameters and results are exchanged between the GPP and accelerator through the Exchange Registers. Instruction Set Architecture (ISA) extension is used to control the reconfigurable accelerator.

In the proposed architecture only 8 instruction are needed to execute any number of functions in the accelerator. These 8 instructions are:

- c-set. Complete set. this instruction configures the hardware accelerator for a specific function.

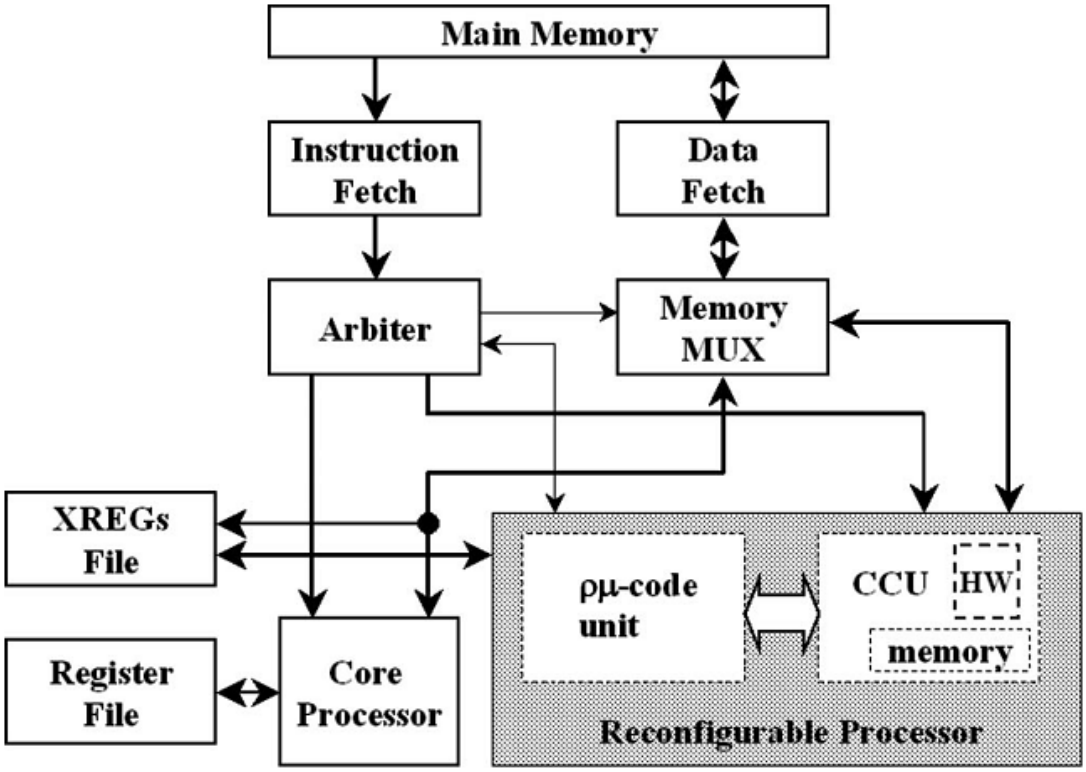


Figure 2.7: Block diagram showing the Molen architecture organization. Source: The MOLEN Polymorphic Processor[10].

- p-set. Partial set, this instruction configures only part of the accelerator. This is useful when the accelerator can perform multiple functions, and only one is needed at a specific time.
- execute. This instruction signals the accelerator to start executing.
- set pre-fetch. Prefetch the reconfiguration code for the accelerator.
- execute pre-fetch. Prefetch code to be executed on the accelerator.
- break. Used to synchronize execution of the accelerator and the GPP. This instruction causes the GPP to halt until the accelerator is finished executing.
- movtx. Move to exchange register. This instruction is used to write parameters to the exchange registers.
- movfx. Move from exchange register. Writes the return result to the exchange register.

Of these 8 instructions, a minimal set of these consists of the 4 instructions: c-set, execute, movtx, movfx. These 4 instructions are needed to provide the simplest working implementation. Using these instructions it is possible to implement an arbitrary number

Table 2.1: Overview of the features in the Altix, Convey, and Molen reconfigurable computing platforms.

Machine	High Speed Link	Virtual Addresses	Platform Dependent Code	ISA Extension
Altix	x		x	
Convey	x	x		x
Molen				x

of hardware accelerators, thus eliminating the problem of opcode explosion due to adding different accelerators. An arbiter is used to determine whether instructions are to be executed by the GPP or by the accelerator.

Programs for the MOLEN architecture are written in C. Functions which will be accelerated in hardware are marked with a pragma, which tells the compiler to insert instructions for setting the accelerator, writing the arguments to the XREGs and executing the accelerator, instead of calling the function.

The MOLEN architecture is implemented on an FPGA with the PPC as general purpose processor. This means that it uses a fine-grain fabric, and it is loosely coupled.

2.5 Conclusions

In this chapter we have discussed several aspects of reconfigurable computing. First we explained what a reconfigurable fabric is, and how it is used in reconfigurable computing. A reconfigurable fabric is made up of multiple processing elements which are connected by a programmable interconnect. The functionality of the fabric is determined by how the different processing elements are connected. FPGAs are reconfigurable devices which can be reprogrammed multiple times and are now advanced enough to be used for complex calculations and simulations because of the large number of processing elements and high clock speeds compared to when they first came out.

Next we discussed the three ways to integrate a reconfigurable fabric into the memory hierarchy of a general purpose computer. The RF can be either loosely coupled, tightly coupled, or connected as co-processor. The co-processor has the advantage that no modifications are required for the chip, however the memory access times can become a bottleneck because of this.

Ultimately we discuss the use of reconfigurable computing in high performance computing (HPC). Several companies now offer high performance supercomputers or clusters with support for reconfigurable computing, including Convey and SGI. This is mostly because Reconfigurable computing is very efficient at solving problems with a large amount of parallelism. We also discuss in more detail two of the currently available systems for HPC, namely the Convey and Altix SGI systems. Convey uses ISA extension and a compiler which automatically detects where these new instructions should be inserted. The new instructions are implemented in an FPGA which has access to main memory trough virtual addresses. Altix on the other hand uses library calls to control the reconfigurable

device, and instead of the FPGA having access to shared memory, the programmer must transfer data from main memory to the FPGAs local memory and back. We also discuss the Molen platform, which consists of a PPC processor and a hardware accelerator as co-processor. Table 2.1 gives an overview of the features of these three platforms.

3 System Architecture

In order to solve the problems identified in Chapter 1 we propose a system for General Purpose computing with a reconfigurable accelerator as co-processor. The proposed system consists of a general purpose machine with a reconfigurable device used for accelerating in hardware computationally intensive functions. Such functions are annotated in the code to indicate that they will be executed in hardware. The compiler is then responsible for generating the binary where the arguments of the respective functions are sent to the reconfigurable device and the return results are read from the device. In order to facilitate the execution of functions in hardware, the reconfigurable device works on **virtual shared memory**, supports **memory protection**, **cache coherence**, **paging**, and virtual to physical **address translation** maintaining a local TLB copy.

Figure 3.1 illustrates the overview of the proposed system. The *Reconfigurable device* (FPGA card) is connected to the *host* general purpose processor and to the *main memory* through a *high-speed link*. On the host side, a *driver* has been developed to support the communication of the FPGA device with the host and the main memory. On the reconfigurable device a module is used to *interface with the high-speed link*. The reconfigurable device further contains the *reconfigurable accelerator* intended to speed-up a software function; depending on the application running in the machine arbitrary accelerators can be designed. Finally, we have designed a *wrapper* around the reconfigurable accelerator to support the integration of the FPGA device in the system. More precisely, the wrapper controls the accelerator and performs tasks such as address translation, DMA operations, handles the interrupts caused by the device (IH), and maintains some memory-mapped IO regions: exchange registers (XREGs), the TLB copy, and control and status registers.

The system supports three different types of communication between the host, the memory and the FPGA device as indicated with red lines in Figure 3.1. The host can write and read data from the FPGA(1), the FPGA can send data to the host in the form of an interrupt(2), and the FPGA can read and write data to/from the main memory(3). More precisely, the following communication types are supported:

- **Host to/from FPGA:** The host processor can send the arguments of the hardware-accelerated function to the FPGA and also read the return result of the reconfigurable accelerator. In addition, the host sends control signals to the FPGA in order to initialize the device and start the execution of the accelerator. Finally, the host can write new entries to the TLB copy of the FPGA device.

- **FPGA to Host:** The FPGA initiates a communication with the host by raising an interrupt in case of a TLB miss or execution completion.

- **FPGA to/from Memory:** The FPGA performs DMA reads and writes from

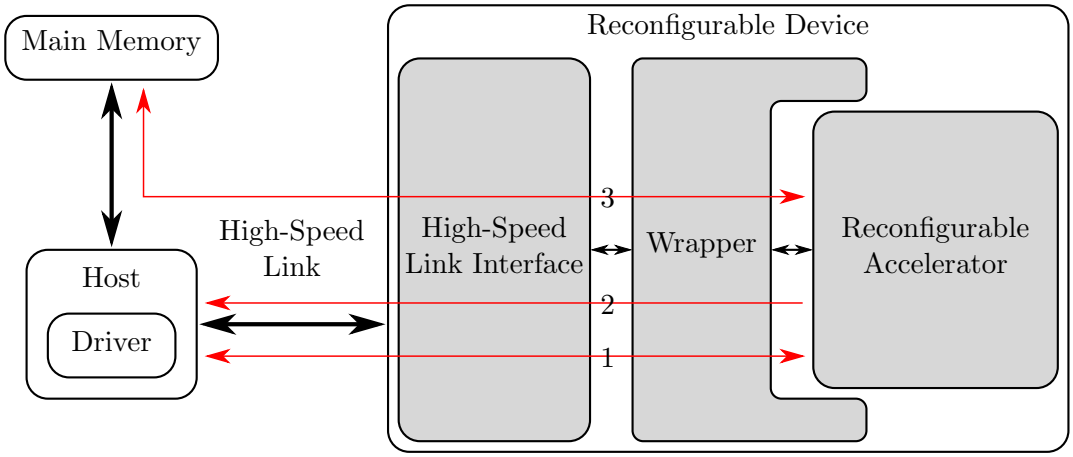


Figure 3.1: Block diagram of the proposed system.

and to the shared main memory in order to feed the reconfigurable accelerator with input data and to write output data back, respectively.

In the following subsections we explain the mechanisms to provide hardware acceleration, virtual shared memory support, memory protection, cache coherence, and paging in the proposed system. We describe in more detail the functionality of the driver, the compiler extensions, and the FPGA Device.

The remainder of this chapter is organized as follows. Section 3.1 describes how the device driver interfaces with hardware and software, and how it provides the features mentioned above. In Section 3.2 we present the compiler plug-in and explain how it replaces the annotated functions with calls to hardware. Section 3.3 describes the implementation of the FPGA device, including support for virtual memory and three approaches to improve DMA performance. Finally, in Section 3.4 we summarize these three sections.

3.1 The Driver

In order to use the FPGA device from software there are two options. The first is to use a library which implements the necessary initializations and communication with the device. The second is to implement this in a device driver. The main advantages of using a library to do this are that it is simpler to write and debug a library than it is a driver. Also if there is bug in the driver, there is a good chance of crashing the entire system, while this is not the case for the library. Both of these are because the driver is executed inside the kernel, which means standard debugging techniques cannot be applied, and bugs can directly effect the kernel. However the disadvantage of using a library is that it is not possible to support interrupts. Because interrupts are needed to implement execution completion and address translation effectively, we use a device driver.

The driver is responsible for several things. First of all it must provide communication between the software part of the application and the hardware instance of the

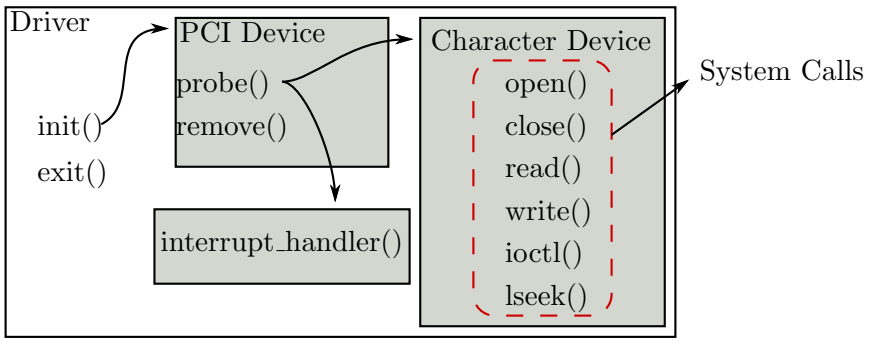


Figure 3.2: Overview of the driver.

accelerated function. To do this the driver provides an API consisting of several system calls. Secondly, the driver must also handle any interrupts from the device, and lastly the driver also initializes the device when the driver is loaded.

3.1.1 Application Programming Interface

The programming interface of the driver provides the system calls shown in listing 3.1. These system calls are used to implement three of the eight instructions defined by MOLEN. These are the *execute*, *xreg read* and *xreg write* instructions. The execute instruction is implemented using the `ioctl()` system call, while the xreg read and write instructions are implemented using the `read()`, `write()` and `lseek()` system calls. The `open()` and `close()` system calls are used by an application to get a lock on the device and unlock the device respectively and do not correspond to any instruction defined by the MOLEN architecture.

3.1.1.1 Open and Close

The `open()` system call is used by a program to get a lock on the device specified by `pathname`. In doing so it performs the following tasks: First it acquires a semaphore to lock the device. If this fails the device is already in use by a different application, and the call returns an error value. Next it stores a pointer to the process structure of the calling process. This structure contains pointers to various other structures, including one which contains all the information about the virtual address space belonging to the process. These structures are needed in order to perform address translation in the interrupt routine. Finally, it allocates an array which is used to store all the pages mapped by the device during its execution.

The `close()` system call releases the lock acquired by the `open()` system call. It performs these two tasks: first it frees the array used to store the mapped page entries, and secondly it releases the semaphore used to lock the device.

3.1.1.2 Read, Write and Lseek

The `write()` and `read()` system calls are used in combination with `lseek()` to write the arguments of function calls to the exchange registers (XREGs) of the FPGA device

Listing 3.1: Function prototypes of system calls provided by device driver.

```
/*Open and lock the device and
return an integer which identifies it.*/
int open(const char *pathname, int flags);

/*Release the lock obtained by calling open.*/
int close(int fd);

/*Read count bytes from the device into buf.*/
ssize_t read(int fd, void *buf, size_t count);

/*Write count bytes from buf to the device.*/
ssize_t write(int fd, void *buf, size_t count);

/*Set the offset where reading or writing will start.*/
off_t lseek(int fd, off_t offset, int whence);

/*Send a signal to the driver,
telling it to start the device.*/
int ioctl(int fd, int request, ...);
```

and to read the return values, respectively. The `lseek()` system call is used to select which register will be read or written.

The read and write system calls access the registers through the memory mapped region of memory which allows them to be accessed as normal memory. The buffer is a character array containing the data to write to the device in the case of the write system call, or to which the data is written in the case of the read system call. `Count` indicates the number of characters to read or write, which should be 8 for reading a single register. Because these system calls do not have an argument to specify which register to read or write, the `lseek()` system call is used first to set the correct index into the memory region.

The system calls check that the operation will not overflow the boundaries of the mapped memory region. If it does, the operation is truncated so only part of the data is read, or written. The return value indicates the number of bytes read or written. After each operation the position is incremented by this value.

3.1.1.3 ioctl

The `ioctl` system call is a general purpose system call, which can be used by the programmer in case none of the standard system calls provide a similar functionality. The call is used by first defining several commands which can be used with the call in a header file. This implementation supports only the `HTEX_IOEXECUTE` command. When the system call is called with this command as an argument it executes the code shown

in listing 3.2. First a reset command is sent to the device to ensure that the device will be in a correct state after any possible previous executions. Next the process calling the ioctl is put to sleep with a call to `wait_event_interruptible`. This functions checks to see if the condition is true, if it is execution continues, if not, the process is put to sleep and added to a wait_queue until it is restarted at a later time. This specific function is also interruptible, allowing the user to interrupt the process while the accelerator is still executing. This is useful when debugging the hardware design, since the accelerator can enter an infinite loop resulting in the application hanging indefinitely. If the wait was not interruptible the user would have to restart the computer in order to kill the process. When the interrupt handler resumes the execution of the process, the process continues its execution within the ioctl system call. The next step after waiting is a call to `release_all_pages` which traverses the array of mapped pages and releases them.

Listing 3.2: Implementation of the ioctl system call.

```
case HTEX_IOEXECUTE:
  iowrite32(HTEX_RESET_C, htex_dev->bar2->bar);
  iowrite32(HTEX_EXECUTE_C, htex_dev->bar2->bar);
  result = wait_event_interruptible(wait_queue, htex_dev->done
     != 0);
  release_all_pages(htex_dev);
  htex_dev->done = 0;
  if (result != 0)
    return -ERESTARTSYS;
  break;
```

3.1.2 Interrupts

Interrupt handlers should be as short as possible, otherwise the kernel will not be able to schedule other tasks to run while the interrupt is being handled. However in many cases an interrupt requires a significant amount of processing to be handled. To solve this issue, interrupts in Linux can be split in a *top half* and *bottom half*. The top half is the actual interrupt handler which is called by the kernel when the corresponding interrupt is raised. The top half then schedules the bottom half to be executed at a later time. This means that the interrupt may not be handled as fast as normally possible, but it allows the interrupt handler to perform more complex tasks. A second advantage to this is that the top half has certain restrictions on the functions that can be called from it. For instance functions that may be put to sleep while waiting for some event (for instance blocking memory allocation functions) may not be called.

There are two different techniques for implementing the bottom half: tasklets and workqueues.

- Tasklets are functions which run in interrupt context, so the same restrictions apply with regards to function calls. However they are scheduled at a later time than the top half and run with interrupts enabled. This allows tasklets to perform long tasks while still being scheduled quickly.

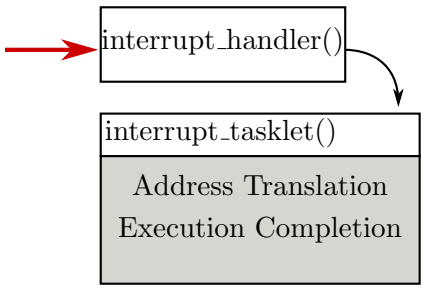


Figure 3.3: The interrupt handler schedules the interrupt tasklet to be executed at a later time. The interrupt tasklet then performs handles either the TLB miss, or execution completion.

- Workqueues are similar to tasklets, however they run in the context of a kernel process meaning that the restrictions on not being able to sleep are removed. The main disadvantage of workqueues is that they can be scheduled much later than tasklets. In this implementation workqueues are used to implement the bottom half, since the ability to sleep is needed for using `get_user_pages`. The `get_user_pages` function is explained in more detail in Subsection 3.1.2.1.

The driver receives interrupts from the FPGA device in two cases: on a TLB miss during address translation, and when the reconfigurable accelerator has completed execution. Since the High-Speed link interface supports only one interrupt identifier, we use a status device register to differentiate between the two cases above. The status register has different values for a TLB miss and for an execution completion interrupt.

3.1.2.1 Address Translation

During address translation at the reconfigurable device, a TLB miss may occur causing an interrupt. The device driver, then handles the interrupt providing the device with the missing TLB entry. During this process the driver should support *memory protection* and *memory paging*. The reconfigurable device inherits the memory access permissions of the program that called it. In order to support memory protection the driver ensures that the FPGA device receives new TLB entries only for pages that it has permissions to read or write; furthermore, read-only pages are protected from write accesses. In order to support memory paging the driver checks whether a page accessed by the FPGA device exists in the memory; in case a page is swapped out to disk, the driver loads it back to the memory. Finally, the driver maintains cache coherency after the completion of a hardware-accelerated function.

This functionality is implemented in the device driver using two kernel functions. The first function is `get_user_pages()` which provides memory protection and paging. In addition, `get_user_pages()` also locks all pages used by the device in memory, meaning they cannot be swapped back to disk. Because this function also ensures that the relevant page is in memory it requires the ability to sleep while waiting for the disk access to complete. The alternative to locking pages in memory is to update the FPGA TLB each time a useful page were swapped to disk. Locking useful pages to the memory does not

limit the total number of pages used by the device. The maximum number of pages locked by the device is equal to the number of FPGA TLB entries; it is however possible to use more pages than that by replacing -and hence unlocking- existing old TLB entries. The page returned by this function is stored in an array to keep track of all pages used by the device, in order to release the pages after execution completes. This page is also passed to the second functions, `pci_map_page()`, which returns the physical address of the page and also maintains cache coherency for the host processor.

3.1.2.2 Execution Completion

Upon execution completion of the reconfigurable acceleration the FPGA will raise an interrupt. The driver then wakes up the program that initiated the execution with a call to `wake_up_interruptible()`. This function resumes the execution of all processes in the wait_queue mentioned in section 3.1.1.3. However, before the program is allowed to continue executing, all the pages that were locked and mapped during execution must be unlocked to prevent memory from being lost to the kernel and to flush any old entries in the host-cache. To do so, the driver keeps track of all the FPGA-TLB entries, unlocks the corresponding pages and invalidates their cache entries[1] with calls to `page_cache_release` and `pci_unmap_page` respectively. Finally, the driver sends a reset command to the device to reset the reconfigurable device.

3.1.3 Initialization

The third task of the driver is to initialize the FPGA-device and the drivers internal data-structures. The driver exports an initialization function, and an exit function to the kernel. When the driver is first loaded into the kernel, the initialization function is called. Internally the Linux kernel treats HyperTransport devices the same as PCI devices. This allows us to use the same functions used to manage pci-devices to do the same with the htx-device. The init functions of the implemented driver calls a function called pci_register_driver. The argument of this function is a structure containing pointers to a string containing the driver name, a structure containing FPGA device-id, and 2 functions called probe and remove. The name is used to display to the user when listing all loaded modules. The device-id is used by the kernel to determine what driver should be used to handle a piece of hardware.

Each PCI device has 2 fields containing a vendor id and a device id. These should be unique for each device. When the kernel detects a PCI device, or in this case a HyperTransport device, with a vendor and device id matching those provided by a driver, the probe function of that driver is called. When the machine is shut down the kernel will call the remove function. The remove function can also be called if the device is hotpluggable, meaning it can be removed while the machine is running. This is not the case for the hardware used in this design.

The probe function is meant to initialize the device and set up certain structures so that the device can be used by user-space software. The three most important steps

[1] The FPGA device may write to a memory location, a cache line corresponding to this memory location should be invalidated before the program resumes execution.

performed by the initialization function are:

- first of all mapping the three memory regions of the device into the address space of the kernel, so they can be used to access resources on the device.
- The second step is to create a new interrupt identifier, and use it to register the interrupt handler with the kernel. This step also writes the address associated with this identifier to the device. This way, when the device writes to this address, the kernel knows what interrupt handler it belongs to.
- The last step is registering a character device with the kernel. This allows the creation of a device node in the `/dev/` directory through which user-space software can access the device.

The first step is accomplished by using the functions pci_resource_len, and pci_resource_start, and finally request_mem_region and ioremap_nocache to request and map the regions into the kernels address space. The second step is done using two functions ht_create_irq and request_irq. The first function creates a new interrupt identifier associated with the device. The second registers the interrupt handler with the interrupt number. The last step consists of initializing a structure with pointers to the implementations of the various system calls (open, close, read, write, lseek, ioctl), requesting a device number and registering the resulting structure with the kernel.

3.2 The Compiler

The programmer can indicate which functions are to be executed in hardware, by annotating them with GCC attributes: `__attribute__((user("replace")))`. The compiler will then automatically insert the appropriate system calls to the drivers API in order to achieve the hardware execution of the function. To do so, we have extended our GCC 4.5 compiler with a plug-in which scans the source code for such annotated functions. The body of such a function is then replaced with the required system calls. As depicted in the example of Fig 3.4, such system calls are the following: `open()`, `write()`, `ioctl()`, `read()`; the `close()` system call is called by the kernel when the program exits, hence not inserted by the compiler. As opposed to ISA extensions approaches, using system calls to control the reconfigurable device makes our approach generic. It is noteworthy that the function arguments passed to the FPGA device cannot be more than 64-bit numbers, since this is the size of each FPGA XREG. However, this is not a limitation; *supporting virtual addresses in the FPGA allows to pass as function-arguments pointers to larger data-structures in memory* which can then be accessed during the execution of the function with DMA transactions.

The plug-in is executed by GCC in four stages:

- An entry function is called when the plug-in is loaded before any compilation is done. The developed plug-in registers a function which when called registers a new attribute with the compiler. The plug-in also registers a new pass called the replace pass. This optimization pass is run by GCC after the code has been transformed

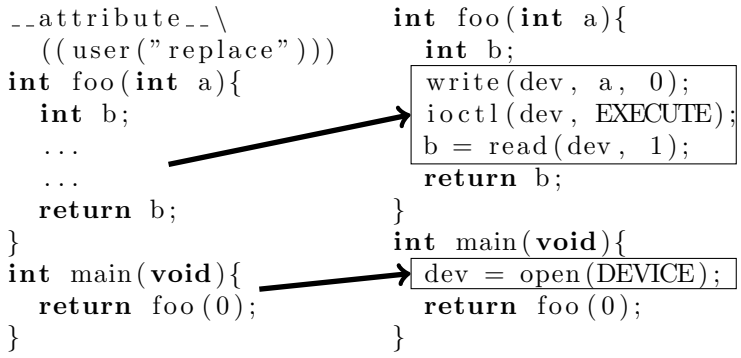


Figure 3.4: Code modification by the extended compiler.

into the internal representation, and replaces the function body with calls to the wrapper functions.

- Attribute registration. During this stage the function registered during the initialization of the plug-in is called. This function then registers a new attribute with the compiler which will be recognized during later compilation passes.
- During the compilation of the source file, the function registered along with the attribute is called every time the attribute occurs.
- The actual replacement pass is called once for every function in a source file after the code has been translated into the internal representation used by GCC.

3.2.1 Wrapper Functions

In order to simplify programming without compiler support and to reduce the complexity of the compiler plug-in, a library of simple wrapper functions is used. These functions can be easily used by the programmer, and also inserted by the compiler. Without wrapper functions, the compiler would have to insert error checking code, which is now simply part of the wrapper function. While this is certainly possible, using the wrapper functions greatly simplifies the replacement process. The prototypes of the wrapper functions are shown in listing 3.3. The complete implementation of the wrapper functions is shown in appendix A.

Listing 3.3: Function prototypes of wrapper functions inserted by GCC plug-in.

```
int htx_init();
uint64_t htx_read(uint64_t index);
void htx_write(uint64_t arg, uint64_t index);
void htx_execute();
```

The htx_read and htx_write functions are wrappers for the read and write system calls, and they are used to implement the reading and writing of exchange registers. Both have an argument called index which is the register number to access. Internally the lseek system call uses this this argument to select the register. Write takes an

additional argument, which is the value to write, as a 64-bit unsigned integer. Read returns its result as the same type. The htx_execute wrapper function implements the execute instruction using the ioctl system call. Finally, the htx_init wrapper function uses the open system call to lock the device for the current application. This initializes a global variable which stores the file descriptor returned by the open system call and is used by the other wrapper functions internally. There is no wrapper function which uses the close system call, since when the application exits the operating system will the detect the opened file descriptor and automatically call the close system call.

3.2.2 Attributes

After all plug-ins have been loaded, but before compilation starts, the register attributes function is called by GCC. This function creates a new attribute called *user*. This Attribute is used as follows `__attribute__((user(arguments)))`. The arguments allowed for this attribute are: `"write"`, `"read"`, `"execute"`, `"init"` and `"replace"`. The first 4 are used in the library of wrapper functions to tell the compiler which is which. This way the plug-in does not rely on hard coded names for the wrapper functions.

Whenever the compiler encounters an attribute of the type *user* a function defined in the plug-in is called. This function checks if the attribute is applied to a function or not, and if the attribute has an argument. If it does not, a warning is generated to indicate improper use of the attribute. Otherwise, if the argument matches either write, read, execute or init, the plug-in stores a pointer to the function declaration for later use in the replacement function. This is done to remove the need to create structures representing these functions during the replacement step. It also has the benefit of not depending on specific names for the wrapper functions.

3.2.3 Replacement

After the code has been transformed into the internal representation, the replacement function will be called once for every function in the source code. The replacement function checks whether the function for which it is called is marked with the replace attribute, or if it is the main function. In the first case the plug-in will remove the body of the function, and replace it with calls to the wrapper functions as shown in Figure 3.5. In the second case the plug-in inserts a call to the init function at the very start of the main.

When a function marked with `__attribute__((user("replace")))` the function body is completely removed. Next a call to the write system call is inserted once for each function argument. Because the exchange registers are 64 bit registers, the wrapper functions also expect 64 bit values. Because of this casts are inserted to cast any arguments to unsigned 64 bit values. This means that the programmer will have to ensure that all arguments fit within a 64 bit value. It also means that structs cannot be used as parameters directly, however pointers to structs can be used. This is not recommended however, since there is no way to guarantee how the compiler pads the struct to meet alignment restrictions.

Next a call to the execute wrapper function is inserted, which signals the accelerator to start, and puts the calling process to sleep until execution is completed. If the func-

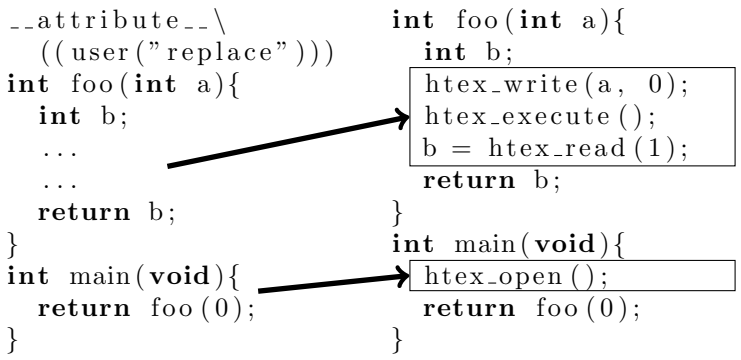


Figure 3.5: Function body replaced with calls to wrapper functions.

tion being accelerated has a return value, a call to the read wrapper function with the appropriate casts is inserted, and the result is returned. If there is no return value, a return statement with no argument is inserted.

3.3 The FPGA Device

The FPGA Device consists of three parts: the high-speed link interface, the Reconfigurable accelerator and its wrapper. The high-speed link interface abstracts the communication details of the bus allowing the rest of the FPGA device to send and receive data using a simpler protocol. The Reconfigurable accelerator is the hardware implementation of the accelerated function and has a standard interface to exchange data with the rest of the system. The wrapper is located between the high-speed link interface and the reconfigurable accelerator; it is used to provide all the necessary functionality for integrating the FPGA device with the rest of the system.

Figure 3.6 shows a block diagram of the wrapper. In order to support communication over the HyperTransport interface the wrapper contains a DMA unit and an IO read-/write unit. The DMA unit performs all the necessary DMA reads and writes required to provide data to the accelerator, while the IO read/write unit handles all host initiated communication. Additionally, we have the control unit which handles the commands received from the host through the IO unit and the Exchange Registers which are used to transfer data between host and accelerator without going through memory. The IH generates interrupts in case of an address translation request or execution completion of the accelerator, which are sent to the host through the DMA unit. In order to support the use of virtual addresses by the accelerator we have the Address Translation unit and TLB, which are used by the DMA unit to translate virtual addresses from the accelerator into physical addresses. Finally, the status unit is used to read various information from the wrapper such as the cause of an interrupt, or the address which is to be translated.

In the rest of this section, we describe in more detail the functionality of the wrapper. The wrapper is responsible for a number of tasks: it performs address translation, handles DMA transfers and interrupts, maintains the memory-mapped IO device registers and in general manages the interface between the device and its driver.

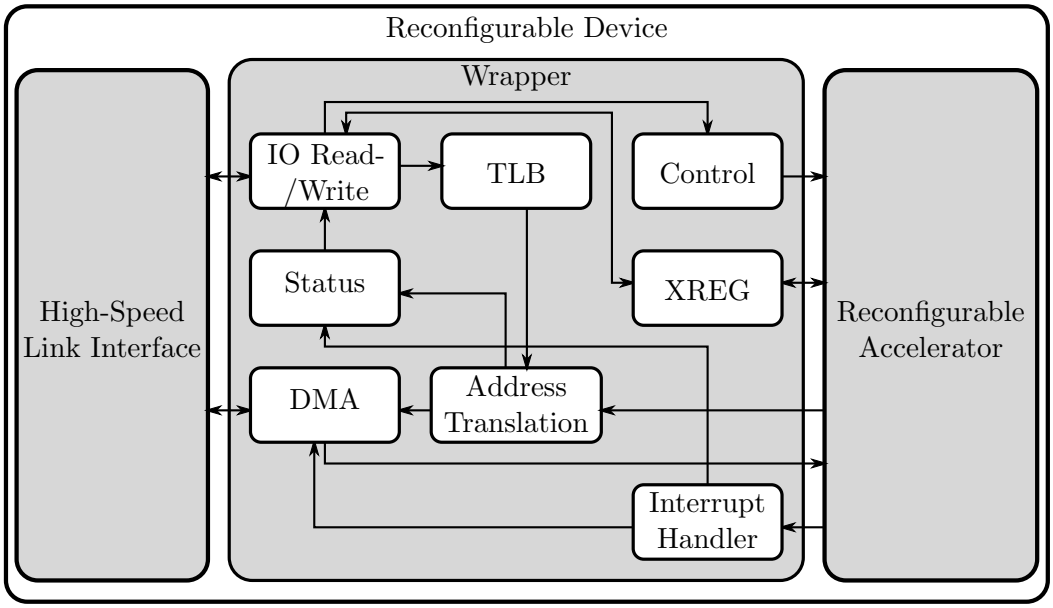


Figure 3.6: Block diagram of the wrapper.

3.3.1 Address Translation

The FPGA device works on the virtual address space. Consequently, in order to access memory the device needs to have address translation support. To do so, the virtual addresses are translated into physical addresses using a local TLB copy in the FPGA. TLB misses are handled by interrupts raised by the device. In such cases, the driver will write the missing entry to the FPGA TLB which then will be able to proceed with the respective translation. All pages stored in the FPGA TLB copy are locked in memory by the driver, while pages which are replaced in the TLB copy are unlocked. More details on virtual addresses in Linux is given in appendix B.

Address translation is handled by the Address Translation unit in Figure 3.6. Whenever an address must be translated, the address translation unit takes the following steps: First the input address, and in the case of a write the data, are stored in registers. Next, bits 20 to 12 (the page index) of the virtual address are used as an index into the TLB, to find the entry corresponding to the virtual address. The remaining most significant bits (63 to 21) are then compared to the tag stored in the TLB at the given index. If these bits match, and the valid bit is set to true, otherwise the TLB entry is invalid and an interrupt is generated to indicate to the driver that address translation is required. Once the correct entry is written to the TLB the address translation unit will append the lowest 12 bits of the virtual address (the page offset) to the physical address stored in the TLB.

The TLB is implemented as a BRAM with 512 entries of 96 bits. Of these 96 bits 40 are used to store the page number for the hardware address, 43 are used to store the tag for the virtual address. The remaining bits are used to store the valid flags, and any future flags such as read and write protection.

Figure 3.7 shows how the address translation unit uses the TLB to translate a virtual

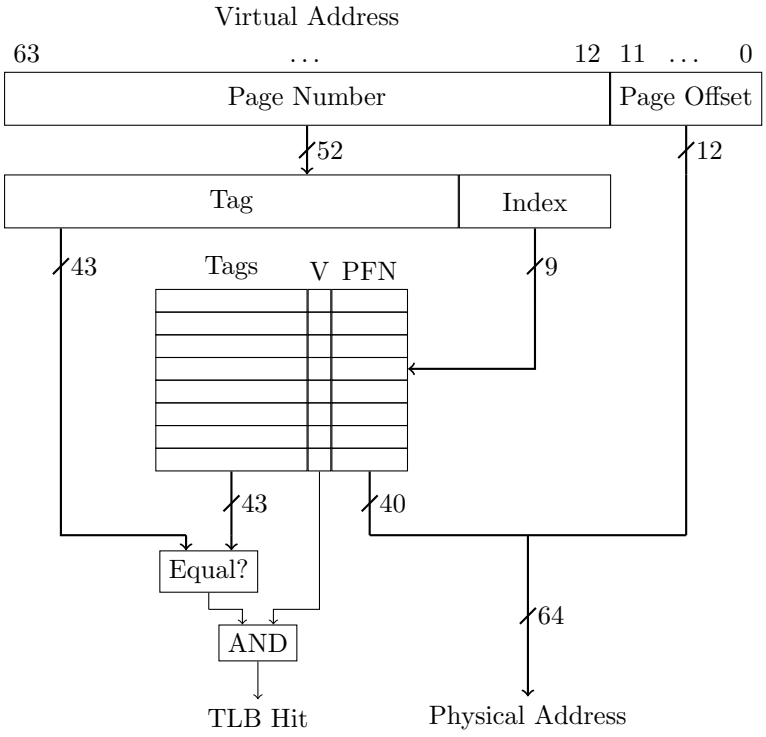


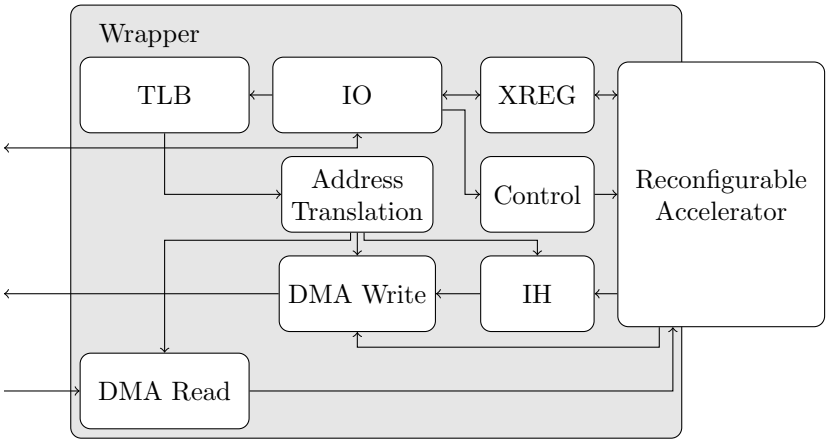
Figure 3.7: Diagram showing how the TLB is used to translate a virtual address to a physical address.

address to a physical address. The TLB works as follows: The 52 bit virtual page number is split into a 9 bit index and a 43 bit tag. The TLB also has a valid bit which indicates whether the entry at the given index has been set or not. When an address is translated the index is used to index the TLB array, and the tag stored in the TLB is compared to the one from the address being translated. If they are the same and the valid bit is set there is a hit in the TLB and the page number from the TLB is combined with the page offset to form the physical address used by the DMA units.

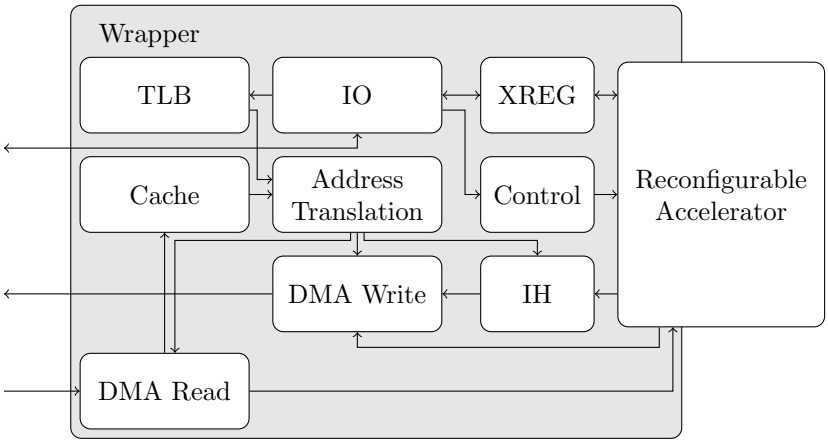
3.3.2 DMA: Direct Memory Accesses

The FPGA device accesses the memory using DMA reads and writes. The Reconfigurable accelerator places the memory access requests and subsequently these are handled by the DMA manager module located in the wrapper. We have designed three different versions of the DMA manager:

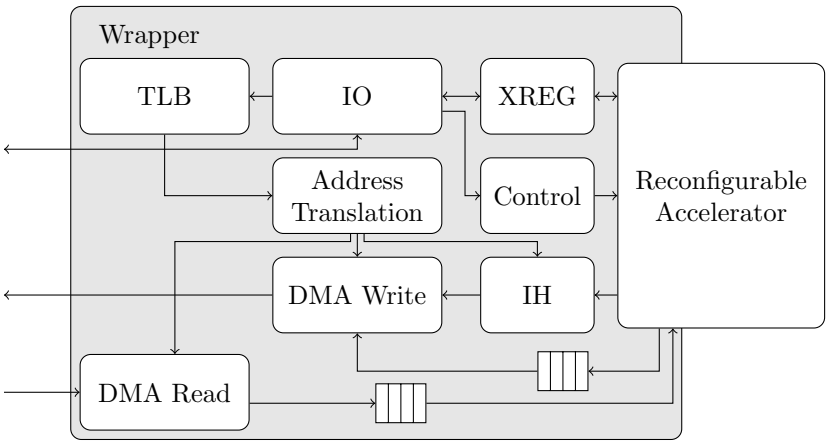
- A basic DMA manager, shown in figure 3.8a serves the accelerator requests sequentially; this means that there are no concurrent DMA accesses and in case of reads the data read from memory need to arrive to the FPGA device before a new request takes place. This first approach introduces significant overhead since after each memory access the accelerator needs to stall waiting for the access to be completed.



a: Basic DMA support.



b: DMA manager with cache.



c: DMA manager with queues.

Figure 3.8: Alternative designs for the accelerator wrapper, using different DMA managers.

- A DMA manager with a cache (Figure 3.8b), which requests 1 cache line instead of 64 bits for each read and stores it in the cache in an effort to reduce the number of DMA operations.

- A DMA manager with FIFO queues instead of a cache in order to exploit the regularity of memory accesses in some types of applications; e.g. streaming applications. In these types of applications, the DMA manager can pre-fetch data from memory. The read and write data are queued in the read and write FIFOs as depicted in Figure 3.8c. Multiple requests can be active at the same time while the reconfigurable accelerator continues processing.

3.3.2.1 Basic DMA

In the basic DMA hierarchy the memory read and write requests go through the address translation unit. If there is a TLB miss, an address translation request is sent to the host, and he address translation unit waits until the address is translation. Then there is a hit, the read or write request is sent to either the DMA Read unit or the DMA write unit. The accelerator does requests on 64 bit values, meaning that each read or write results in a either 64 bits being sent by the DMA write unit, or read by the DMA Read unit. As each request includes 96 bit command packet this greatly increases the overhead of memory accesses and thus reduces the usable link capacity.

When the DMA write unit receives a write command from the address translation unit it sends a write command and data packet on the posted queue, unless the queues are full in which case it waits. An acknowledgment signal is sent back to the accelerator indicating that the write has been handled and that the accelerator can continue execution. This is done to prevent the accelerator from writing data while the queues are full, or while there is a TLB miss, to prevent data loss.

When the DMA Read unit receives a read request from the address translation unit it sends a read command packet on the non-posted queue, and waits until it receives the response data and command packets on the response queue. Once it receives these it asserts an acknowledge signal to the accelerator indicating that the data is ready. This means that the reads are blocking, and only one read request can be outstanding at any one time. Because of this the latency of the memory accesses drastically reduces the performance of the accelerators, since most of the time is spent waiting for data to arrive.

3.3.2.2 DMA with Cache

In the second DMA hierarchy, in order to reduce the impact of the memory latency on the accelerator, a cache was added to the design. This way whenever there is a cache miss, an entire cache line of 512 bits is read from memory and stored in the cache. This reduces both the average memory latency, and the packetization overhead for consecutive read requests.

The implemented cache is a simple direct mapped cache with 64 cache lines. Each cache line stores both a 512 bit data entry and the associated tag, which identifies the

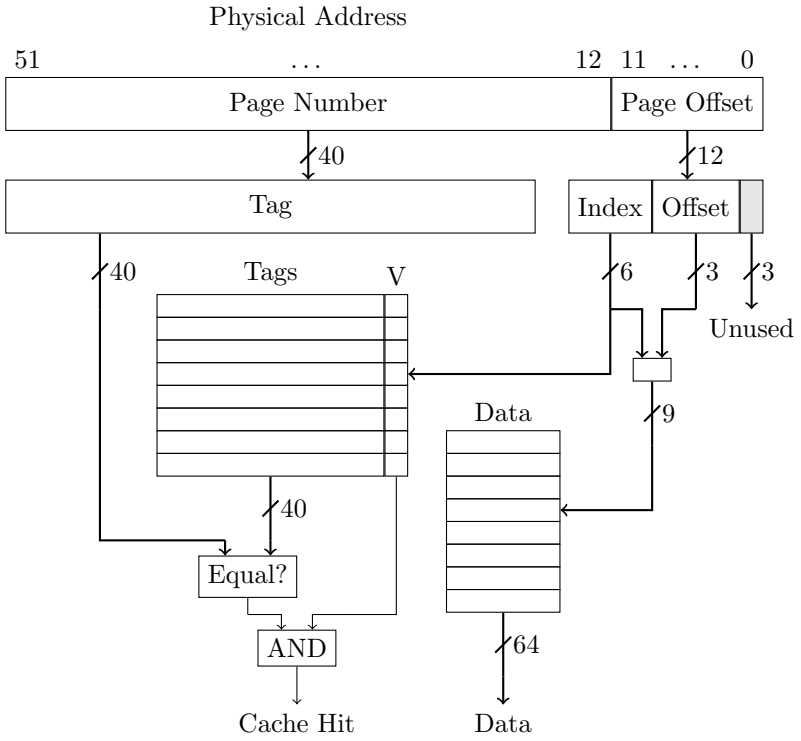


Figure 3.9: Implementation of the cache. The 6 most significant bits are used to index the tags and valid bits. These 9 most significant bits of the offset are used to index the the data. The 3 least significant bits address less than 64 bits and are ignored. In the cache the physical page number is used as tag, as opposed to the virtual page number in the TLB.

virtual page the data belongs to, and a valid flag. The cache is checked in the same cycle as the TLB. Figure 3.9 shows how the correct data is read from the cache.

Unlike the TLB, the cache uses the physical addresses as tags, however the lower 12 bits of the virtual address are used to index the cache, since they are the same for both the virtual and physical address. The page offset is split into two parts, the upper 6 bits are used to index the cache, and the lower 6 bits are used to index the word within a single cache line. The tag is used to determine weather the correct data was found at the given index.

If the tags for both the cache and TLB match, and both valid flags are true the accelerator reads from the cache, otherwise if there is a TLB miss, the address translation unit will generate an address translation request the same as in the basic design. If there is a TLB hit and a cache miss in the case of a read request, the read request is sent to the DMA read unit which reads the cache line from memory and writes it to the cache, at which point the accelerator is signaled that the data has arrived. If however the TLB hit and cache miss occur on a write request, the cache is ignored and the data is only written to main memory. This is done to avoid unnecessary reads in the case where the accelerator reads from one array and writes to another.

3.3.2.3 DMA with Queues

While the cache reduces the average memory latency significantly, and also reduces the packetization overhead, the design still does not use the full bandwidth of the HyperTransport link. In order to do this multiple outstanding requests must be sent. The HyperTransport protocol supports a maximum of 32 outstanding requests for reads. For posted writes (non-blocking writes) there is no real limit as there is no need for the DMA controller to send a response packet. In order to support this, and also reduce the time it takes to perform address translation a new hierarchy using queues is designed.

Read and write request from the accelerator no longer go through the address translation unit, but instead to separate read manager, and write manager units. Also the interface to the accelerator is altered. Several signals for managing the queues are added, and also signals indicating whether the manager units are ready. When the accelerator wants to read or write data it first sets the address and size signals to the starting address and the total number of 64 bit values being written or read, and asserts the read enable or write enable signal respectively. The corresponding manager unit then translates the starting address to the physical address through the address translation unit.

Once the address is translated the manager sends the request on to the either the DMA read or write unit. If the request from the accelerator crosses a page boundary, the manager reduces the size of the request to the DMA unit to assure that no page boundary is crossed. This is done because consecutive pages in virtual memory do not necessarily map to consecutive pages in physical memory. The manager subtracts this reduced size from total number of data elements, and adds it to the starting address. If the resulting size is larger than 0 the manager will start translating the address for the next page. Since a TLB miss takes a very large number of cycles (thousands of cycles) this hides some of the latency of the miss.

When the DMA read unit receives a request from its manager it checks if there are available tags, and if so it sends a read request using the next available tag and increments the tag counter, and decrements the total number of free tags. Because individual memory accesses cannot cross the 512 bit boundary, the DMA unit also checks if the request crosses this boundary and if so reduces the size of the data requested. Otherwise it request the maximum number of data elements (8 64 bit data elements) or the total number of requested elements, whichever is smaller.

When the DMA read unit receives a response command packet it stores the count field in that packet, and decrements it as the data packets arrive until all the data has arrived. Each data packet is pushed into the read FIFO, from which it will be read by the accelerator. Once all the data packets belonging to a single response command packet have arrived, the response packet is discarded and the number of free tags is incremented. This works only when ordered commands are used. If flags are set allowing packets to arrive out of order, a more complicated scheme for managing the message tags is needed.

In the case of the DMA write unit, things work very similarly except there is no need for tags, and for receiving data. When the DMA write unit receives a request, it waits until the write queue is filled with the number of data elements needed to fulfill the next write command. This number is determined in the same manner is in the DMA read unit. It depends on the total number of data elements, and whether or not request cross

the 512 bit boundary. The write command packet is only sent once all the data is in the write FIFO, since otherwise, if there is an error in the accelerator and not enough data is written to the FIFO, while the write request is already sent, the PC will hang while waiting for all the data to arrive.

3.3.3 Interrupts

The interrupt caused by the reconfigurable device, in case of TLB misses and execution completion, are supported in the Interrupt Handler (IH) in the wrapper. The interrupt manager is responsible for prioritizing multiple simultaneous interrupts, and indicating which of these caused an interrupt. Whenever one of these conditions occurs, the interrupt manager sets a flip-flop corresponding to the cause of the interrupt to one. This ensures that in the case of multiple simultaneous interrupts all interrupts are handled.

Next if there are multiple interrupts the interrupt manager selects the one with the highest priority. Execution completion has the highest priority, while address translation has the lowest. The IH then sets a status register which can be read by the host to a value indicating the cause of the interrupt. The manager also signals the DMA write unit that there is an interrupt request, and waits until the DMA write manager has sent the interrupt to the host. At this point the flip flop corresponding to the interrupt being handled is reset.

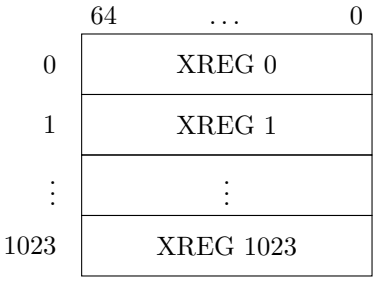
Next the interrupt manager waits until the host signals that the interrupt has been handled, before sending the next interrupt, if any. This ensures that a second interrupt will not automatically overwrite the value in the status register, which indicates the cause of an interrupt. The host signals that the interrupt is handled by writing a specific value to the command register, explained in Section 3.3.4. In the queued version of the design the execution completion interrupt is not sent unless the write queue is empty. This is done because the accelerator can finish execution while the write queue is still being emptied by the DMA write manager.

Interrupts are sent over HyperTransport by writing to a memory address. This address is specified by the driver during the initialization of the device. This means that the DMA write unit is also responsible for sending interrupts. There is a small difference in how interrupts are sent in design with queued DMA manager and the other two designs. In the Basic and Cached versions of the designs in the idle state of the DMA write unit there is a check to see if there is an interrupt, and if there is the interrupt command is sent.

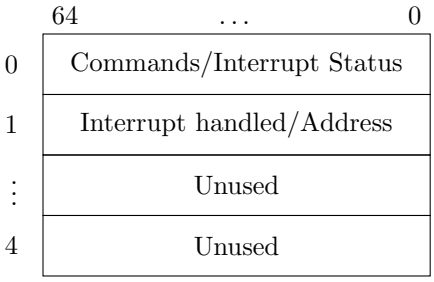
In the queued version however, this is not possible because the state machine can stay in the writing state for a long time while waiting to write data from the accelerator. This means that the interrupt would not be sent until all data elements in the request have been written. To prevent this, whenever there is an interrupt from the interrupt manager, the current state is stored, the interrupt is sent and the state machine returns to the previous state. The only case where the interrupt is not immediately sent, is when a write command packet has already been sent. In this case, all the data belonging to this packet should be sent before sending the interrupt.

3.3.4 Memory mapped IO

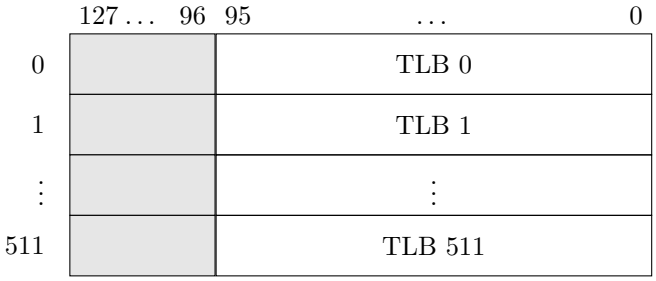
The host processor controls the FPGA device through the driver using three regions of memory mapped IO. The host uses these regions by mapping them into the kernel address space and accessing them as arrays of data. The FPGA device uses the IO regions as follows: The first region is mapped to the exchange registers (XREGs)(Figure 3.10a), which allow the driver to write function arguments to the device and read return values. The second region is used to send commands, such as *EXECUTE*, to the device and to get information about the status of the device such as the cause of an interrupt or the address to be translated (Figure 3.10b). In this region every address is mapped to a different function e.g. for sending commands, reading the cause of an interrupt, and reading the address to be translated. The third region is used for the driver to read and write directly to/from the TLB(Figure 3.10c).



a: Region 1 is mapped to the Exchange Registers.



b: The actions in region 2 depend on the address, and whether it is being written or read.



c: Region 3 is mapped to the TLB. Because the TLB is only 96 bits wide, the top 32 bits of every 128 bits (gray area) are ignored.

Figure 3.10: The three IO regions and how they are mapped to hardware.

When the host processor writes to one of these regions, a command and data packet arrive on the posted input queue. When the host processor reads from on of these regions, a command packet arrives on the non-posted input queue. A hardware component called the read-write unit handles these requests. It consists of a state machine which detect a command packet in either the posted or non-posted queue. In case a command and data

packet arrive on the posted queue, the state-machine checks what memory region the request belongs to and either writes the data to exchange registers or TLB in case of the first or third region, or in the case of the second region, it signals the control unit that there is a command. If instead there is a command packet on the non-posted queue, the device sends a response command packet accompanied by a data packet on the response queue. If the request was for the first or third memory region, the response data packet is read from the exchange registers or TLB respectively. If the request is for the second region however, the data is read from a set of status registers, which store information such as the cause of an interrupt and the address to be translated in case of a TLB miss. After the read and write request have been handled, they are removed from the queue.

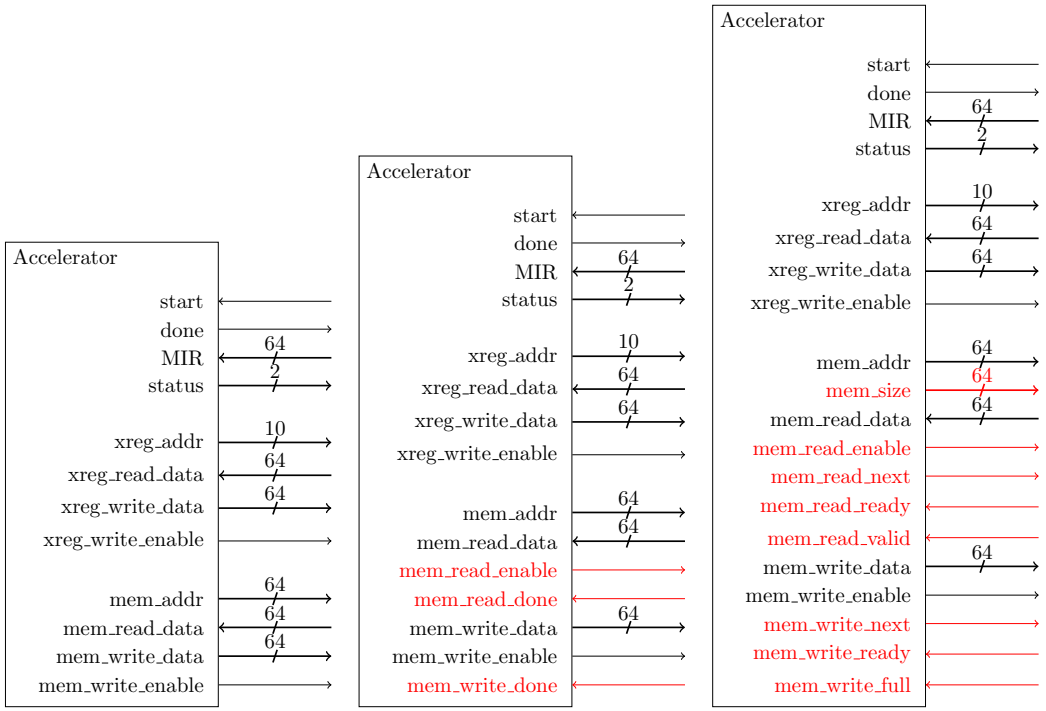
The control unit is also a state-machine which waits until it receives a command from the host processor. When it does it checks what address the command was written to. If the value is written to address 0, then the accompanying data packet is interpreted as one of three commands: *EXECUTE*, *RESET*, or *HANDLED*. In the case of a *RESET* command, the control unit asserts the reset signal for the reconfigurable accelerator, and the address translation unit. This is done to ensure that the device returns to a known state in case there was a mistake in the accelerator. In the case of an *EXECUTE* command the control unit asserts the start signal for the accelerator for 1 clock cycle, which tells the accelerator that it should start its execution. The *HANDLED* command is sent by the driver to indicate that an interrupt has been handled. The control unit then signals the IH, which can then send the next interrupt, or wait until the next interrupt event occurs. Writing to addresses other than 0 has no effect, and is reserved for adding partial reconfiguration support. After the command has been handled, or if the command was not recognized, the read-write unit is signaled that the command packet has been handled and can be removed from the queue.

3.3.5 Accelerator Interface

The interface from the wrapper to the reconfigurable accelerator is based on that of the MOLEN system(Figure 3.11a). The signals which make up the interface can be divided into three sub-interfaces.

- The control signals are the start and done signals, which tell the accelerator to start executing, and when execution has finished.
- The XREG signals are used by the accelerator to read from and write to the exchange registers. It contains the signals xreg_read_data, xreg_write_data, xreg_write_enable, xreg_address.
- The memory signals are used to access main memory from the accelerator. It consists of the signals mem_read_data, mem_write_data, mem_write_enable, mem_address.

For the implementation of the MOLEN over HyperTransport these signals stay mostly the same. The only difference is in the memory signals(Figure 3.11b). In the original MOLEN implementation the accelerator and GPP are both connected to main



a: Original accelerator interface in MOLEN.

b: Modified interface for use with basic and cache DMA hierarchies.

c: Accelerator interface modified for use with DMA with queues.

Figure 3.11: Three versions of the interface between wrapper and accelerator. The first is the original interface, the second is used for basic DMA, and for DMA with cache, while the third is used for DMA with queues. Signals which are new with respect to the original interface are indicated in red.

memory through an arbiter. Because of this memory reads always take 2 cycles to complete. When performing DMA reads over HyperTransport the latency will be much larger, and it is also not guaranteed to take a constant number of cycles because of the address translation. Because of this an extra signal called `mem_read_done` is added to the interface. This signal is set to 1 when the data from a read arrives.

Additionally, because it would be a waste of bandwidth to perform a DMA read on every clock cycle, a read enable(`mem_read_enable`) signal is also needed in the interface. Finally, because the queue with write commands can fill up a signal which indicates that a write has succeeded(`mem_write_done`) is also needed. This signal indicates that the write command has been sent to the HT-core. Otherwise if the posted queue is full, the accelerator will have to wait until the command can be sent before continuing with its execution. These three signals are all the changes that are needed for the basic DMA design and for the DMA with cache design.

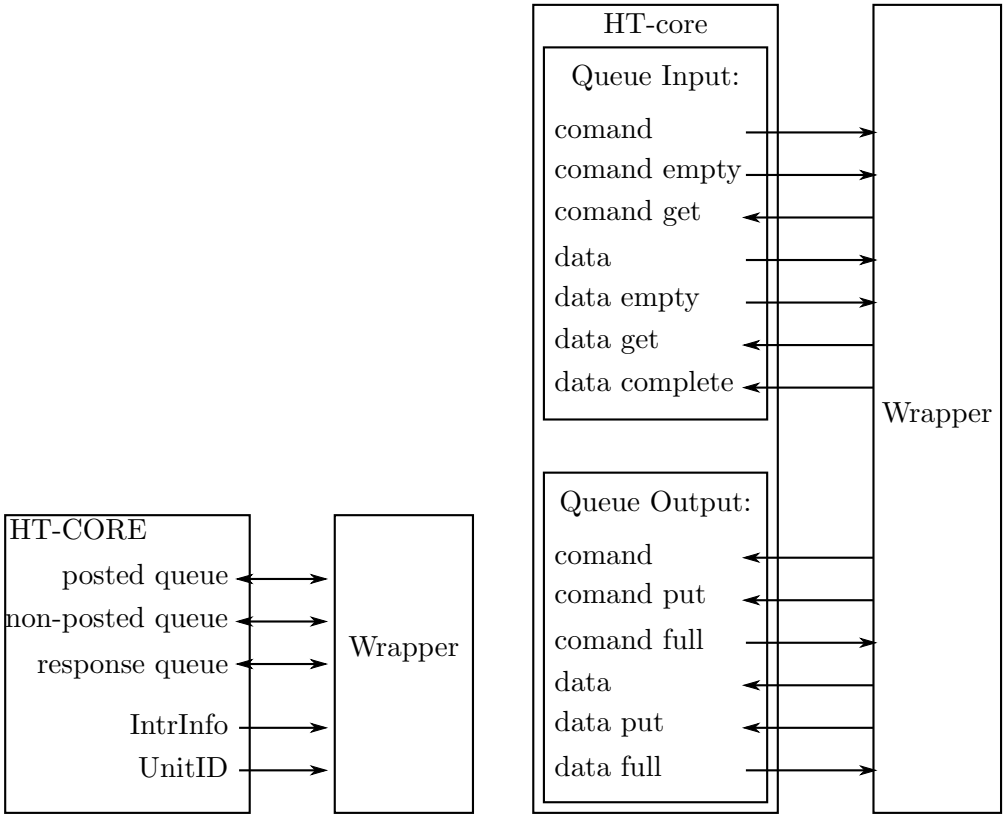
For the DMA design using queues a different interface is needed(Figure 3.11c). First of all the signals for read enable and write enable are now used to indicate to

the DMA managers that the accelerator will read or write `size` 64-bit data elements starting at the given address. This means that 2 additional signals, one to remove an item from the read queue(`mem_read_next`), and one to add an item to the write queue(`mem_write_next`) are needed. Additionally a signal indicating whether the read queue is empty(`mem_read_valid`) and a signal indicating whether the write queue is full(`mem_write_full`) are also needed. Finally the two signals `mem_read_ready` and `mem_write_ready` indicate whether the read and write manager are ready for a new request. These two signals replace the read complete and write complete signals from the basic and cached versions.

3.3.6 HyperTransport interface

The wrapper communicates with the HyperTransport core to perform any type of communication with the host system. The three different types of communication, DMA, PIO, and interrupts, all go through the HT-cores predefined interface. Figure 3.12 shows the interface between the wrapper and HyperTransport core. The interface consists of three "queues": the posted queue, the non-posted queue, and the response queue as seen in Figure 3.12a. The two additional signals, IntrInfo and UnitID are used when creating packets for sending over the HyperTransport bus. Figure 3.12b shows how each of these queues consists of an input and an output, and each of these again has different inputs and outputs for data and commands. The HyperTransport protocol uses commands to perform actions such as reading, writing, and receiving data, and the different queues are used for sending different commands. The posted queue is used for sending and receiving commands that do not require a response such as writes. The wrapper uses the posted queue for receiving commands from the driver in the form of memory mapped IO. The posted queue output is used for DMA writes and interrupts. The non-posted queue is used for sending and receiving commands that do expect a response such as reads, and also non-posted writes. The wrapper uses the non-posted queue input to receive read requests for the XREGs and the status registers. The non-posted output is used to send DMA read requests. The response queue is used to respond to read requests from the driver by sending the relevant data, and also to receive the data resulting from DMA read requests.

The HyperTransport command packets consist of various bit-fields as shown in Figure 3.13. The gray fields are not used by the wrapper and are always set to 0. The command packets all use the same basic layout of bit-fields, however the meaning and usage of different fields changes, depending on the command. The first bit-field (Cmd) specifies the type of command (read, write, posted, non-posted, response, etc.). The HyperTransport protocol defines many different commands, however the wrapper supports only a subset needed to have a working system. The next bit-field, UnitID, is used by the HyperTransport bus to determine at what device a command packet originated. Each device connected to the bus is assigned a unique UnitID during the initialization of the link. The wrapper gets this value from the HT-core as seen in figure 3.12a. The next field, SrcTag, is used to uniquely identify non-posted command packets. The tag is used to match a response packet to the original request packet; this way the HT devices can know when a specific request has been handled, especially in the case where packets



a: The three queues used to communicate between the HT-core and wrapper. And additional signals for used to supply the wrapper with information needed for making command packets and interrupts.

b: The signals for input and output in each queue.

Figure 3.12: The interface between HT-core and wrapper consists of three queues: the posted, non-posted and response queue. Each queue is made up of several more signals.

arrive out of order. In DMA write operations no response is required, and so the SrcTag can be set to 0. In response packets, the tag from the request packet is used for the SrcTag. The Count field determines the number of 32-bit data elements that will be sent along with the command packet. Count is the number of elements minus one, so 0 means a 32-bit data packet, while 15 means the maximum of 512 bits.

The address field is set a physical address in main memory in the case of a DMA read or write. The two least significant bits are not sent, since the values being read are 32-bits. HyperTransport also supports reading 8-bit values, however this is not supported by the wrapper. When doing this the Count field is used as a mask to determine what bytes within the 32-bit word are to be read or written. In order to send an interrupt, the wrapper must write to a certain address in memory. The packet format for this is

byte	7	6	5	4	3	2	1	0
0	SeqID[3:2]			Cmd[5:0]				
1	PassPW	SeqID[1:0]		UnitID[4:0]				
2	Count[1:0]		Compat	SrcTag[4:0]				
3	Addr[7:2]						Count[3:2]	
4	Addr[15:8]							
5	Addr[23:16]							
6	Addr[31:24]							
7	Addr[39:32]							
8	Addr[47:40]							
9	Addr[55:48]							
10	Addr[63:56]							
11	Ext_F	W_cmd	D_att	B_type		Reserved		

Figure 3.13: A HyperTransport request command packet split up into the various bit fields. In a response command packet the layout is the same apart from the address field, which is not used in that case.

shown in figure 3.14. Instead of an address, the IntrInfo value supplied by the HT-core is used, and split over the command and data packet. The Addr[39:32] is always set to FD. When sending a response packet the address field is left empty, since the SrcTag already indicates what request the response corresponds to.

Finally in the last byte of the command packet there are several flags. The B_type field determines which of the three memory mapped regions being addressed, and it is used only used when receiving commands from the host. The D_att field is set to 1 if there is data accompanying the packet, as is the case for writes, interrupts, and read response packets. Finally the Ext_F and W_cmd fields determine the size of the command packet. If both are 0 the size is 32-bits, W_cmd is 1 the size is 64 bits, and if Ext_F is 1 the command is 96 bits. The case where both are 1 is not used. The wrapper uses 96 bit command packets in order to support the full physical address space.

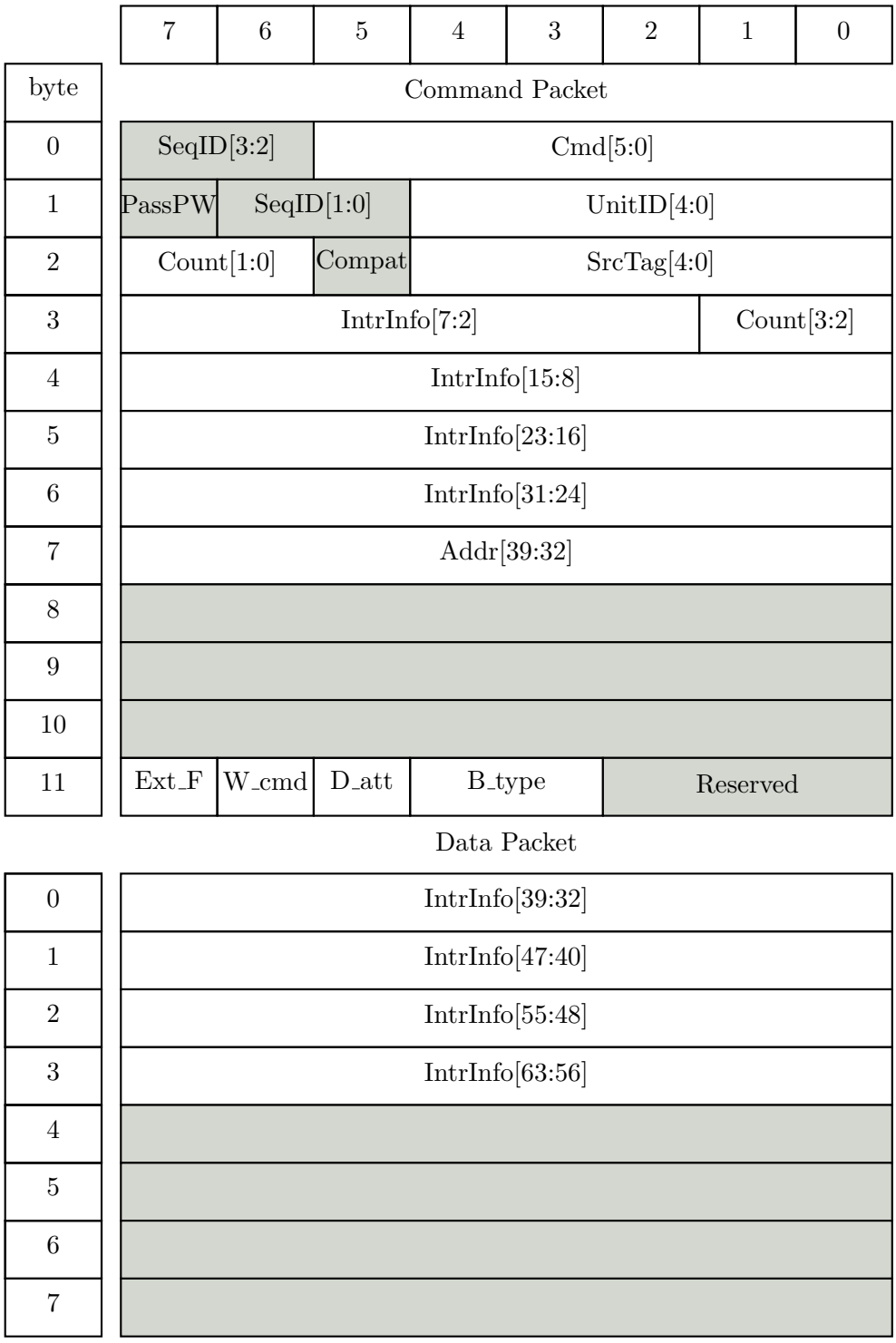


Figure 3.14: The packets sent by the wrapper in order to trigger an interrupt, divided into fields. Gray areas indicates fields which are not used and set to 0.

3.4 Conclusion

In this section we have presented the device driver, compiler plug-in and the wrapper. We explained how the device driver interfaces with software through the API, and how it supports memory protection, paging and cache coherency through the use of interrupts. Next we discussed how the compiler plug-in replaces the function body of annotated functions with calls to the drivers API through the use of a custom attribute. Finally we explained the implementation of the hardware wrapper and how it supports three types of communication: host to FPGA communication through the memory mapped regions, FPGA to host through interrupts, and finally FPGA to memory through DMA. The memory mapped regions are used to allow the host access to the XREGS, TLB, and control and status registers while interrupts are used to implement both execution completion and address translation, needed for supporting virtual memory. Also three different approaches for optimizing DMA access were shown, one which maps each memory access from the accelerator to a DMA operation, one which adds a Cache to the system to reduce DMA operations, and finally one which is optimized for streaming applications which access memory sequentially.

4 Implementation Platform

We implemented the generic System Architecture proposed in Chapter 3 on to a platform providing a reconfigurable device and a GPP, connected with a high-speed link. The platform runs an unmodified version of the Linux kernel, and also the compiler plugin. A second PC, connected by a USB cable to the reconfigurable device, is used to generate bitstreams and program the device. The implementation platform, used to instantiate the proposed system, is shown in Figure 4.1.

It consists of an AMD Opteron-244 1.8GHz 64-bit host processor and a 1-GByte DDR memory at an IWILL DK8-HTX motherboard. The motherboard provides a HyperTransport bus[9] and an HTX connector which we use to connect an HTX FPGA board[22]. The HTX FPGA board has a Xilinx Virtex4-100 FPGA device. For the FPGA interface with the HyperTransport bus we used the HTX module developed in [23] which supports 3.2 GBytes/sec throughput (unidirectional, 16-bit link width). The HTX interface allows data transfers of 32 to 512 bit data while command packets are 96-bits. Up to 32 concurrent DMA requests are possible and memory accesses need to be cache-aligned. Finally, the system runs on Linux kernel 2.6.30. It should be noted that our approach is generic and hence can be applied to other platforms, such as Intel's QuickPath[8] with slight modifications to the implementation.

In this chapter we describe the implemented test setup. In Section 4.1 we describe

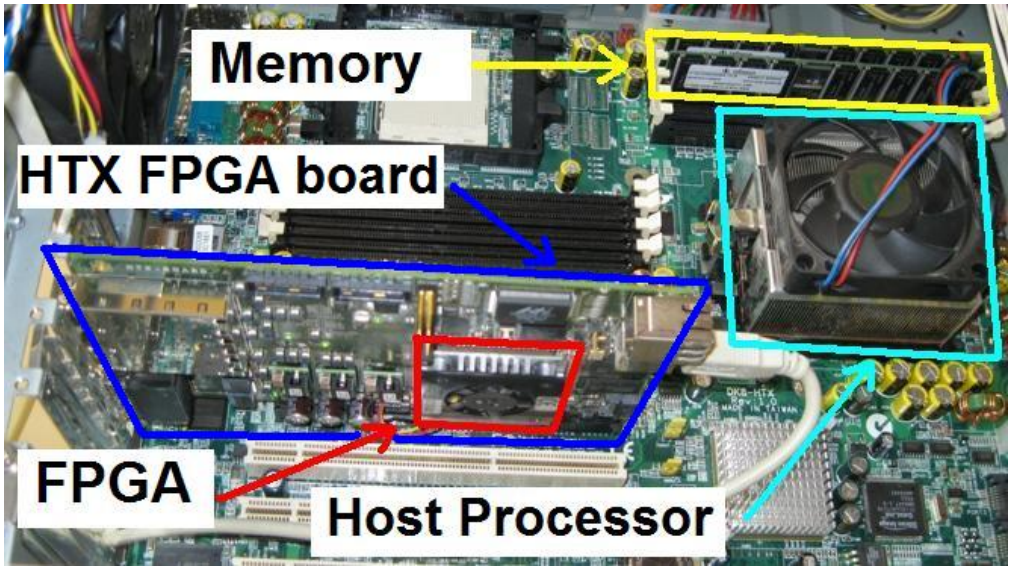


Figure 4.1: A picture of the implemented platform showing the various components of the system.

briefly the HyperTransport bus, the HyperTransport interface used to facilitate communication by the wrapper over HyperTransport, and the FPGA board with Virtex4 FPGA and HTX connector. Next in Section 4.2 we walk step by step through the execution of an accelerated application on the test setup.

4.1 Implementation Setup

As described in Chapter 3, the reconfigurable platform requires a High-Speed interconnect between the FPGA, the host processor and main memory. In our implementation we used the HyperTransport bus for this purpose. HyperTransport is a bus designed by AMD to provide high speed communication between processors, peripherals and main memory. HyperTransport supports the connecting of peripherals through the HTX connector. In order to use HyperTransport in the reconfigurable computing platform we additionally need an FPGA board which can be connected to the HTX connector, and we need an interface or controller to initialize the HT link during startup. The university of Mannheim has developed both a development board with an HTX connector, and a HyperTransport interface for use with this board.

4.1.1 Hyper-Transport

HyperTransport is a high speed bus developed by AMD to address the problem of the lag between processing power and bus performance. Every 18 months the number of transistors on a chip doubles, however the different buss architectures do not double in performance in this same time. This causes a gap between IO-bandwidth and processing speed. Additionally new technologies such as high resolutions, high speed networking and multi-core systems require an ever increasing amount of bandwidth. HyperTransport was developed to address these and other problems. HyperTransport uses a packet based protocol to establish point to point uni- and bidirectional connections between devices within the same system. It provides high speed, low latency communication between processors and peripherals and main memory. HyperTransport is also software compatible with PCI, reducing the cost of developing software for it.

In Figure 4.2 we see four kinds of HyperTransport devices which can be connected in different topologies: hosts, caves, tunnels, and bridges. Each topology must have at least one host which configures and maintains the HyperTransport connection, and one cave or endpoint. Multiple HT devices can be connected in a daisy chain by using tunnels. Finally bridges connect the HyperTransport link to other types of interconnect such as PCI, PCI express, and AGP.

Communication between these devices is done in packets,with sizes that are multiples of four bytes. The command packets are kept small, between 4 and 12 bytes, while data packets can be between 4 and 64 bytes (Figure 4.3) to reduce the overhead.

The host is required to maintain the HyperTransport connection between the various devices. During startup the host sets up the connection between all the devices, assigning identifiers to each of these, and using the highest link frequency supported by all devices.

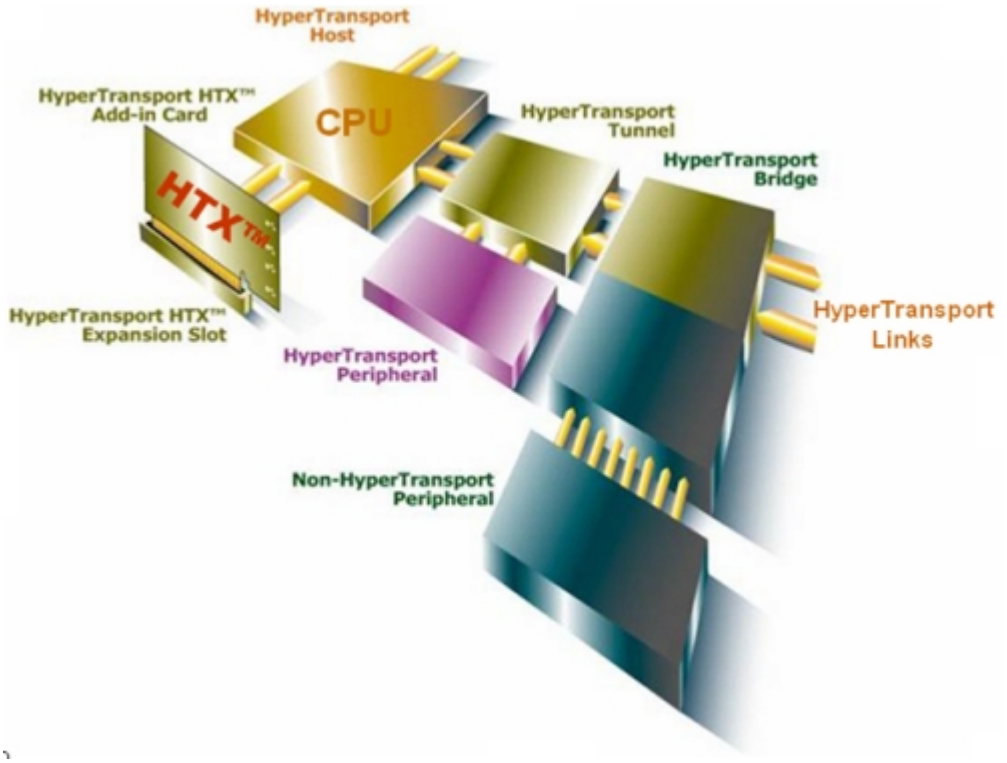


Figure 4.2: HyperTransport topology with the host connected to a cave in the form of the add-in card, and to a tunnel. The tunnel connects the host to a peripheral and a bridge, which in turn connects the host to a peripheral on a different type of interconnect. Source: HyperTransport Consortium website[9]

Figure 4.3: HyperTransport command packet and optional data packet.

4.1.2 HyperTransport FPGA Interface IP Core

The HyperTransport FPGA Interface is a verilog based hardware design for a HyperTransport core, developed for use on Virtex4 and Virtex5 chips. It supports link widths of up to 16 bits at a frequency of 400MHz, giving a maximum unidirectional throughput of 1.6GB/s. The core also handles the initialization of the links and the low level communication details of the HyperTransport bus. This core presents the user with a queue based interface to the HT-bus, which can be used to communicate with the rest of the system. The core is developed by the university of Mannheim.

The core presents an interface consisting of 3 queues to the user. These queues are the posted, non-posted and response queues. The posted queue is used for messages which don't expect a reply, such as writes, whereas the non-posted queue is used for messages which do expect a response, such as read requests. The response queue is used

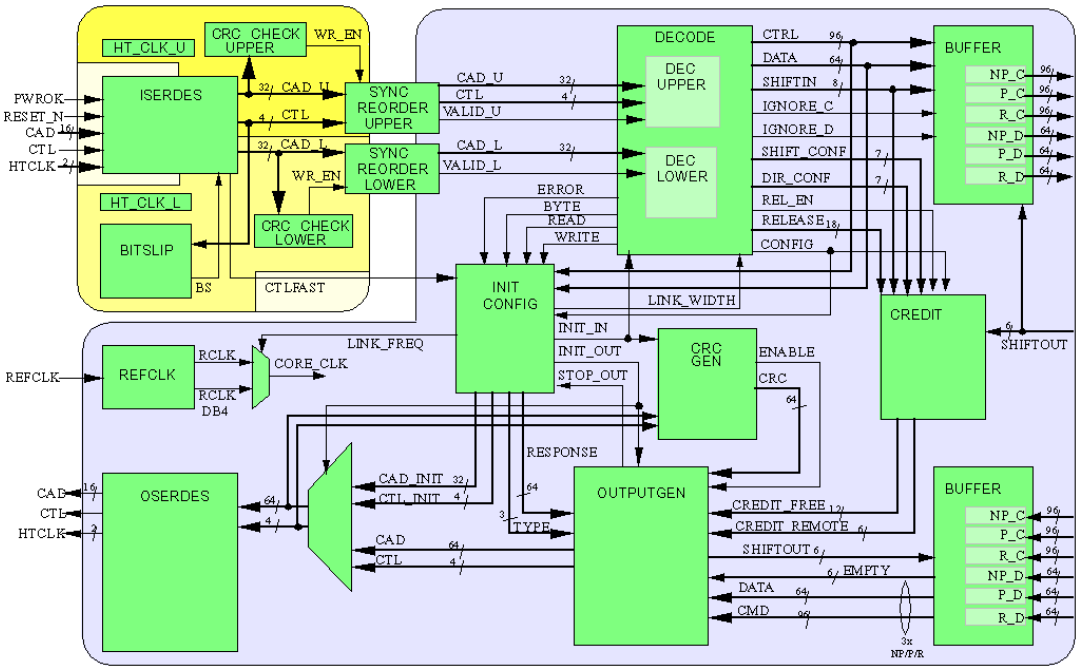


Figure 4.4: HyperTransport core block diagram. Source: HyperTransport core website[24].

for the responses to messages sent over the non-posted queue. Each queue has both an input and an output, meaning messages can be both sent and received on each queue. Here the input queue is used for data coming into the FPGA, while the output queue is used to send data out of the FPGA. The input queue has the following signal interface: command, data, command_get, data_get, data_complete, command_empty, data_empty. The output queue has the following signal interface: command, data, command_put, data_put, command_full, data_full.

Figure 4.4 shows the different functional units that make up the HT-core. On the left are the input and output signals connected to the HyperTransport bus. The input signals are first serialized, because the HyperTransport bus operates at a higher frequency than the FPGA. The packets are then decoded and placed in the appropriate queues, where the wrapper can read these. The init config unit is involved in initializing the HyperTransport link on system startup. On the right side of the figure we can see the signals connecting to the wrapper. The top buffer is were packets received from the host are stored, while the bottom buffer is where packets sent by the wrapper are stored before they are sent over the bus.

4.1.3 FPGA board with HTX Interface

The HTX-board is a prototyping platform developed by the university of Mannheim. It features a Virtex4 FPGA, both a 100MHz and 200Mhz clock, and an HTX connector, which allows it to be used to implement HyperTransport based expansion cards. The

FPGA can be programmed directly either using a USB cable or using a J-TAG programming cable. It is also possible to program the PROM, which then programs the FPGA whenever the board is powered on.

4.2 Example

In this section we describe the execution of a simple example program using the reconfigurable accelerator on this platform. The application in question copies the values from on array into another, and this functionality is implemented as an accelerator. The code for the application is shown in listing 4.1. This application is a slightly modified version of an example for the Molen platform. The program allocates two arrays and initializes them, and next copies the values from the first array to the second array. The copy function is accelerated in hardware. In the original example the copy function takes two arguments: one for the input array and one for the output array. In order to better test the address translation, cache and queues various sizes of arrays are needed. To avoid re-synthesizing the accelerator, we modified it to receive a third parameter, namely the number of data elements to copy from the input array to the output array.

4.2.1 Running an Experiment

In order to run the experiment the following starting point is required: On a computer used for synthesis an programming the FPGA: An ISE project containing the files for the ht-core, the wrapper and the accelerator designed for the experiment. On the test platform the code for the compiler, driver and the program begin accelerated is needed. The assumed directory structure is:

Running the complete experiment consists of several steps. The first of these is to synthesize the code for the reconfigurable device and create a bitstream using ISE. The device is then programmed from the second PC with this bitstream. After programming the device, the test platform must be reset by pressing the reset button. This ensures that the new device is correctly detected, and the HT link is reestablished. This is necessary because there is no support for partial reconfiguration, which means that the High-Speed link interface is also reprogrammed.

After the test platform starts up we load the driver and create the corresponding device nodes in the `/dev/` directory using the `load_driver` script. We can now compile the test application with the GCC 4.5 and the plugin to create a hardware accelerated application using the command: `gcc --plugin=/home/ce/plugin/plugin.o copy.c`. The result is a binary which when run uses the reconfigurable device to copy the data from one array to the other. The equivalent code of this binary is shown in listing 4.2, where we see the calls inserted by the plugin as explained in Section 3.2.

4.2.1.1 Starting Execution

When the program is started, the first thing it does is lock the device using the htx_init wrapper function, inserted by the compiler plugin. The application continues execution normally until it reaches the call to the accelerated function. The function takes three

Listing 4.1: The C code for the copy example program. Two arrays are allocated and initialized; next the values in the first array are copied to the second.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include "../molen_htx.h"

__attribute__((user("replace")))
void copy(long *a, long *b, long count)
{
        long i;
        for (i = 0; i < count; i++)
        {
                b[i] = a[i];
        }
}

int main (int argc, char **argv)
{
        long *A, *B, count = 0;
        int seconds, useconds, i;
        double time;
        struct timeval start, end;
        if(argc > 1)
                count = atoi(argv[1]);
        if(count <= 0 )
                count = 8;
        A = malloc(sizeof(long)*count);
        B = malloc(sizeof(long)*count);
        for (i=0;i<count;i++)
        {
                A[i]=i+1;
                B[i] = 0x123456789ABCDEF0;
        }
        gettimeofday(&start, 0);
        copy(A, B, count);
        gettimeofday(&end,0);
        seconds = end.tv_sec - start.tv_sec;
        useconds = end.tv_usec - start.tv_usec;
        time = seconds + ((double)useconds/1000000);
        printf("time in seconds: %f\n",time);
        return 0;
}
```

Listing 4.2: The code with calls to the wrapper functions inserted by the compiler plugin.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include "../molen_htx.h"

void copy(long *a, long *b, long count)
{
        htx_write(a, 0);
        htx_write(b, 1);
        htx_write(count, 2);
        htx_execute();
        return;
}

int main (int argc, char **argv)
{
        long *A, *B, count = 0;
        int seconds, useconds, i;
        double time;
        struct timeval start, end;
        htx_init();
        if(argc > 1)
                count = atoi(argv[1]);
        if(count <= 0 )
                count = 8;
        A = malloc(sizeof(long)*count);
        B = malloc(sizeof(long)*count);

        ...

        return 0;
}
```

arguments: a pointer to the array of input values, and a pointer to the array where these values will be copied, and an integer for the number of data elements to be copied. The function first writes these parameters to the XREGs through successive calls to the htx_write wrapper function, also inserted by the compiler plugin. The wrapper function writes the values to the XREGs through the memory mapped array, using the write system call.

The signal changes resulting from writing to the XREGs can be seen in figure 4.5. Firs the *posted command empty* and *posted data empty* signals go from 1 to 0, indicating that a command and data packet have been received on the posted queue. The B_type

field is checked to determine whether the write is destined for the XREGs, TLB or is a command. The address in the command field is then used to write the data to the correct XREG. Finally the posted command get, posted data get, and posted data complete signals are set to 1 for 1 cycle, indicating to the HT-core that these packets have been handled and that all data associated with the command packet has been received. The signal transition shown in Figure 4.5 are for writing three values to the XREGs, assuming that there is no delay between the packets.

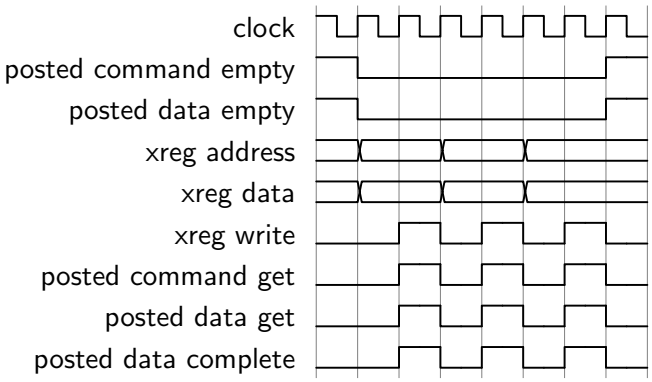


Figure 4.5: Signal transitions when writing arguments to the XREGs.

Next the accelerated function tells the hardware to start execution by calling the ioctl system call. The ioctl system call checks the argument to determine what to do, and writes the execute command to the second memory mapped array. In hardware this again results in the signals posted command empty and posted data empty to go from 1 to 0. The data is sent to the control unit after it is determined that the data was sent to the second memory mapped array. The control unit checks the data to see what command was sent, and as a result asserts the start signal for 1 cycle. The accelerator detects this and starts executing. The accelerator first reads the addresses of the input and output arrays, and the number of data elements from the XREGs. Next the accelerator performs the first read from memory from the lowest address of the input array. This is done in two different ways depending on whether the hierarchy with queues is used or not.

4.2.1.2 DMA read

In the basic and cache hierarchies a read from memory is performed by setting the address signal to the address of the array, and asserting the read request signal for 1 cycle. The address translation unit detects this and checks the TLB in order to translate the address. Since this is the first DMA request the TLB is still empty, and this causes the address translation unit to assert the address translation request signal (Figure 4.6). This is in turn detected by the interrupt handler unit, which sets the status register for the cause of interrupt to address translation, and signals the DMA write unit to send an interrupt. The DMA write unit sends the interrupt by writing a command and data packet as described in Section 3.3.6 by asserting the posted command put and posted data put signals for 1 cycle.

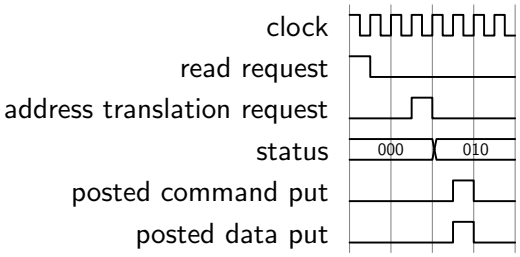


Figure 4.6: Signal transitions an interrupt caused by a TLB miss while performing a DMA read.

In software this results in the interrupt handler being called, which in turn schedules the execution of the interrupt handler work queue. The work queue then reads the status register, and determines that there is an address translation request. Next the address is also read from the device and is then translated as explained in Section 3.3.1, and the result is written to the TLB together with the tag and valid bit. Reading data from the device is shown in Figure 4.7. The nonposted command empty signals is de-asserted, indicating a new read request. The address in the command packet is used to read from either the XREGs, or in this case the status registers. In the next cycle the data is sent along with a response command by asserting the response command put, and response data put signals. At the same time the nonposted data get signal is asserted to remove the request command from the queue. This is repeated for every read request. Once the hardware determines that the physical address is written to the TLB, the address translation unit signals the DMA read unit to perform a read from the address obtained from the TLB.

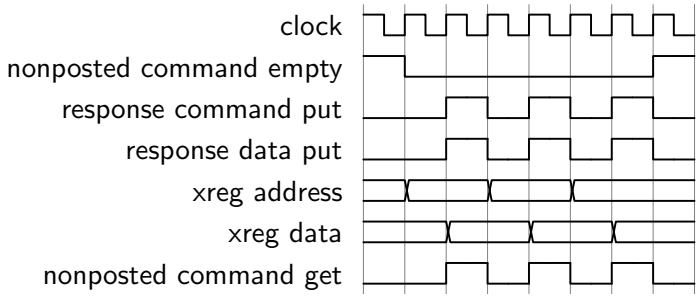


Figure 4.7: Signal transitions resulting from the host reading from the device.

In the basic DMA hierarchy, after the address is translated by the address translation unit, the DMA read unit sends a read request for 8 bytes from memory. After a number of cycles the data is received and the accelerator is notified by asserting the data valid signal for one cycle, indicating that it can continue execution. This is shown in Figure 4.8, where it is assumed that there was a TLB hit. In case of a TLB miss there would be a larger number of cycles between asserting the read request signal and the nonposted put signal, caused the driver having to update the TLB. In the figure we show that four cycles after the read request signal is asserted, the DMA read unit asserts the nonposted command put signal, sending a read request packet over HyperTransport. After an

additional 50 cycles the data is received, and the data valid signal is asserted for 1 cycle.

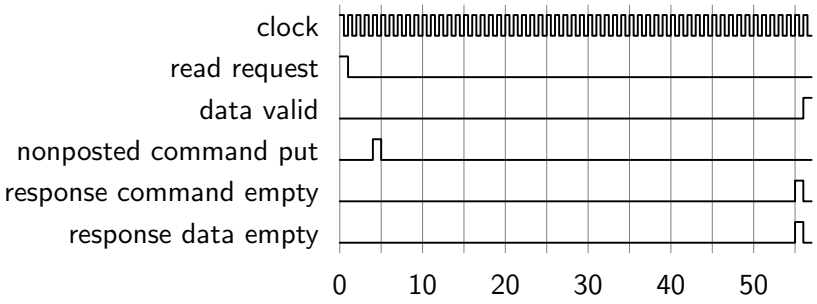


Figure 4.8: Signal transitions when performing a DMA Read request in the basic DMA hierarchy.

In the case of the hierarchy with a cache, 64 bytes are requested instead of 8 whenever there is a cache miss. When there is a cache hit the number of cycles required to read the data is reduced to the number of cycles required for checking the TLB and cache. Once all data packets have been received, the accelerator is notified by asserting the data valid signal for 1 cycle. This is shown in Figure 4.9 where it is again assumed there was a TLB hit. The first read results in a cache miss, causing the DMA read unit to send the read request for 64 bytes of data. After a latency of 50 cycles, data starts arriving and is written to the cache. Once all data has arrived, the data valid signal is asserted for 1 cycle. The result of a second consecutive read is also shown. Here the data valid is asserted after 4 cycles, since there is a TLB hit.

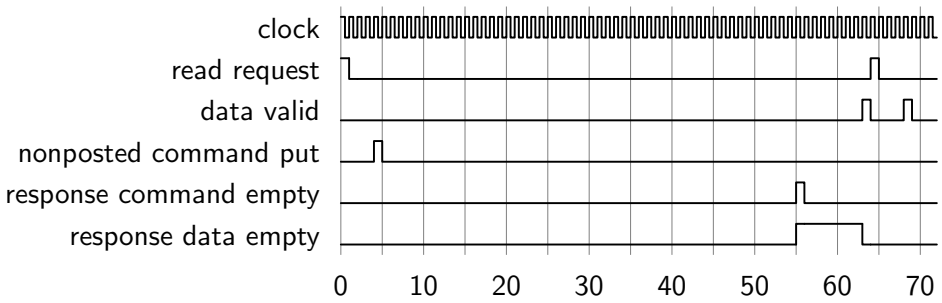


Figure 4.9: Signal transitions when performing a DMA Read in the memory hierarchy with a cache.

The read request is performed differently in the hierarchy fit queues. Here the accelerator sets the size signal to the number of 64 bit values to be read, and the address to the address for the input array, and asserts the read request signal for 1 cycle. The read manager unit then sends the address to the address translation unit to be translated before sending the request on to the DMA read unit. The read manager splits the request into page aligned request before sending these to the DMA read unit. The DMA read unit then starts sending read request aligned to 64 bytes. Unlike in the other two hierarchies, this one will send multiple requests until the maximum of 32 outstanding requests have been sent, or until all data has been requested. Once one of the outstand-

ing requests has been handled, more requests can be sent if necessary. The result is that after the initial delay, the device can receive new data almost every cycle. The received data is put into a queue, and the accelerator is notified that the input queue is no longer empty. We show this in Figure 4.10, where we show what would happen if the accelerator has to read 16 64-bit values. After the read request is asserted, and the address translated the DMA read unit start sending read requests for 64 bytes per request in this example. The delay here is larger than in the other two hierarchies because the request goes through both the DMA read manager and address translation unit before reaching the DMA read unit. After the initial delay of 50 cycles data starts arriving every cycle, and is inserted into the queue. After the initial data has propagated through the queue, the data valid signal is asserted, and the accelerator can start reading values from the queue by asserting the data get signal. This way the wrapper can prefetch data without the accelerator having to wait.

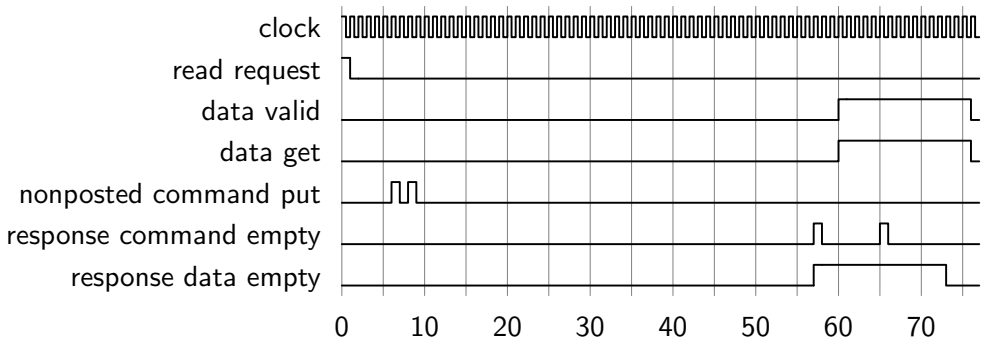


Figure 4.10: Signal transitions when performing a read request in the DMA hierarchy with queues.

4.2.1.3 DMA write

At this point the accelerator has received its first input data from the input array, and must write it to the output array. As shown in Figure 4.11, in the basic hierarchy, and the hierarchy with a cache, it does this by setting the data signal to the data received from the DMA units, setting the address to the output arrays address, and asserting the write request signal for 1 cycle. The address translation unit again performs address translation; however, assuming the output address is in the same page as the input address, there is no TLB miss. The DMA write unit then sends the write command packet along with the data. In the cached hierarchy, the address translation unit also checks if the cache line containing this data is in the cache, and if so the data is also written to the cache; other than that these two hierarchies are identical with regard to writes.

In the hierarchy with queues on the other hand, the accelerator must initially set the address signal to the address of the output array, and the size to the number of 64 bit data elements, and then assert the write enable signal for 1 cycle. The accelerator can then immediately start pushing data into the write queue until it is full. Meanwhile the assertion of the write enable signal results in the write manager performing address

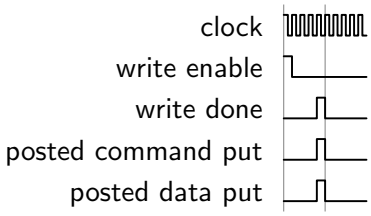


Figure 4.11: DMA Write signal transitions in basic hierarchy and hierarchy with cache.

translation, and splitting of the request into page aligned requests, before sending them to the DMA write unit. The DMA write unit then waits until the write queue contains enough data elements to send the next command packet (As explained in Section 3.3.2.3). Figure 4.12 shows how the accelerator can write multiple data elements without waiting. Once the queue contains enough data, the DMA write unit starts sending data and command packets.

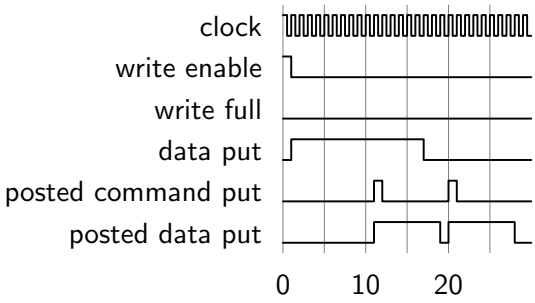


Figure 4.12: DMA with queues write latencies

4.2.1.4 Execution Complete

Once the accelerator has read and written all the data it asserts the done signal. The interrupt handler unit detects this, and sets the status register to execution complete, before sending the interrupt as explained earlier (Figure 4.13). In software the interrupt handler again starts the workqueue, which determines that the accelerator has completed by reading the status register. The driver then unlocks all pages, and wakes up the application. The application can now use the results of the accelerator, which in this case is a copy of the array.

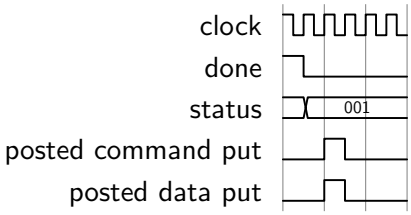


Figure 4.13: Signal transitions caused by execution completion of the accelerator.

4.3 Conclusions

In this chapter we have described the realization of the proposes system architecture. It integrates a GPP and an FPGA device interconnected with a High-Speed bus which supports high throughput and low latency. The platform consists of a AMD Opteron processor connected by HyperTransport to a Virtex4 FPGA. HyperTransport is a bus developed specifically for high bandwidth and low latency communication between processors and peripherals. The HTX development board allows us to connect an FPGA to the HyperTransport bus, which in combination with the HyperTransport FPGA core allows us to develop accelerators which can access the main memory and communicate with the host processor. We also describe step by step how to synthesize, program and compile software for reconfigurable acceleration. And then we take a step by step look at what happens in the hardware and software while executing a simple program on this platform.

5 Evaluation

We tested the implementation setup described in Chapter 4 with several accelerators. During the implementation we used the array copy and minmax accelerators from the Molen platform to determine whether the platform was functioning correctly. Once the platform provided the correct output, we used an AES accelerator also developed for Molen, to determine the performance of our platform.

In this Chapter we evaluate the results of the proposed system in terms of total area on the FPGA, and the performance of the AES application. First, in Section 5.1 we describe the AES encryption algorithm and the AES accelerator and application used as a case study for the evaluation of the proposed system architecture. Next in Section 5.2 we discuss the implementation results in number of slices and number of BRAMs used. We also evaluate the latency of memory accesses in the three different memory hierarchies, and then discuss the performance of the AES application in terms of speedup when used for different file sizes. Finally in Section 5.3 we summarize the results presented in this chapter.

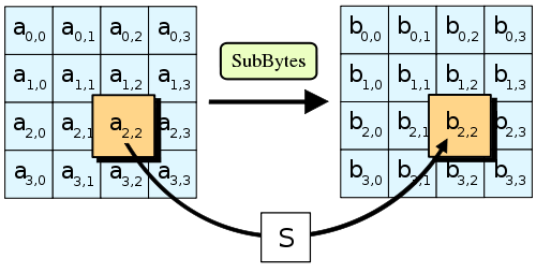
5.1 AES

The application used to test our system is an Advanced Encryption Standard (AES) cryptography application proposed in [25]. AES is a symmetric-key encryption algorithm, which means that the same key is used both for encryption and decryption. There are three possible key sizes: 128 bits, 192 bits or 256 bits key sizes. The larger the key used to encrypt the data, the more secure the encryption is, however larger key sizes take longer to encrypt and decrypt. The AES algorithm operates on blocks of 128 bits at a time, regardless of the key size. Each block is subjected to several rounds of encryption; the number of rounds depending on the key size used for encryption. In number of rounds of encryption for each block are 10, 12 and 14 in the case of 128-bit, 192-bit and 256-bit key sizes respectively. Each round can be divided into the four steps in Figure 5.2: *Substitute Bytes*, *Shift Rows*, *Mix Columns*, and *Add Round Key*. The first round consist only of the add round key step, whereas the last round consists of all but the mix columns step. All other rounds consist of all four steps.

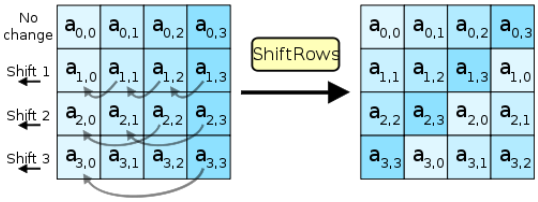
Figure 5.2a shows the first step, substitute bytes, where each byte in the block is replaced with values stored in a lookup table. These values are calculated according to the Rijndael S-box. In the second step the bytes in a block are cyclically shifted as seen in Figure 5.2b. The bytes in each row are shifted by a different amount, the first row is not shifted, the second is shifted left by 1, the third by 2 and the fourth by 3. Next, during the MixColumns step (Figure 5.2c, each column is multiplied by a predefined matrix which depends on the key-size. For 128 bit keys the matrix is 5.1, where multiplication with a certain number has a specific meaning. Multiplication with

$$\begin{matrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{matrix}$$

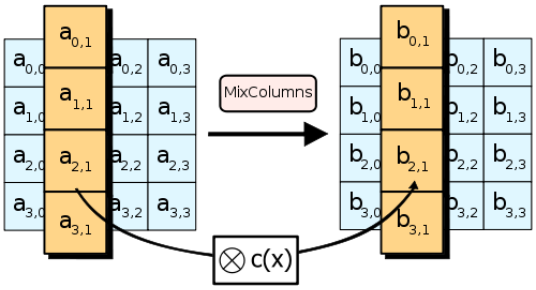
Figure 5.1: Matrix for Mix Columns step.



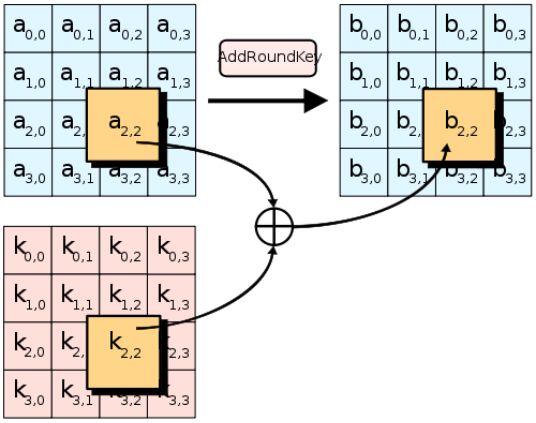
a: Each byte in the block is replaced with a value from a fixed lookup table.



b: Each byte in a row is shifted by a different amount, depending on the row.



c: Each column is multiplied by a polynomial to obtain a new column.



d: Each byte in the block is XOR-ed with a byte from the key.

Figure 5.2: The four steps in an AES encryption round. Source: Wikipedia[26].

1 means the byte stays the same, 2 means shit left, and 3 means shift left and XOR with the original value. Finally, during the addroundkey step, the block is XOR-ed with a subkey as shown in Figure 5.2d. The subkey is derived for each round from the key according to the Rijndael key schedule. Block cyphers like AES can operate in different modes. In the normal mode explained above, called Electronic Code Book (ECB), each block is encrypted and decrypted separately, and since the key schedule is the same for each block, this results in identical values being encrypted identically. When using the algorithm in Cipher-Block Chaining (CBC) mode, the first block is XOR-ed with an initialization vector, and each subsequent block is XOR-ed with the previous cypher-text.

The AES algorithm can be optimized by reducing the first three steps (substitute

Table 5.1: Implementation Resources (in slices and BRAMS) of the FPGA design blocks.

	Basic DMA		**DMA with Cache**		**DMA with Queues**	
	Slices	BRAMs	Slices	BRAMs	Slices	BRAMs
HTX interface	5,066	29	5,066	29	5,066	29
Wrapper	865	7	1,081	11	1,847	9
AES Accelerator	1,546	12	1,546	12	1,546	12
Total	7,477	48	7,693	52	8,459	50

bytes, shift rows, and mix columns) into a series of table lookups. We used an implementation of the approach described in [25]. In this implementation the shift rows and mix columns steps are combined into a single table lookup. This way a single round can be completed by two table lookups and an XOR, resulting in one round per cycle encryption and decryption times. The implementation was modified to use the new interfaces used to access memory, causing an additional 2 cycles to be needed for reading and writing data in the case of the memory hierarchy with queues, and an additional 8 for the basic memory hierarchy, and the hierarchy with cache. This results in a lower theoretical maximum throughput compared to the implementation described in [25].

5.2 Results

In this Section, we evaluate the proposed system implemented in the HTX platform described above. We use the AES program and accelerate in hardware the core encryption/decryption function following our methodology described in Chapter 3. We experiment using the three design alternatives for our FPGA wrapper and compare them with each other as well as with running the purely software version of the program to the AMD Opteron processor. The three alternative wrappers are the following: (i) basic DMA, (ii) DMA with cache, and (iii) DMA with queues. We first present the implementation results of the three design alternatives in terms of area resources and operating frequency of each module. Subsequently, we show the performance results of our approach showing the execution time of the AES program as well as the speedup over the purely software solution.

5.2.1 Implementation

Table 5.1 offers the area requirements of the FPGA modules. The HTX interface occupies about 5,000 slices and 29 BRAMs; that is due to multiple queues needed for the communication with the HyperTransport bus as well as tables to keep track of ongoing transactions. The wrapper with the basic DMA manager needs more than 800 slices and 7 BRAMs, when adding a cache to the DMA it needs about 1,100 Slices and 11 BRAMs, while the wrapper with the queues in the DMA needs roughly 1,800 Slices and 9 BRAMs. All wrappers need BRAMs for the TLB implementation while having queues or cache in the DMA adds 2-4 memory blocks. The wrapper with queues in the DMA

manager occupies more logic in order to be able to merge multiple requests in one and to support concurrent requests.

5.2.2 Latencies

The two main factors that determine how fast the accelerator will be compared to software are the data processing time of the accelerator, and the memory latency, namely the time spent waiting for data to arrive by the accelerator. We developed the three different DMA hierarchies in order to optimize and reduce the memory latency seen by the accelerator. Memory latency as seen by the accelerator originates from three sources: address translation, TLB misses, and DMA latency. The first, address translation, is the minimal delay caused by address translation unit in the case of a TLB hit. The second is the delay caused by the driver performing an address translation request in the case of a TLB miss. The last is the latency between sending a DMA read request and receiving the response data.

Table 5.2: Latencies in DMA read requests caused by Address Translation, TLB misses, and DMA requests. The averages show how each of these contributes to the total average latency of a DMA read.

	Basic DMA		**DMA with Cache**		**DMA with Queues**	
	Maximum	Average	Maximum	Average	Maximum	Average
TLB hit	4	4	4	4	6	0[1]
TLB miss	2000[2]	4	2000[2]	4	2000[2]	0[1]
DMA request	50	50	57	7.1	52	1[1]
Total	2054	**58**	2061	**15.1**	2058	**1**

In table 5.2 we show the maximum and average number of cycles for these delays in each of the three DMA hierarchies. The values for TLB misses in this table are averages for the read behavior of the AES example which reads sequentially from memory. An actual TLB miss has a greatly varying latency of several thousand cycles; however the average value lies around 2000 cycles. A TLB miss occurs once in every 4096 bytes when reading consecutively, or once for every 512 reads by the accelerator, resulting in an average value of 4 cycles per read. The number of cycles for the TLB hit is constant, and is determined by the way the address translation unit is implemented. In the basic DMA and cache DMA it is 4, while in the more complex implementation with queues it is 6 cycles. The latency for a single DMA read request of 64 bits has been observed to be 50 cycles.

In the DMA hierarchy with cache, where we request 512 bits, the 64 bit data packets arrive one cycle after the other resulting in 57 cycles. These 57 cycles are in the case of cache miss, meaning that on average for 8 consecutive reads it amounts to only 7.1 cycles per request. In the implementation using queues the latency of 52 cycles is the

[1]The latency is actually greater, however because multiple requests are sent in parallel, an access takes on average 1 cycle form the accelerators point of view.

[2]This value is an average over multiple TLB misses during a single run of the application.

Table 5.3: This figure shows the best case packetization and protocol overhead for reading and writing data in the Basic DMA hierarchy and in the other two hierarchies.

	Basic DMA	DMA with cache or Queues
Read	66.7%	20,0%
Write	60,0%	15,8%

sum of the 50 cycles and an additional 2 cycles needed by the data to propagate through the read queue. The DMA hierarchy using queues has latencies of 0 for the average TLB hit and miss and 1 for DMA request, because after the initial latency data arrives faster than the accelerator can process it. This happens because in this implementation DMA request, address translation requests, DMA writes, and processing are all done in parallel. This means that from the point of view of the accelerator there is only 1 cycle latency for read requests. For write requests the situation is similar, however there is no delay due to DMA requests since there is no need for a response.

5.2.3 Packetization and Protocol Overhead

HyperTransport was partly developed to have a low overhead in communication when compared to other standards such as PCI-Express. Command packets in HyperTransport can be either 32, 64 or 96 bits long, with the size of the packet depending on the number of address bits used. This platform uses 64 bit addresses, and so the command packets are 96 bits long. The three different hierarchies used for accessing memory each have a different overhead. We represent this by comparing the number of command bits sent to read or write a certain number of data bits. This is an important metric, since the total bandwidth available for transferring data is limited if most of it is used by command packets.

The HT-core supports a bandwidth of up to 1.6GB/s in each direction, meaning that by reducing the overhead more data can be transferred to and from the main memory. The difference in overhead is between the basic memory hierarchy, where each memory access by the accelerator translates to a DMA request, and the other two hierarchies were multiple data is requested with each request. In the basic hierarchy, reading data requires a single 96 bit command packet to be sent, and a 32-bit response packet and 64 bit data packet to be received. Writing on the other hand requires sending both a 64 bit data packet and 96 bit command packet. In the DMA hierarchy using a cache and queues, up to 8 64-bit data elements can be read and written using a single request. This means that reading 512 bits requires a 96-bit read request packet and a 32 bit response packet, while writing 512 bits requires only a 96-bit request packet. The resulting overhead percentages are shown in table 5.3.

The table shows that requesting multiple data per request significantly reduces the overhead of the command packets. The overhead shown in the table is computed based only on the difference between data and command packets. In the memory hierarchy with cache however, it is possible that not all of the data is needed, which can increase the overhead when compared with the queued implementation. This is because the

hierarchy with cache always requests 512 bits per request, while the queued version only requests the needed parts.

5.2.4 Timing

We evaluate the performance of the three alternative designs of our system and compare with software using the ECB mode of the AES application having 256 bit keys and input files ranging from 16Bytes to 64 MBytes. We have performed 10 different runs for each file size and report the median execution time and speedup; that is in order to avoid biasing our results with the cold-start of the first run which is always significantly slower due to multiple page faults. The above configuration of the AES reconfigurable accelerator processes 128-bits every 16 cycles on a cycle time of 10 ns, having a processing throughput of 80 Mbytes/sec. Although the bus supports IO throughput of 3.2 GBytes/sec, the above means that our accelerated function is compute bounded.

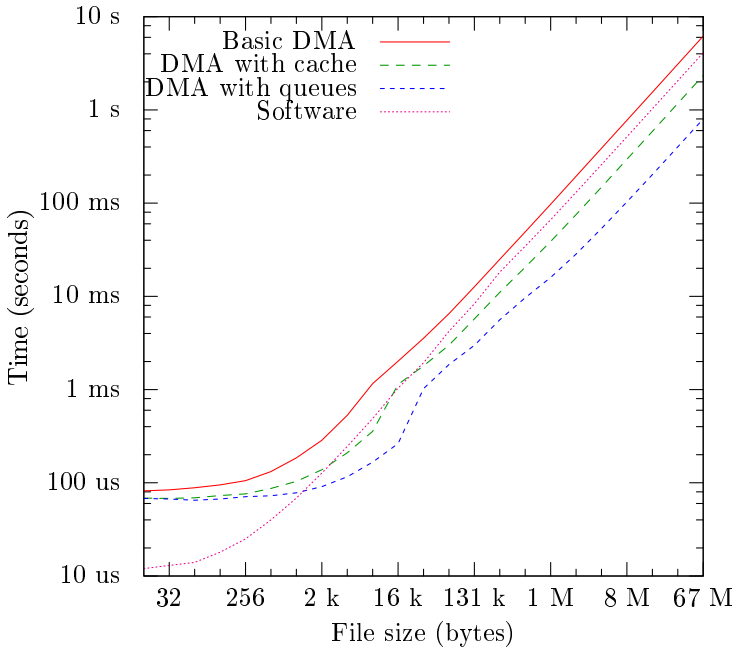


Figure 5.3: Execution times of the AES application in the HTX platform when running in software, and with hardware acceleration.

Next, we compare the performance of the AES application in four different alternative approaches. Figure 5.3 shows the execution time of the four alternatives for different file sizes to be encrypted, while Figure 5.4 shows the speedup of the hardware approaches compared to software. For small file sizes the software version is significantly better, up to 5× faster than the FPGA accelerated approaches. However, when the file size gets larger than 1KByte then accelerating the core AES function in hardware gets faster. For 64MByte files the FPGA accelerated version is 1.5-5× better than software. As depicted in the results, the basic DMA configuration is always slower than in software; practically

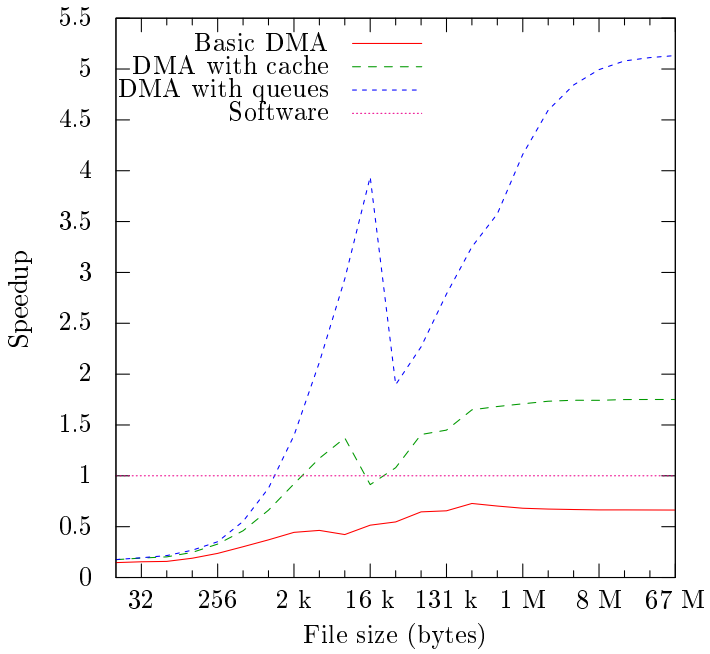


Figure 5.4: Speedup of the hardware accelerated approaches in the HTX platform compared to purely software.

the memory latency is dominating and the reconfigurable accelerator spends most of the time waiting for IO rather than processing.

The DMA version with cache is up to 2× faster than the basic DMA and up to 1.7 better than software. In this case, several memory accesses are avoided; a memory access for a single word (64-bits) brings to the FPGA cache a cache line of 8 words (512-bits) which may potentially give subsequent cache hits. The DMA with queues is able to improve about 3× the DMA with cache for large file sizes and up to 5× compared to software. In the DMA with queues, the memory latency is almost totally hidden since data are prefetched and stored in queues before they are consumed by the accelerator, while both read and write requests are packetized more efficiently allowing a single command to handle memory accesses for more than one words (up to 8 64-bit words).

In figure 5.4 we notice a significant reduction in speedup for files between 8 and 32 Kbytes. We have observed that this is due to TLB misses which are handled very slowly. Because TLB misses are handled by the driver using the workqueue approach, there is no way to guarantee the fastest possible handling of the interrupt. In some cases this causes TLB misses to take tens of thousands of cycles before they are handled. The result is that in very small files these long delays rarely occur, while in large files the delay is averaged out due to the large amount of data being processed. In the range between 8 and 32 Kbytes these long delays regularly occur, but cannot be averaged out. Another interesting note is that for large files (64 MBytes) the design with Queues in the DMA manager supports a processing throughput of about 80Mbytes/sec which is equal

to the theoretical maximum throughput. Finally, as expected, the larger the input sizes the better the performance of the system, since the setup overhead at the beginning of the program is then hidden by the long processing of the accelerated function.

5.3 Conclusions

We used AES to evaluate our platform. We measured the resources, communication overhead, communication latency, execution times, and speedup of the application. The largest implementation of the wrapper, which is the DMA with queues implementation, uses 1847 slices compared to only 865 slices for the basic DMA implementation. The average latency for the basic DMA hierarchy is 58 cycles, while for the DMA with queues hierarchy it is on average 1 cycle. The DMA with queues hierarchy also has at most 20% communication overhead, compared to 66% for the basic hierarchy. All this is also reflected in the timing and speedup results, where we see that while the basic DMA implementation is actually slower than software, the DMA with queues implementation is up to 5× faster than software. In summary, attempting to hide the memory latency is very crucial for system performance. Adding queues in the DMA can be applied in certain types of applications, such as streaming ones, due to their regular memory accesses. A cache next to the DMA seems to be the alternative for functions that have a more irregular memory access pattern. Finally, the overhead of integrating an FPGA card in a system as proposed in this paper is 10-15% in a Xilinx Virtex4-100.

6 Conclusion and future work

In this thesis we described a way to integrate reconfigurable computing into a general purpose computing machine. This is done using a high-speed link to provide low latency and high bandwidth access to the main memory. The specific objectives were to have a virtual shared memory, memory protection and paging, support for interrupts, an API to control the reconfigurable device, and a compiler extension to automatically convert C code into code which uses this API to accelerate the software.

In this chapter we summarize the work presented in this thesis, and how the objectives have been met. Next we discuss what future work can be done to improve the existing implementation.

6.1 Summary

In Chapter 2 we discussed previous work done on integrating reconfigurable computing with general purpose machines, and also some commercially available systems. In this chapter we learn that while implementing the reconfigurable fabric as a co-processor allows us to use off the shelf freely available components, it also introduces a significant bottleneck in the communication between the co-processor and main memory. We propose to get around this bottleneck by using one of the currently available high-speed on-board links such as HyperTransport or Intel Quickpath. In the rest of the chapter we also examine the MOLEN platform, which we use as a starting point for our design.

Next in Chapter 3 we present the actual design which consist of a device driver, accelerator wrapper and compiler plug-in. The driver provides support for interrupt handling, which is used for both address translation, and handling execution completion. The driver also provides software with an API to interact with the reconfigurable device. By using the kernel to perform address translation, we can provide both memory protection and paging transparently, without needing any support in hardware. The wrapper uses a TLB to provide address translation for the accelerator, allowing it to access the virtual shared memory. The wrapper also supports three modes of communication: Host to device, device to host, and host to main memory. host to device communication is used to exchange parameters, results and status information between the host and device, while device to host communication happens in the form of interrupts. In this chapter we also describe three different DMA hierarchies which we implemented. A basic but slow design, a design with an added cache to reduce the memory latency, and a design using queues, optimized for accelerators that access memory consecutively.

In Chapter 4 we describe the platform which was used to implement the design. We use HyperTransport for our high-speed link, and a development board made by the university of Mannheim to connect to it. A HyperTransport core developed at the same university is also used. We then continue to give a step by step example of what happens

in the wrapper, and in the driver when executing a simple program using hardware acceleration on this platform.

Finally in Chapter 5 we present the results of running an AES application on our platform. The results show that using the DMA hierarchy with queues can provide up to 5 times speedup, while hierarchy with a cache only provides up to 1.2 times speedup. The basic hierarchy is actually slower than a software implementation. The results also show that because of the immense startup time caused by TLB misses and other communication overhead, hardware acceleration is only advantageous for an application operating on large (several kB) data sets. Another interesting observation is that the speedup initially climes quickly before dropping and then slowly climbing again.

6.2 Contributions

In Chapter 1 we describe the requirements of the system in order to create a platform-independent solution for reconfigurable acceleration in general purpose system. The system architecture proposed in Chapter 3 meets all of these objectives, namely:

- **Virtual shared memory:** The reconfigurable accelerator can directly access the main memory, which is shared with the host, using virtual addresses. Whenever the accelerator performs a memory access, the wrapper translates the address to a physical address and sends the request over the HyperTransport bus. This allows the accelerator to access data without requiring explicit data transfer from main memory to a local memory on the FPGA.

- **Memory protection and paging:** Whenever the accelerator attempts to access a page in memory, the driver ensures that the accelerator is accessing a page belonging to the application begin accelerated. This prevents the accelerator from accidentally corrupting memory belonging to other applications, or the kernel. Additionally, if the page has been swapped out, the driver loads the data into memory and locks it, preventing it from being swapped back to disk.

- **Interrupt support:** We use interrupts to indicate TLB misses and execution completion. Interrupts allow the driver to be notified of these events without the need for continuously polling the FPGA device, allowing the system to perform usefull computations while waiting for the accelerator.

- **Controlling the Reconfigurable device:** The architecture uses system calls instead of ISA extension to control and communicate with the FPGA device. This precludes the need for hardware modifications, which allows us to use off the shelf components for the implementation

- **Compiler extension:** Our compiler plugin automatically inserts system calls necessary to lock the device, transfer parameters to the device, and start its execution. The only additional work on the part of the programmer is to mark the function being accelerated in the code.

6.3 Future work

The platform described in this thesis supports accelerating specific functions in hardware, by marking the function in code with a special attribute, and designing a hardware component which replaces it in hardware; However there are still areas which are not addressed. The main topics which are interesting for this platform are: Partial reconfiguration, non-blocking execution, multiple accelerators, and improving the startup time of the device. At the moment using different accelerators requires restarting the entire system, reprogramming the FPGA, and restarting again. Additionally, only one process can use the capabilities of the device at any one time.

6.3.1 Non-blocking execution

Currently when a process calls the execute function it is put to sleep until the execution is completed. In some cases it may be advantages to let the process continue execution until it reaches code which is dependent on the results of the function being accelerated in hardware. Doing so would require only changes in the driver and compiler plug-in. The drivers execute functionality should be changed to simply return instead of sleeping. Furthermore an additional system call should be added which when called by the software, checks if execution has completed and if not sleeps until it has. Freeing the pages locked by the device should also be moved to this new system call.

Additional support in the compiler plug-in would consist of inserting the new system call after the execute call. Additionally the plug-in should check the dependencies of the code, and only insert the call when execution cannon continue because of dependencies on data from the accelerated function. In order to do this successfully the plug-in must check not only the return value, but also any global variables used in the function, and also pointers to data passed into the function.

6.3.2 Partial reconfiguration

Partial reconfiguration support requires changes in both the hardware and software design. First of all, a set instruction must be added to the driver in the form of the ioctl system call. This call should write the starting address of a bitstream to the FPGA device, which should then read this bitstream and write it into the Internal Configuration Access Port (ICAP). Additionally the compiler plug-in must be modified to insert a call to this set instruction before any call to the execute instruction. A simplistic method to achieve this would be to simply insert the set instruction before the execute instruction, but [27] has already shown that better performance can be gained by inserting the call at earlier times. The methods described there can easily be applied to the compiler plug-in, or the other way around. Hardware support for runtime partial reconfiguration consists of support for the Set instruction, but also a component which reads the bitstream from memory and writes it to the ICAP. Additionally the design must be split into static and dynamic parts, with bus macros inserted in any signals crossing the boundary from the static to the dynamic part. This is needed to ensure that only the FPGA fabric assigned to the accelerator is reconfigured, and not parts used by the HT-core, and wrapper. This would result in undefined behavior of the HT-core, most likely resulting in the crashing

of the host machine. The bus macros are used to ensure that the signals are routed correctly between the static and dynamic parts, and also to prevent the signals from changing while the accelerator is being reconfigured.

6.3.3 Multiple Accelerators

Support for multiple accelerators would mean not only that multiple processes could be accelerated at the same time, but in combination with non-blocking execution it also allows a single process to accelerate multiple independent functions concurrently. Support for this requires some additional support in the driver, and also some changes to the plug-in. The wrapper will need to be significantly redesigned to support this. Driver support for this would consist of allowing multiple programs to access the device, the exact number depending on the number of accelerators supported by the FPGA. Additionally the interrupt handler would need to identify which accelerator caused an interrupt, and also which process a specific accelerator belongs to. This can be accomplished by adding more data to the interrupt register. Allowing multiple accelerators on the same reconfigurable device presents the hardware designer with some problems, the main ones being:

- How do multiple accelerators access main memory?
- How do we define which registers belong to which accelerator?

6.3.4 Startup Improvements

By analyzing the exact cause of the large delays during startup and possibly resolving these, the device can give large speedup even for small data sizes. This in turn would make the device more useful for applications which are not streaming, or only process a small amount of data at a time. Analyzing the interrupt behavior and making it more consistent would also provide greater speedup for intermediate file sizes.

Bibliography

[1] P. Bertin and H. Touati, "Pam programming environments: practice and experience," in *IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 133 –138, April 1994.

[2] J. M. Arnold, "The splash 2 software environment," *J. Supercomput.*, vol. 9, no. 3, pp. 277–290, 1995.

[3] M. Wirthlin and B. Hutchings, "A dynamic instruction set computer," in *IEEE FPGAs for Custom Computing Machines,*, pp. 99 –107, 1995.

[4] J. Hauser and J. Wawrzynek, "Garp: a mips processor with a reconfigurable coprocessor," in *The 5th Annual IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 12 –21, April 1997.

[5] S. Hauck, T. Fry, M. Hosler, and J. Kao, "The chimaera reconfigurable functional unit," in *The 5th Annual IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 87 –96, April 1997.

[6] R. Razdan and M. Smith, "A high-performance microarchitecture with hardware-programmable functional units," in *27th Annual International Symposium on Microarchitecture, MICRO-27*, pp. 172 – 180, nov.-2 dec. 1994.

[7] R. Wittig and P. Chow, "Onechip: an fpga processor with reconfigurable logic," in *FPGAs for Custom Computing Machines, 1996. Proceedings. IEEE Symposium on*, pp. 126 –135, apr 1996.

[8] "Intel quickpath." http://www.intel.com/technology/quickpath/.

[9] "HyperTransport Bus." www.hypertransport.org.

[10] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Panainte, "The molen polymorphic processor," *IEEE Transactions on Computers*, vol. 53, pp. 1363–1375, 2004.

[11] Y. Yankova, G. Kuzmanov, K. Bertels, G. Gaydadjiev, Y. Lu, and S. Vassiliadis, "Dwarv: Delftworkbench automated reconfigurable vhdl generator," *Field Programmable Logic and Applications*, pp. 697–701, Aug. 2007.

[12] Z. Guo, B. Buyukkurt, W. Najjar, and K. Vissers, "Optimized generation of datapath from c codes for fpgas," in *DATE '05: Design, Automation and Test in Europe*, pp. 112–117, 2005.

[13] "Altix sgi machines." http://www.sgi.com/products/servers/.

[14] "Convey computer." http://www.conveycomputer.com/.

[15] B. Salefski and L. Caglar, "Re-configurable computing in wireless," *Design Automation Conference*, June 2001.

[16] B. Holden, "Latency comparison between hypertransport™ and pci-express™ in communications systems," 2006.

[17] "Cray xd1 supercomputer." http://www.cray.com/products/Legacy.aspx.

[18] "SRC computers." http://www.srccomp.com/.

[19] S. Hauck and A. DeHon, *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. Morgan Kaufman Publishers, 2007.

[20] *Reconfigurable Application-Specific Computing User's Guide*, 2006.

[21] "The convey hc-1™ computer," 2008.

[22] U. Brüning, "The HTX board a universal HTX test platform." http://ra.ziti.uni-heidelberg.de/index.php?page=projects&id=htx.

[23] D. Slogsnat, A. Giese, M. Nüssle, and U. Brüning, "An open-source HyperTransport core," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 1, no. 3, pp. 1–21, 2008.

[24] U. Brüning, "HyperTransport core website." http://ra.ziti.uni-heidelberg.de/coeht/?page=projects&id=htcore.

[25] R. Chaves, G. Kuzmanov, S. Vassiliadis, and L. Sousa, "Reconfigurable memory based aes co-processor," in *13th Reconfigurable Architectures Workshop (RAW 2006)*, 2006.

[26] Wikipedia, "Advanced Encryption Standard — Wikipedia, the free encyclopedia." http://en.wikipedia.org/w/index.php?title=Advanced_Encryption_Standard&oldid=383379077.

[27] E. M. Panainte, *The Molen Compiler for Reconfigurable Architectures*. PhD thesis, TUDelft, June 2007.

[28] "Linux memory management." http://linux-mm.org/.

Wrapper functions

A

Listing A.1: The complete code of the wrapper functions inserted by the compiler plug-in.

```
#include <stdint.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/ioctl.h>
#include "htex_fcntl.h"

int htex_handle = 0;

int molen_htx_init(void)
{
        int handle = 0;
        if(htex_handle)
                return 0;
        handle = open("/dev/htex",O_RDWR);
        if(handle < 0)
        {
                perror("open");
                return 1;
        }
        htex_handle = handle;
        return 0;
}

void molen_write(unsigned long arg, unsigned long index)
{
        lseek(htex_handle, index*sizeof(unsigned long),
           SEEK_SET);
        write(htex_handle, (void *)&arg, sizeof(unsigned long))
           ;
}

unsigned long molen_read(unsigned long index)
{
        unsigned long arg;
```

```
        lseek(htex_handle, index*sizeof(unsigned long),
            SEEK_SET);
        read(htex_handle, (void *)&arg, sizeof(unsigned long));
        return arg;
}

void molen_execute(void)
{
        ioctl(htex_handle, HTEX_IOEXECUTE);
}

void molen_set(const char* filename)
{
        fprintf(stderr, "molen_set not yet implemented\n");
}

void molen_reset(void)
{
        ioctl(htex_handle, HTEX_IORESET);
}
```

Virtual Memory

B

Modern operating systems operate on virtual addresses instead of physical addresses. Each process including the kernel works in its own virtual address space. Whenever there is a memory access, the virtual addresses are translated to physical addresses, usually with the help of hardware support in the processor to speed up the translation. The advantages of each process working in a separate address space are:

- Programs can only access pages within their own address space, reducing the likelihood of a single program crashing the system.
- Multiple programs can use the same physical page in main memory, increasing the efficiency of memory usage by the programs.
- Fragmentation is hidden from the programs as they see a contiguous range of free memory, while it may be mapped to non-contiguous physical pages.

The virtual and physical address space are both divided into multiple parts called pages. Each virtual page used by the kernel or a program is mapped to a physical page. On the host used for this design the page size is 4096 kB. All virtual addresses can be divided into a 12 bit page offset, which addresses the bytes within a page, and a 52 bit

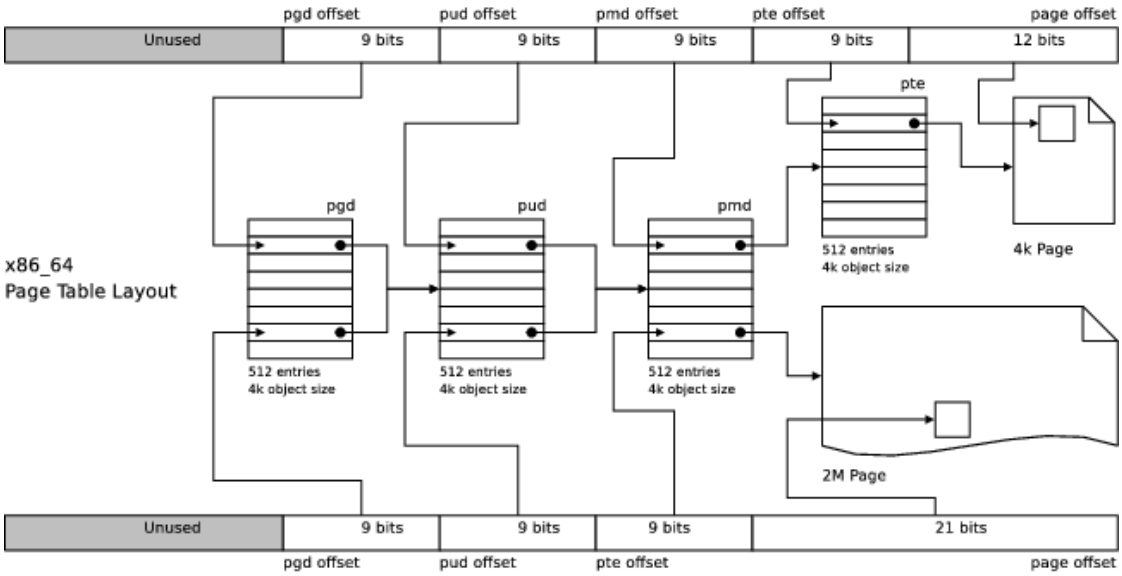


Figure B.1: Example of translating virtual to physical address using page tables on a 64 bit machine. Source: Linux Memory Management Website[28].

page number. Each page number identifies the page, and can be mapped to a physical page. During address translation this virtual page number is translated to a physical page using tables maintained by the operating system. These tables are called page tables, and can contain a mapping for every single virtual page. Figure B.1 shows how a virtual address can be translated into a physical address using the page tables. In the figure the page table is split into several levels. Each segment of the address (pgd offset, pud offset, pmd offset, pte offset) provides the index into the corresponding level of the page table. The entries in each of these point to the starting address of the next level of the page table. In order to look up the address first the entry in the page global directory is looked up using the pgd offset. This provides the address of the pud table, which is used to look up the address of the page middle directory. The page middle directory provides the address of the page table entries, which finally results in the actual page address.

Address translation requires hardware support in the form of a unit called the Memory Management Unit. The MMU performs the lookups described in figure B.1 transparently from the software. All the operating system needs to do is maintain the tables, and set the address of the page global directory at startup. When the MMU encounters a virtual address for which the physical address is not currently mapped, a page fault is generated. The operating system is then responsible for finding the page in secondary storage (usually the hard-disk), reading it into memory, and updating the page tables. If the page is not found the memory access is invalid, and the operating system handles this by terminating the application.