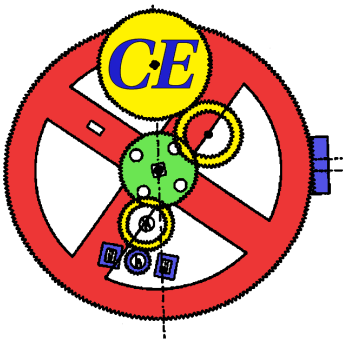

Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2011

MSc THESIS

Hardware Support for Dynamic Partial Reconfiguration

Venkatasubramanian Viswanathan

CE-MS-2011-23

Abstract

Dynamically reconfigurable architectures have demonstrated superior performance in comparison to the general-purpose processors. This thesis describes a generic approach for Dynamic Partial Reconfiguration (DPR) of a reconfigurable platform, connected to a general purpose system through a high-speed interconnect. Thus, the system can dynamically install and execute hardware instances of software functions (bitstreams) on-demand. Furthermore, the thesis also serves as a starting point for accelerating multiple software functions concurrently on the hardware. To achieve DPR, system calls are inserted into the original program. The host processor of the system, thus manages the hardware reconfiguration and execution through a Linux device driver. Accelerating multiple software functions is achieved by implementing multiple accelerators on the hardware, managing their interfaces for communication, and providing memory consistency for read and write requests. To do so, the driver provides support for non-blocking execution of multiple software functions. The above system is implemented on a general purpose machine providing a Hyper Transport bus to connect a Xilinx Virtex4-100 FPGA, an AMD Opteron-244, and 1 GB of DDR main memory. The efficiency of the system is evaluated using audio processing and encryption workloads. The proposed system achieves a 12x speedup over software with audio processing workload and 13x speedup when the workloads were accelerated concurrently.

TU Delft

Faculty of Electrical Engineering, Mathematics and Computer Science

Hardware Support for Dynamic Partial Reconfiguration

Accelerating multiple functions

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Venkatasubramanian Viswanathan
born in Chennai, India

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

Hardware Support for Dynamic Partial Reconfiguration

by Venkatasubramanian Viswanathan

Abstract

Dynamically reconfigurable architectures have demonstrated superior performance in comparison to the general-purpose processors. This thesis describes a generic approach for Dynamic Partial Reconfiguration (DPR) of a reconfigurable platform, connected to a general purpose system through a high-speed interconnect. Thus, the system can dynamically install and execute hardware instances of software functions (bitstreams) on-demand. Furthermore, the thesis also serves as a starting point for accelerating multiple software functions concurrently on the hardware. To achieve DPR, system calls are inserted into the original program. The host processor of the system, thus manages the hardware reconfiguration and execution through a Linux device driver. Accelerating multiple software functions is achieved by implementing multiple accelerators on the hardware, managing their interfaces for communication, and providing memory consistency for read and write requests. To do so, the driver provides support for non-blocking execution of multiple software functions. The above system is implemented on a general purpose machine providing a Hyper Transport bus to connect a Xilinx Virtex4-100 FPGA, an AMD Opteron-244, and 1 GB of DDR main memory. The efficiency of the system is evaluated using audio processing and encryption workloads. The proposed system achieves a 12x speedup over software with audio processing workload and 13x speedup when the workloads were accelerated concurrently.

Laboratory : Computer Engineering
Codenumber : CE-MS-2011-23

Committee Members :

Advisor:	Ioannis Sourdis, CSE, Chalmers University
Advisor:	Georgi Gaydadjiev, CE, TU Delft
Chairperson:	Koen Bertels, CE, TU Delft
Member:	Koen Langendoen, ES, TU Delft

Dedicated to my family and friends. Especially my parents and brother for always supporting and encouraging me in all my endeavors.

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank Georgi Gaydadjiev for having given me the opportunity to work on this master thesis topic, taking his time to read and correct my thesis and providing me financial support throughout the second year. I am very grateful to Ioannis Sourdis for providing me his guidance throughout my thesis and spending time to read and correct my thesis work. It would not have been possible to publish my work in FPT'11 if not for his valuable inputs and suggestions. I would like to extend my gratitude to Anthony Brandon for helping me understand the HTX platform during the initial stages in my work. I would also like to thank Google for supporting this work in the context of their Faculty Research Awards program.

Words certainly can not describe how grateful I am to my parents and my brother for all their motivation and encouragement. My parents have always been a constant source of support; emotional, moral and of course financial, and this thesis would certainly not have existed without them. My family and my cousins have been, always, my pillar, my joy and my guiding light, and I would like to thank them all. Finally, I am indebted to my friends Srini, Ram, Abhijit, Venkatraman, Shreyas Bhargav, Ajith, Ganesh, Chokka and many others who have made my stay very enjoyable and memorable through these years. Their support and care has made these years of my graduate study priceless. I greatly value their friendship and hope to stay in touch with them throughout the rest of my life.

Venkatasubramanian Viswanathan
Delft, The Netherlands
October 1, 2011

Achievements

Conference Publications

- Ioannis Sourdis, Venkatasubramanian Viswanathan, Abhijit Nandy, Anthony Brandon, Dimitris Theodoropoulos, Georgi N. Gaydadjiev, **Reconfigurable Acceleration and Dynamic Partial Self-Reconfiguration in General Purpose Computing,** *International Conference on Field-Programmable Technology (FPT), New Delhi, India, December 2011.*

1 Introduction

Traditional computation involves either implementing the application in hardware (e.g., ASICs) or implementing them in software running on processors (e.g., DSPs, micro controllers, embedded or general-purpose microprocessors). More recently, Field-Programmable Gate Arrays (FPGAs) have emerged as a platform for implementing compute intensive applications. Such devices combine the post-fabrication programmability of processors with the spatial computational style most commonly employed in hardware designs. Using FPGAs for computing led the way to a new class of computer organizations which we now call reconfigurable computing architectures. Reconfigurable computing bridges the gap between traditional hardware/ software approach by giving the programmer/systems designer the flexibility of hardware implementation. By combining reconfigurable computing with general purpose computing we can accelerate critical time consuming parts of the program on the FPGA while still allowing other non-critical parts of the program to be run on the general purpose processor. This eliminates the need to implement the complete design as a hardware structure.

Integrating Reconfigurable computing into traditional Computing systems has been a challenge since the early days of FPGAs; several machines have been designed towards this direction, such as PAM [5], Splash [6], and DISC [7]. The primary drawback of all these attempts was the limited bandwidth and high communication latency between the general purpose host-processor and the reconfigurable device. In order to overcome this

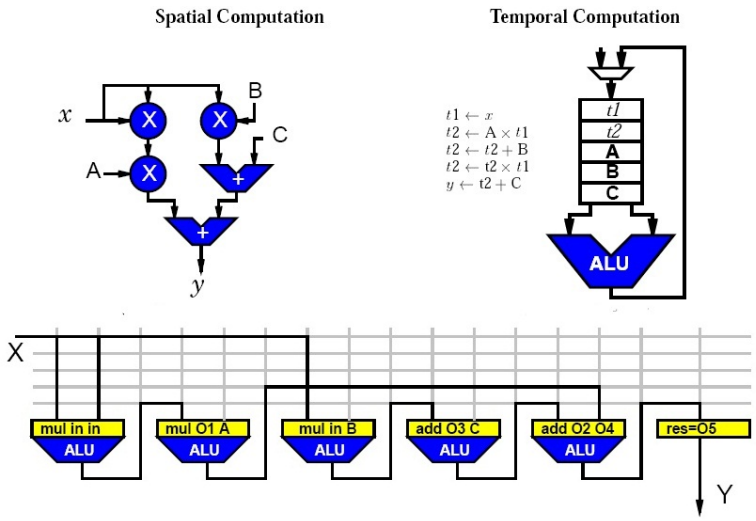

Figure 1.1: Spatial versus Temporal Computation for the expression Ax^2 +Bx+C

bottleneck, other solutions proposed a more tightly coupled reconfigurable-unit which is at the cost of significant changes in the host-processor architecture, e.g., Garp[8], Chimaera[9], PRISC [10], and OneChip [11]. Recently, however, components such as motherboards, have been released which support standard high-speed communication between general purpose processor(s), memory, peripheral and other input/ouput devices. These components use high-speed on-board links such as the Intel QuickPath (QP) [12] and the AMD HyperTransport bus (HTX) [13] and provide multi-GByte/sec bandwidth, low-latency communication. The above mentioned developments offer a new opportunity for integrating reconfigurable computing into general purpose computing systems. Several commercial systems have been released offering reconfigurable acceleration combined with traditional computers, but they all are in general hard to program. Examples of such machines are the Altix SGI[14], ClearSpeed[15], and Convey[16], which mostly target High-Performance scientific computing rather than the General Purpose domain. Another interesting approach, recently announced by Intel, is the new Atom embedded processor with an Altera Arria FPGA built into the same package connected to the Atom via two PCIe buses[17]; however, even on the same die the PCIe bandwidth is about an order of magnitude lower than an HTX or a QP.

Rather than building a machine from scratch or suggesting fundamental architectural changes, it is more performance and cost-efficient to propose a generic solution that uses existing off-the-shelf components with only software (and configuration) modifications. Existing platforms such as HTX[18] describe a generic solution that requires only an FPGA device driver, few compiler extensions and a reconfigurable wrapper inside the FPGA to provide reconfigurable acceleration for General Purpose Computing. However, this approach also has few shortcomings as described in the following sections. Thus according to Moore's law, the saturating computational power of processors calls not only for a whole new era of custom computing, but also techniques to exploit the available resources and computational power of FPGAs.

The remainder of this chapter is organized in four sections. Section 1.1 states the motivation for work. Section 1.2 describes the problems with current platforms for combining general purpose computing with reconfigurable computing. In Section 1.3 states the main objectives and goals of the thesis and gives an outline of the main contributions. Finally, Section 1.4 gives an overview of the remaining chapters of the thesis.

1.1 Motivation

Some decades ago a circuit designer had little choices when implementing a new system. If flexibility was needed, it was built using general purpose processors (GPPs). Demand for more speed meant building custom solutions in hardware, so application specific integrated circuits (ASICs) were chosen. Over the years, more and more alternatives appeared on the market. Digital signal processors (DSPs) were designed for data-flow intensive computation. Glue logic grew to programmable logic arrays (PLAs) and

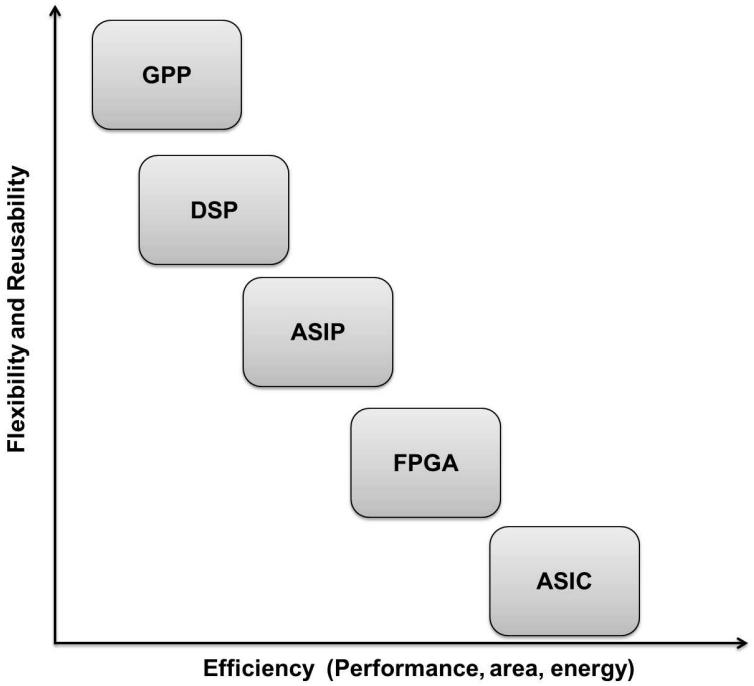

Figure 1.2: Trade off between flexibility and efficiency

complex programmable logic devices (CPLDs). Along with them, Field Programmable Gate Arrays (FPGAs) were introduced. Like all the other techniques, FPGAs fill a gap. From a system designer's point of view, they can be seen as programmable ASICs offering less speed, less power efficiency but more flexibility. Since FPGAs offer spatial computing capabilities just as ASICs do, FPGAs offer the same concurrency in calculation as the ASICs. Figure 1.2 shows the flexibility/efficiency trade-off provided by different devices.

On the other side, the computational power of GPPs were reaching a saturation due to the decrease in the size of the transistors which demands an ever increasing power utilization. Therefore, to overcome this limitation of GPPs, by exploiting the computational power and flexibility of FPGAs, there came a need for custom computing which combined the flexibility of GPPs with the efficiency of FPGAs. This gave rise to a whole new field of opportunity as well as challenges which are described in the following sections.

1.2 Problem Statement

The above mentioned platforms for custom computing have various drawbacks which has been explained in the following paragraphs. The first drawback being the memory access latency and the data bandwidth between the reconfigurable accelerator and the main memory. Since most of the applications that are chosen for reconfigurable acceleration

are compute intensive, they require the access of huge amount of input data. This is an important point of concern because, having memory access latency comparable to that of compute time will almost halve the speedup and thereby reducing the efficiency of accelerating the application on the hardware. Therefore, the accelerators should have low memory access latency. Some architectures such as GARP [8], Chameleon[15], Chimaera[9], OneChip[11] and PRISC[10] attempt to solve this by integrating the reconfigurable accelerator in the same chip as the general purpose processor. However, this requires redesign of the entire chip and it has to undergo a separate fabrication process. Thus making the process more time consuming and 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 [19], help reducing the bottleneck presented by data access.

The second drawback is that, most of the architectures that has been developed are very much platform specific and depend on specific compilation process and tool chain. This makes the task of programming the device dependent on the knowledge of a particular platform and tool chain. This also means that significant work is needed to port either the application or accelerator (or both) to a different implementation of the same platform. Maxeler Technologies MAX3 acceleration card uses a platform specific bus connection to connect to the processor, memory and other peripheral devices [20]. It also uses platform specific tools for compiling the code for the platform. Several other systems with RPUs as co-processors are being developed such as DRC Accelium Coprocessors [21], SRC Computers [22] have been developed commercially. But all these systems also use specific compilers, APIs and tool chains. Some platforms such as Molen[23], Convey[16], and Altix[14] address this by specifying a specific interface to the accelerator and software.

The next drawback is that most of these systems are not suitable for general purpose computing. Convey and Altix are aimed at high performance computing intended for scientific research instead of general purpose computing. Molen, which is not aimed at high performance computing, nor uses any custom hardware, is not ideal for general purpose computing. This is because, Molen is implemented entirely on an FPGA board. Further, 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. The final drawback of reconfigurable computing systems in general is the programmability. The common solutions for programming these systems are as follows. First, either some new or extended programming languages are used, which is then transformed by a compiler into intermediate form which uses hardware. Or, they use system specific function calls to manage the reconfigurable device.

HTX [18] is a general purpose platform with a reconfigurable device integrated into the system, using a high-speed interconnect. Architecture and details of the HTX has been explained in Chapter 2.4. Significant work has been done on the HTX reconfigurable platform, to address the above mentioned problems and create to a

generic platform for accelerating applications. The platform uses a general purpose machine, with a reconfigurable accelerator, a Linux device driver and a GCC compiler plug-in to identify parts of source program and accelerate parts of it on the hardware. However, the platform still has the following drawbacks. First, it does not provide support for dynamic partial reconfiguration. This means that significant amount of time has to be spent on reprogramming the FPGA device every time a different software function has to be accelerated. Moreover, the platform does not provide non-blocking execution of the function. Whenever a software function is accelerated on the hardware, the Linux device driver puts the calling software function to sleep, thereby preventing any further execution of the program until the device returns the results. Finally the device does not provide hardware support to accelerate multiple software functions of a process, simultaneously on the hardware.

All these existing problems with integrating a reconfigurable platform and programming it can be summarized as follows:

- The reconfigurable accelerator should be connected to the host system using an existing low latency/high bandwidth link;
- The architecture should be easily implementable and portable on different platforms with slight modifications of the existing one;
- Platform should provide runtime partial reconfiguration support to support the acceleration of any software function on the fly;
- Platform should provide non blocking execution to accelerate multiple software functions simultaneously whenever possible.

1.3 Objectives and Goals

The previous section described the various problems that exists in the field of reconfigurable computing and also explains the various challenges involved in integrating a reconfigurable platform into a general purpose system. Although HTX [24] addresses most of the issues, there are still a few drawbacks on the existing system as mentioned in the previous section. The goal of this thesis is to address those issues by improving the architecture of the existing system by providing the following functionalities:

- **Dynamic Partial Reconfiguration:** Xilinx FPGA technology provides the flexibility of modifying an operating FPGA design by performing partial reconfiguration, using a partial bit file as shown in Figure 1.3. Dynamic Partial Reconfiguration, also known as an active partial reconfiguration permits to change the part of the device while the rest of the FPGA is still running. Having DPR allows us to dynamically install and execute hardware instances of software functions (bitstreams) on-demand. It thus gives the ability to accelerate any application on the fly, without having to spend much time on reprogramming the entire FPGA. The

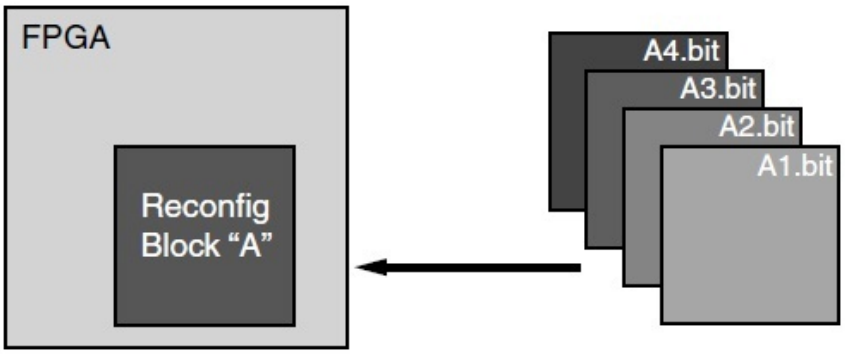

Figure 1.3: Basic Premise of Partial Reconfiguration

Dynamic Reconfiguration is initiated by the program, which has locked the device. In the hardware an ICAP controller reads the bitstreams from the memory and feeds it to the Internal Configuration Access Port (ICAP) thus initiating Dynamic Partial Self-Reconfiguration (DPSR);

- **Multiple accelerators:** The existing architecture supports the acceleration of a single software function at a time. However, advanced architectures and compute intensive applications demand the acceleration of multiple software functions simultaneously, so as to minimize the execution time. Having such an architecture not only allows us to accelerate two functions at the same time whenever possible, but also minimizes partial reconfiguration overhead time. This can be achieved by having multiple accelerators on the hardware, managing their interfaces with the hardware wrapper for read/write requests, and providing memory consistency for requests from different accelerators;

- **Hardware support for multiple accelerators:** When having two accelerators on the hardware, the FPGA Wrapper has to be extensively modified to provide proper management of resources and provide memory consistency for requests from different accelerators. First, the arbiter manages the parallel resource access requests from multiple accelerators. Furthermore, it also provides memory consistency for read and write requests from individual accelerators by identifying the request/response packets, and pushing them into corresponding read/write queues. Finally there has to be a synchronization mechanism between the reads and writes of different accelerators;

- **TLB support for streaming applications:** When executing two streaming applications on the hardware concurrently, the arbiter detects the address dependencies for read and write requests from the accelerators. When it does so, the accelerators are scheduled to execute sequentially one after the other. However, this results in the consumer accelerator waiting for a longer period, until the producer writes back all its results to the memory. In order to avoid this, the data

produced by the first accelerator should be immediately available to the second, their execution has to be pipelined and synchronized. In order to do so, memory access and address lookup mechanisms are modified by providing additional support for locking and unlocking the TLB entries;

- **Compiler support for multiple accelerators:** The system calls will be inserted in the original code using a modified GCC compiler plug-in after annotating the function to be accelerated. However, additional support has to be provided to the compiler to identify multiple annotated functions and insert system calls accordingly;
- **Device Driver support for non-blocking execution:** The existing Linux device driver has been modified to provide non-blocking execution support. It allows a single process to accelerate multiple independent functions concurrently.

1.4 Overview

This chapter has described the main motivations for the work, provided an overview of the problems in the existing platforms and has stated the main objectives of the work. This thesis has further been divided of four chapters organized as follows.

Chapter 2 discusses some of the background on reconfigurable computing. We explain the difference between course grained and fine grained reconfigurable fabrics, and their influences in architectural choices. It then explains the different memory hierarchies in which an FPGA can be integrated in a General Purpose processing system. It further discusses some of the commercially available reconfigurable General purpose systems and their features. Finally we also look at how the HTX architecture solves the problems by providing solutions such as easy of programmability, virtual memory, memory protection and paging, interrupt support, compiler and driver support.

Chapter 3 explains the architectural changes required for Dynamic Partial Reconfiguration support. Furthermore, it also describes accelerating multiple software functions on hardware. Several modifications have been made to the existing FPGA Wrapper modules to provide hardware support. Compiler and driver support have been provided for non-blocking execution. Finally, it provides an overview of the architecture of the audio beamforming processor used to evaluate the proposed system.

Chapter 4 evaluates the system using different performance metrics. Two different applications are accelerated in hardware and the execution time and speedup achieved are compared with that of a pure software implementation of the same. This is followed by an evaluation of the latency for performing a Dynamic Partial Self-Reconfiguration. Other parameters such as, area, power are measured. Finally, Chapter 5 draws some conclusions on the architecture and the results. It also describes some future works to enhance the features of the platform.

2 Background

Chapter 1 discussed about the various problems that we face in integrating reconfigurable and general purpose computing, and gave an introduction to the thesis work highlighting its main goals and objectives. This chapter explains the architecture, working and the implementation of the HTX platform [18] in detail, which forms the basis of this work. Furthermore it discusses some commercially available reconfigurable general purpose systems and also gives a short introduction to the Xilinx partition-based partial reconfiguration flow and terminology.

One of the main considerations when integrating reconfigurable systems with general purpose systems is the question of how to integrate the Reconfigurable Processing Fabric (RPF) into the system as explained in section 2.1. This plays an important role because, when accelerating compute intensive data processing applications, the memory access latency and bandwidth are critical factors. Examples include data-streaming applications with significant digital signal processing, such as multimedia applications like image compression and decompression, and encryption. Many architectures have been designed in this direction. Most of these are aimed at high performance computing such as XD1 [25], Altix [14], Convey [16] and SRC [22]. These systems are sold as supercomputers with added support for reconfigurable computing. Other platforms used for general purpose computing such as MAX3 Acceleration card[20] has the RPF in the same package and use specific tool chain for compilation.

For the above mentioned reasons, we have chosen HTX[18] as a starting platform because of the number of advantages it provides. First the RPF runs on a general purpose platform with a Linux kernel. It uses the existing off the shelf high bandwidth low latency components for communication, and it is easy to port to different platforms with slight modifications. It provides compiler and driver support for accelerating software programs in hardware in a generic way. Therefore the architectural features can be improved to provide dynamic runtime partial reconfiguration support. Moreover, it uses open-source tools and OS, and hence it can be customized at any point according to the needs.

The chapter is further organized as follows. Section 2.1 gives an overview of the different ways in which a reconfigurable platform can be integrated into the memory hierarchy a general purpose system. Section 2.2 discusses some commercially available general purpose reconfigurable systems. Section 2.4 describes in detail about the HTX architecture. Section 2.5 explains the basic terminology of Xilinx Partial Reconfiguration and finally section 2.6 summarizes the chapter.

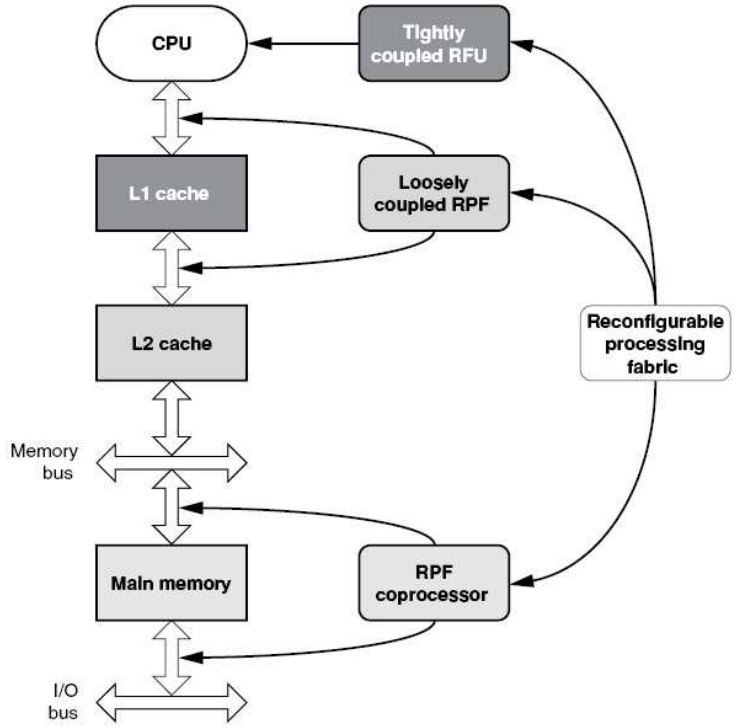

Figure 2.1: Possible locations for the RPF in the memory hierarchy. Source from Reconfigurable Computing: The Theory and Practice [1]

2.1 Memory Hierarchy

There are different ways to integrate the Reconfigurable Processing Fabric(RPF) in the memory hierarchy of a general purpose system as shown in Figure 2.1. In a tightly and loosely coupled hierarchy, the RPF is a part of the processor chip. However in the former, the reconfigurable fabric is used as a functional unit of a processor. In case of the latter, the RPF is connected to processor and the memory through a data bus. However in case of co-processor, the RPF is in a separate chip connected to a peripheral bus, allowing it to communicate with main memory.

Tightly couple hierarchy has the same memory access latency as that of any other functional unit. Whereas, loosely coupled hierarchy will have more latency. Since independent co-processor RPFs are are on separate chips, the integration of the RPF into existing computer systems is simplified through the available PCI slots on the system. Unfortunately, this limits the bandwidth and increases the latency between the RPF and memory in traditional processing systems depending on the peripheral bus interface used. For this reason, independent co-processor RPFs are well suited only to systems with high bandwidth, low latency communication links.

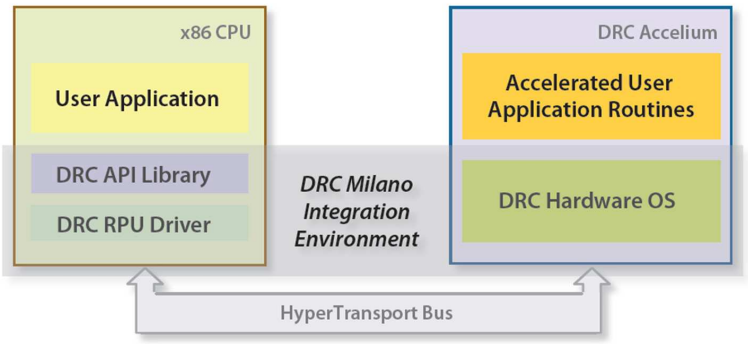

Figure 2.2: DRC Accelium architecture

2.2 Reconfigurable General Purpose Computing

This section explains the use of reconfigurable computing in general purpose systems that are commercially available. Several research organizations have made efforts to integrate FPGAs into a general purpose systems and make it commercially available. The following subsections explains a few commercially available systems and the features they provide.

2.2.1 DRC Computers

The architecture of DRC Accelium is shown in Figure 2.2. DRC computers provides the support for reconfigurable accelerator in the form of coprocessor [21]. The DRC Accelium Coprocessor is integrated with the GPP through a HyperTransport expansion slot. Accelium fits into a standard AMD Opteron PCIe expansion slot and provides very low latency transfers directly to motherboard memory uninterrupted by intermediate memory controllers. It uses a Xilinx Virtex-5 FPGA as its accelerator. In order to provide software support it uses a DRC Milano operating system to integrate the coprocessor into the system. It also provides API, driver and management interface on the CPU, and a hardware operating system on the DRC RPU. Further, it also provides run-time reconfiguration.

2.2.2 SRC Computers

The SRC IMPLICIT+EXPLICIT architecture allows the user to execute existing code, or recompile and develop code for the reconfigurable processor. In order to provide software support for such an architecture, Carte programming environment is used. The seamless integration of a general purpose processor and a reconfiguration processor into a system is called as Direct Execution Logic(DEL). The SRC Carte Programming Environment is an FPGA C and Fortran programming environment, taking applications written in Fortran and/or C and integrates the computational capability of MAP processors and microprocessors into a single application executable code. The main components of the Carte programming environment are MAP Compiler, Libraries, and debugger.

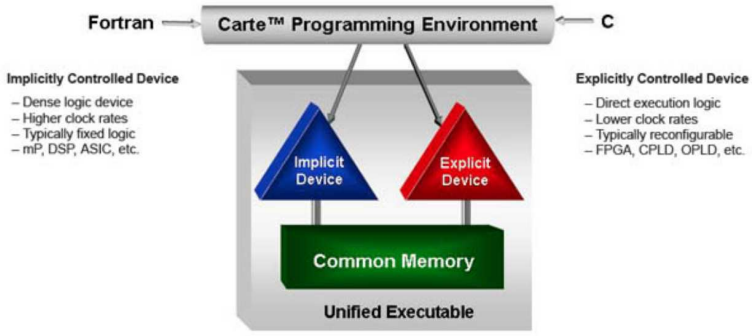

Figure 2.3: SRC IMPLICIT+EXPLICIT architecture

The main element of the architecture is a MAP reconfigurable processor. The MAP processor uses reconfigurable components to accomplish both control and user-defined computation, data prefetch and data access functions. This compute capability is teamed with very high on and off-board interconnect bandwidth. The MAP processor has multiple banks of SRAM memory. MAP modules are also equipped with separate input and output ports. However, it does not provide runtime reconfiguration.

2.2.3 Xtreme Data

XtremeData's XD2000i development system is a complete platform which has an FPGA coprocessor within an x86 COTS computing environment as shown in Figure 2.4. The system has complete hardware, software, and FPGA IP so users can immediately begin developing their FPGA-accelerated applications. The GPP used is an Intel Xeon processor with an XD2000i FPGA coprocessor bases on Altera technology. The coprocessor interfaces with the memory via the FSB available on the motherboard. Application software running on the Xeon can store and load FPGA configuration images that are located on the hard disk. XtremeData provides an open source Linux device driver which allows user space applications to directly access memory mapped FPGA resources, service FPGA interrupt requests, lock physical memory, and perform FPGA controlled DMAs. It also provides support for C to VHDL conversion.

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 where complex calculations are made in a limited time frame. 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 FPGAs use spatial computation models to exploit parallelism, they are increasingly being used in the field of HPC. Two currently available HPC systems with reconfigurable computing are SGI Altix [14] and Convey Computer's HC-1

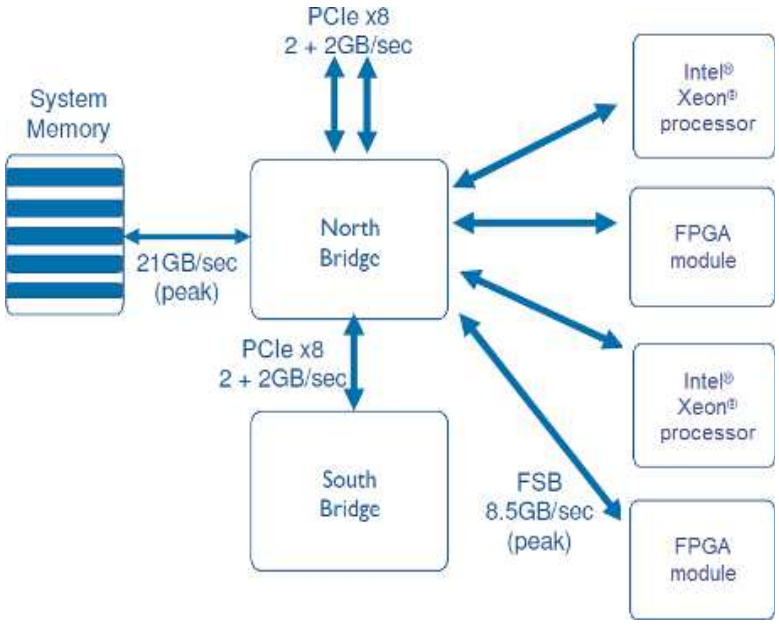

Figure 2.4: Xtreme Data XD2000i architecture

[16]. These systems use RPF with GPPs to allow acceleration. 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 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[2]. As shown in Figure 2.5, in RASC, an FPGA is connected to the SGI NUMA link[2] interconnect as a co-processor. NUMA link 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 service block provides interface between the Algorithmic block and the host system. 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.

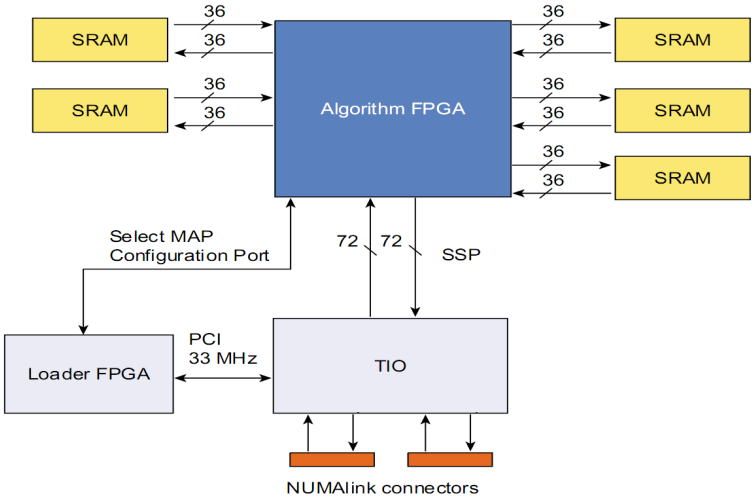

Figure 2.5: Block diagram showing how the FPGA is connected to the NUMA link interconnect and several banks of SRAMs. Source: Reconfigurable Application-Specific Computing User's Guide [2]

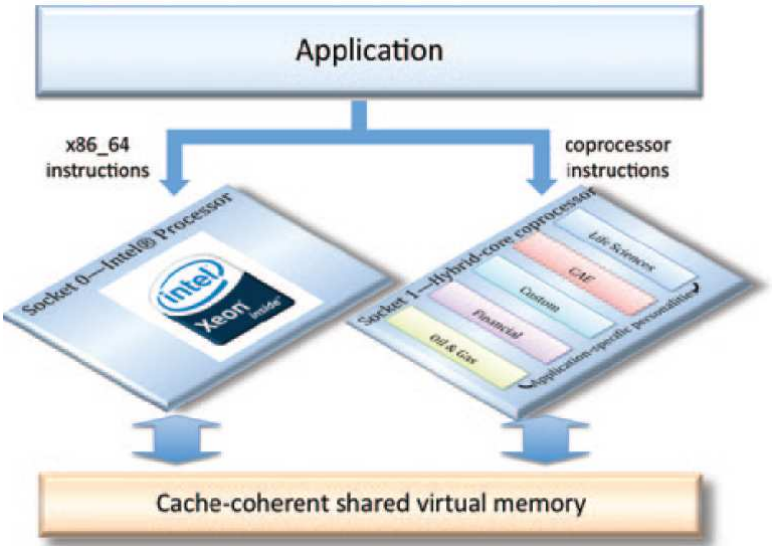

Figure 2.6: The Convey Architecture with Intel host processor, co-processor and shared virtual memory. Source: Convey HC-1 Architecture White Paper[3].

2.3.2 Convey

The Convey architecture (Figure 2.6) consists of off-the-shelf Intel processors in combination with a reconfigurable coprocessor. The Convey Instruction Set Architecture [3] 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

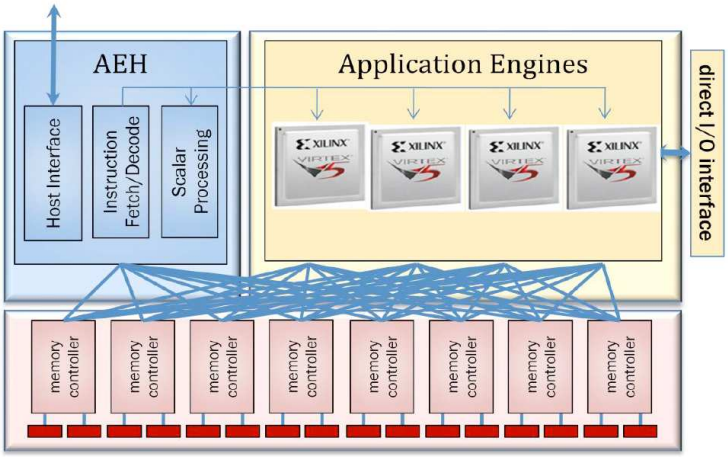

Figure 2.7: The coprocessor is divided into the Application Hub, Memory Controllers, and Application Engines. Source: Convey HC-1 Architecture White Paper[3].

personality is essentially an instruction set tailored to a specific application, and defines all 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 2.7): 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. 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.

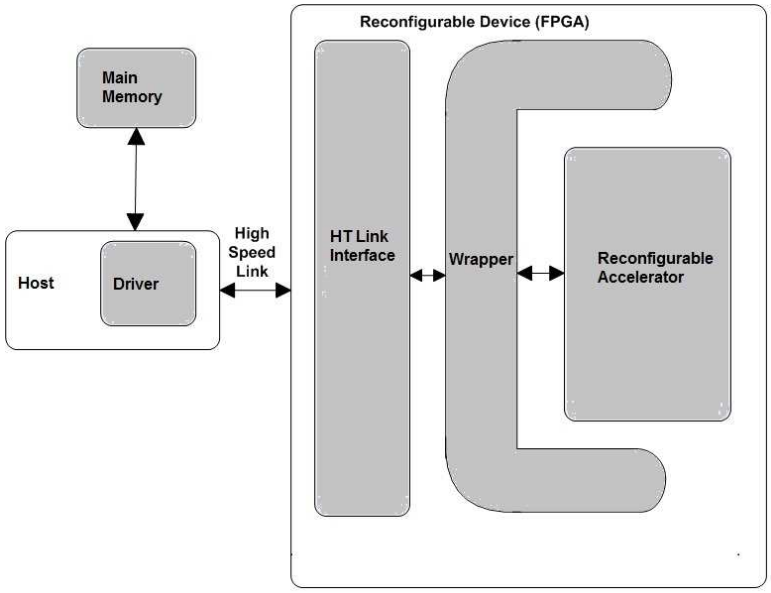

Figure 2.8: Top-level block diagram of the proposed system

2.4 HTX System Architecture

The HTX [24] is a general purpose platform with a reconfigurable device used for accelerating computationally intensive functions in hardware. Such functions are annotated in the code to indicate that they will be executed in hardware. Figure 2.8 illustrates the overview of the proposed system. The Reconfigurable device (FPGA card) is connected to the main memory through a high-speed link. On the host side, a Linux device driver has been developed for the communication of the FPGA device with the host GPP via system calls and interrupts. On the reconfigurable device, a HyperTransport link core is present for the device to communicate with memory. A wrapper is present to support the integration of FPGA device into the system. Finally, a reconfigurable accelerator is present to accelerate software functions on the hardware. It can be customized depending on the function to be accelerated.

2.4.1 Communication

Figure 2.9 offers a more detailed view of the internal modules of the hardware wrapper. The IO module controls all the memory mapped IO of the device. DMA reads and writes are handled by the DMA Manager and DMA Read/Write unit with the support of the Address translation module. Address translation uses a TLB copy of the host to translate virtual addresses to physical before each DMA request. Finally, the Interrupt Handler (IH) manages the interrupts raised by the device. In general, the wrapper is responsible for the following:

- Communication with the memory mapped IO regions; exchange registers (XREGs), a TLB copy, as well as control and status registers;

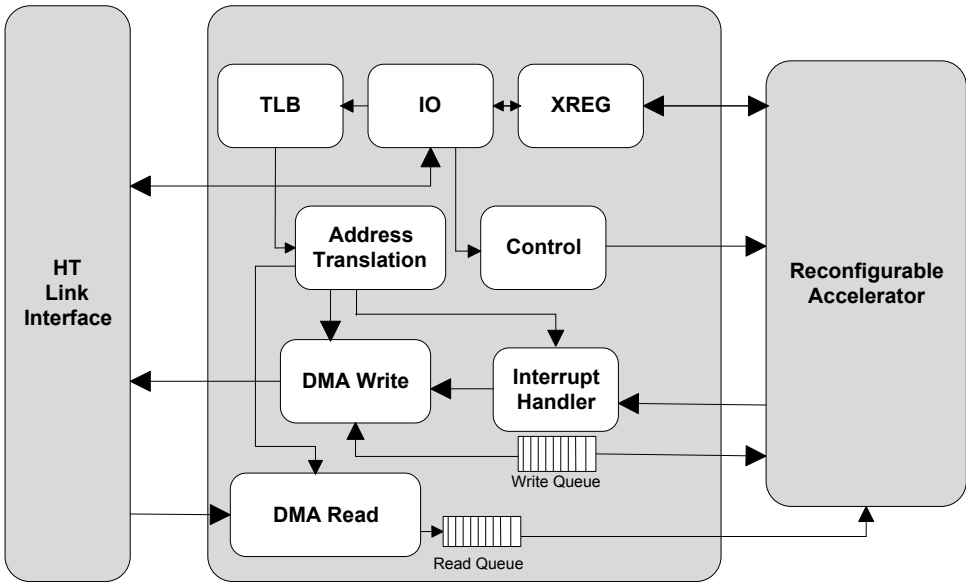

Figure 2.9: Detailed diagram of the modules located in the wrapper

- The control registers are used by the host to pass commands to the FPGA device, while the status registers hold information regarding the status of the device;
- The exchange registers (XREGs) are used for passing function-arguments from the host to the accelerator and return execution results to the host;
- Has an interrupt handler (IH) for the interrupts generated by the FPGA device;
- Performs Direct Memory Access (DMA) operations between the FPGA device and the main shared (virtual) memory;
- Translates virtual addresses to physical using a local TLB copy and handles TLB misses (by raising an interrupt).

The system mainly supports three types of communication between the host, reconfigurable device and the main memory:

- **Host to/from reconfigurable device:** The host can write the arguments of a function call to the FPGA through Linux system calls. The FPGA uses these arguments to read/write from/to the main memory. The host can also send control signals to to start the execution or reset the device. Finally it can write the physical address entries into the device in case of any address translation interrupts and can read back any return results of the device;
- **Reconfigurable device to host:** The device in turn will notify the host by means of interrupt, in case of execution done or address translation by setting a value for the status register which in turn will be be read by the host;

- **Reconfigurable device to main memory:** The FPGA can communicate with the main memory by means of a HyperTransport link interface. It can perform DMA reads and writes to read the input data and write back the output data.

2.4.2 Integration and Control of the Reconfigurable device

In order to integrate the reconfigurable device into the system we use several modules. As mentioned before, the Wrapper module in the FPGA plays an important role for communication from the hardware. There is a Linux device driver which provides the API for various system calls for reading from and writing to the device. Finally, there is also a GCC compiler plug-in that has been developed to identify parts of the program (annotated functions) that has to be accelerated on the hardware and insert corresponding system calls. All these components and their functionalities are explained in the following sections.

2.4.2.1 Driver Support

The Driver is responsible for providing the APIs to control the FPGA device. These APIs internally corresponds to a Linux system call. The following system calls are used to control the functionality of the FPGA device: *open(), close(), read(), write(), ioctl().* The *open()* system call is used by a program, running on the host processor, to get a lock on the device, while the *close()* system call releases the lock. The *write()* and *read()* system calls are used to write the arguments of function calls to the (memory-mapped) exchange registers (XREGs) of the FPGA device and to read the return values, respectively. Furthermore, the *ioctl()* call can be programmed to pass commands to the device by writing into the control registers using memory mapped IO. It is used to initiate the execution of the reconfigurable accelerator *ioctl(EXECUTE).* Using system calls rather than for instance extending the ISA of the host processor, like in [23], provides a more generic approach applicable to any system that supports a connection to an FPGA device.

2.4.2.2 Compiler Support

The programmer indicates the functions to be executed in hardware, by annotating them with GCC attributes: ___attribute___((user("replace"))). The compiler will then automatically insert the appropriate APIs provided by the driver. To do so, the GCC 4.5 compiler has been extended 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. The plug-in is executed by GCC as follows:

- When the compiler plug-in is loaded an entry function is called which registers a new attribute with the compiler. It also registers a new pass called replace pass;

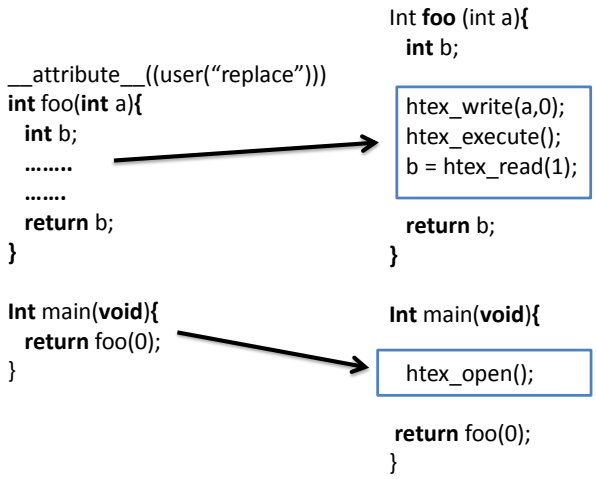

Figure 2.10: Code modification by the extended compiler

- The replace pass is run by the GCC after the code has been transformed into the internal representation. During the compilation of 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 the source file after the code has been translated into the internal representation.

Figure 2.10 illustrates an example of the system calls automatically inserted by the compiler to the original code in order to allow the execution of a function in the FPGA: first, the device is initialized (dev =open(DEVICE)), then the function parameter is passed to the FPGA (write(dev, a, 0)), subsequently, the function is executed in hardware *(ioctl(dev, EXECUTE))*, and finally the result is returned to the host (b=read(dev, 1)).

2.4.2.3 Address Translation

The FPGA device works on the virtual address space. Consequently, in order to access the 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. Address translation is handled by the Address Translation unit. Whenever an address must be translated, the address translation unit takes the following steps: First the input address. 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

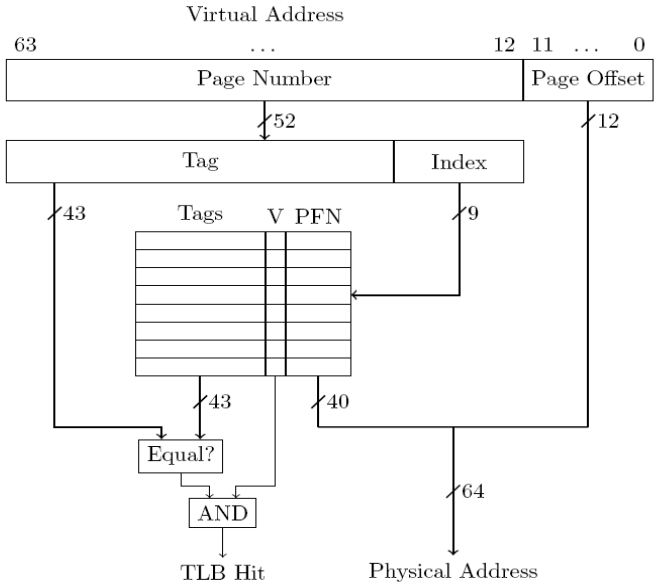

Figure 2.11: Diagram showing how the TLB is used to translate a virtual address to a physical address

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 tags, and any future tags such as read and write protection. Figure2.11 shows how the address translation unit uses the TLB to translate a virtual 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.

2.4.2.4 DMA Read and Write

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 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. Likewise, the manager translates the addresses for all the pages. 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 (HTX supports a maximum of 32 outstanding requests) , the DMA unit also checks if the request crosses this boundary and if so reduces the size of the data requested. Otherwise it requests 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.

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 as 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.

2.4.2.5 Interrupt Handler

Interrupt Handle in the wrapper supports the interrupts caused by the device in case of address translation or execution completion. 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 2.4.2.6. 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.

2.4.2.6 Memory Mapped IO

The host processor controls the FPGA device through the driver using three regions of memory mapped IO explained in Figure 2.12. 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). 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. The third region is used for the driver to read and write directly to/from the TLB. When the host processor writes to one of these regions, a command and data packet arrive on the posted input queue. Whenever the driver writes into the memory mapped IO region, a command and data packet arrives on the input queue of the HTX bus.

A hardware component called the read-write unit handles these requests. It consists of a state machine which detects a command packet in the input queue. It checks what memory region the request belongs to and writes the data either to the XREGs 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. 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 case of *EXECUTE*, the control unit asserts a start signal to the accelerator for one cycle. 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. 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.

2.4.3 Implementation Platform

The implementation platform, used to realize the proposed system, is shown in Figure 2.13. 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

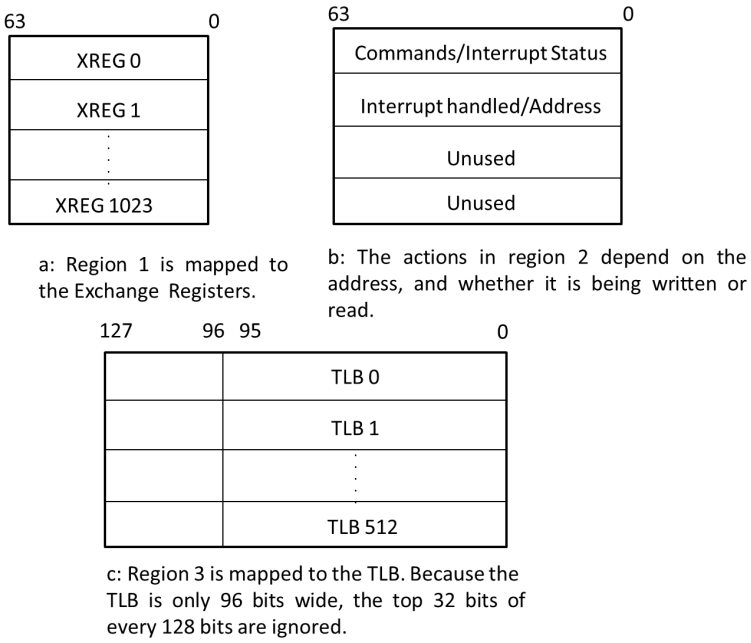

Figure 2.12: The three IO regions and how they are mapped to hardware

Machine	High Speed Link	Partial Reconfiguration	Multiple Accelerators	Open-source Platform
SRC Computers	x			
DRC Computers	x	x		
Xtreme Data	x			
Altix	x			
Convey	x			
HTX	x	x	x	x

Table 2.1: Overview of General Purpose Reconfigurable Platforms

HyperTransport bus [9] and an HTX connector which we use to connect an HTX FPGA board [13]. 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 [26] which supports 3.2 GBytes/sec throughput (bidirectional, 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 Intels QuickPath [27] with slight modifications to the implementation.

Finally the Table summarizes the various platforms available for reconfigurable general purpose computing and their features.

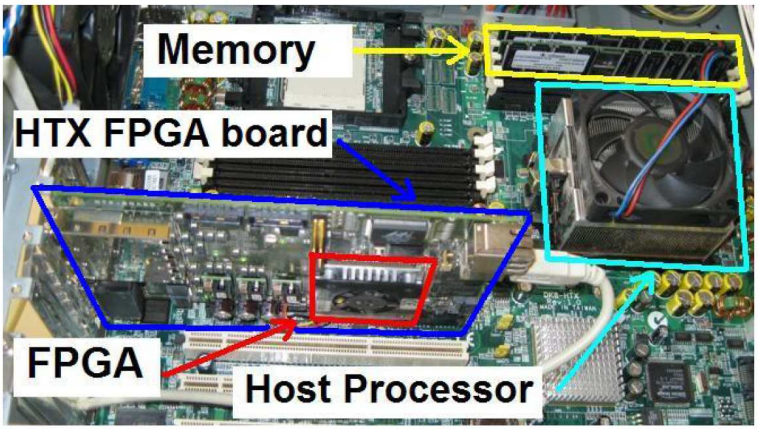

Figure 2.13: Implementation platform connecting a reconfigurable device to the host and the main memory through a HyperTransport bus

2.5 Xilinx Partial Reconfiguration

This section provides some introduction and describes the basic terminologies used in Xilinx Partial Reconfiguration. Partial Reconfiguration (PR) is the modification of an operating FPGA design by loading a partial configuration file. Xilinx FPGAs mostly use three different types of partial reconfiguration. They are XAPP290, Early Access Partial Reconfiguration (EAPR) and Partitioning. Each one of them progressively provides certain advantages compared to the previous one. The most widely used approaches are EAPR and Partitioning. Since the Partition-based partial reconfiguration provides several advantages over the EAPR techniques, in this thesis, the design has been adapted to the Partition based partial reconfiguration. The following states some advantages and some basic terminology.

The advantages of the Partition based technique as compared to EAPR are as follows:

- Bus Macros (BMs) are no longer needed. Bus Macros provide a means of routing the signals between Partial Reconfigurable Modules(PRM) and the base design. As a result, all connections between the PRMs and the base design must pass through a bus macro, with the exception of the clock signal,global signals, GND and VCC, are handled automatically by the tools. Bus Macros are shown in Figure 2.14. Bus Macros are replaced with Partition Pins and are automatically managed, and this automation replaces some of the aspects of Bus Macro functionality i.e. the designer no longer needs to manually route the signals between the static and reconfigurable partitions;
- It provides greater flexibility to the designer to as to the positioning and placement of the reconfigurable modules. This is provided by PlanAhead tool which give a GUI interface for floorplanning the design as required;
- MODE constraints and PR-specific environment variables depreciated. For example, the EAPR solution required the user to explicitly specify the coordinates

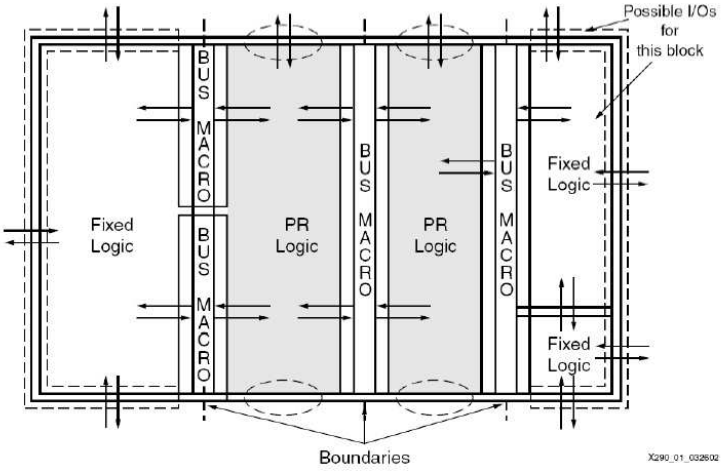

Figure 2.14: Bus Macros

of the reconfigurable module through User Constraint Files (UCF). Several other environment variables that had to be set for EAPR solution has been removed for Partition-based approach. In the latter, after floorplanning, the tool by itself creates an entry for the coordinates of the reconfigurable partition depending on the floorplan;

- User Constraint File (UCF) is required only during NDG build as opposed to EAPR where it was a requirement that the UCF be available for post-Translate implementation processes (MAP and PAR).

2.5.1 Partition-based approach

The various terminologies used in Partition-based partial reconfiguration are as follows:

- **Partition:** A Partition is a logical portion of the design, defined by the designer at a hierarchical boundary, to be considered for design reuse. A Partition is either implemented as new or preserved from a previous implementation. A Partition that is preserved maintains not only identical functionality but also identical implementation;

- **Reconfigurable Partition:** Reconfigurable Partition (RP) is an attribute set on a module, that defines it as reconfigurable. Software tools such as Ngdbuild, Map and Place And Route (PAR) detects the Reconfigurable Partition attribute on the instance and process it correctly;

- **Static Logic:** Static Logic is any logical element that is not part of a Reconfigurable Partition. The logical element is never partially reconfigured and is always active when Reconfigurable Partitions are being reconfigured. Static Logic is also known as Top-level Logic;

- **Reconfigurable Module:** A Reconfigurable Module (RM) is the netlist or HDL description that is implemented when instantiated by an instance that is a Reconfigurable Partition. There may be multiple Reconfigurable Modules for one Reconfigurable Partition;
- **Partition Pins:** Partition Pins are the logical and physical connection between static logic and reconfigurable logic. Partition Pins are automatically created for all Reconfigurable Partition ports.

2.6 Conclusion

This chapter has discussed the various aspects of reconfigurable computing and partial reconfiguration. Section 2.1 explained about the various memory hierarchies in integrating RPF with a general purpose computing system. When the RPF is tightly coupled, lot of modifications needs to be done to the ISA of the processor and also has to undergo a different fabrication process. However, in case of loosely coupled hierarchy, the only important consideration is the bandwidth and memory access latency between the RPF and the memory and processor. Failing which, memory access becomes a bottleneck resulting in very high memory access times.

In Section 2.2 and Section 2.3we have discussed the various commercially available general purpose and high performance reconfigurable platforms. Finally a small comparison of the features provided by each platform was done. Section 2.4 explained the architecture of HTX platform. It is a good starting point to provide runtime partial reconfiguration support because, it has the RPF as a co-processor attached to a general purpose system with a high bandwidth, low latency link. It also runs on a Linux kernel and provides the ease of integration and usability. The hardware Wrapper provides various components for communication with the general purpose processor and the main memory. GCC compiler plug-in scans the source user C program for annotated functions and inserts corresponding system calls. The Linux device driver provides system calls to communicate with the FPGA device thereby making the approach for acceleration generic.

Finally in Section 2.5 we have explained the various partial reconfiguration solutions provided by Xilinx for its FPGAs. Partition-based partial reconfiguration is widely used when compared to the other approaches such as EAPR and XAPP290 because of the various advantages it provides. It gives almost total flexibility to choose the area and size for the reconfigurable module and addresses the signal boundaries between static and reconfigurable partitions automatically. The design and modification of the HTX platform to support Dynamic Partial Reconfiguration and acceleration of multiple software functions are described in the following chapters.

3 Hardware Support

Chapter 1 has identified few drawbacks of the existing HTX platform and Chapter 2 has described the HTX platform in detail. In this chapter the necessary architectural changes are made to address those issues. The design effort has mainly been channelized into three major parts. First, the Dynamic Partial Reconfiguration of the general purpose system to dynamically reconfigure the system on-demand through system calls. The main idea is to have a generic system to accelerate any software function on the fly on a general purpose platform. The next focus of the thesis is to provide support for accelerating multiple software functions simultaneously on the hardware. With DPR, when accelerating two software functions, we accelerate in hardware the first kernel, and perform a DPR and accelerate the second kernel in hardware. However, by providing hardware support for accelerating multiple software functions wherever possible, we can avoid the the reconfiguration time. Furthermore we also take advantage of the high-speed link to immediately read the data that has been written by another accelerator. This is beneficial especially while executing multiple streaming applications concurrently. The system thus provides necessary driver support for non-blocking execution of multiple functions. Finally, we evaluate the performance of the system by means of a beamforming audio processing application.

The ability of the system (FPGA) to reconfigure a part of it, while the rest of it is still operational is called as Dynamic Partial Reconfiguration (DPR). The time for DPR depends on the size of the region of the FPGA to be reconfigured. Moreover, granularity of the reconfigurable block also plays an important part in reconfiguration time. Granularity is defined as the size of the smallest functional unit (configurable logic block, CLB) that is addressed by the mapping tools. Higher granularity (fine-grained) architectures take more time to reconfigure than lower granularity (coarse-grained) because of the number of functional units and the routed data paths that has to be reconfigured.

Therefore there are two important considerations that has to be addressed when doing a partial reconfiguration of the FPGA. First being the mode of reconfiguration and the next is the size and the number of reconfigurable blocks. To address the former, we chose to use the Internal Configuration Access Port (ICAP) to reconfigure the FPGA because, it is a Xilinx primitive hardware and can be instantiated inside any design very easily. And it also provides the flexibility of choosing the data port width, thereby providing an efficient bit-parallel reconfiguration strategy. Furthermore, in this thesis, we fix the accelerator / Custom Computing Unit (CCU) to be the reconfigurable part because, it is just the functionality of the system that we aim to reconfigure at runtime. For this purpose an ICAP controller has been designed specifically to read the

bitstreams from the memory and feed it to the ICAP on-demand, using system calls.

Furthermore, to reduce the reconfiguration overhead when accelerating multiple functions, the platform also provides hardware support to accelerate multiple software functions concurrently. Another advantage of having multiple accelerators is that, the data produced by the first accelerator can be immediately be made available to the second. Thereby, we avoid totally the process of reconfiguration and restarting the the whole execution process. The data can also be immediately made available to the second accelerator, by local buffering of data into BRAMs. However this is not preferred for two reasons. First, storing huge amount of data locally is waste of resources. Second, we try to take advantage of the high-speed HTX link between the memory and FPGA. Moreover, the execution of the accelerators are either pipelined or sequential based on the data dependency between the functions that are accelerated. This is decided during the runtime by the arbiter in the hardware wrapper.

The reminder of the chapter is organized as follows. Section 3.1 describes about the ICAP controller, and the system calls that are involved in reconfiguring the system during runtime. It also describes the modifications needed in the driver, describes the partition pins which are needed to interface the static part and the reconfigurable part and finally a debugging mechanism to read out the configuration memory from the FPGA. Section 3.2 explains about the various hardware, software modifications to provide support for accelerating multiple functions concurrently. Finally, section 3.3 describes the beamforming audio processor used to evaluate the platform.

3.1 Dynamic Partial Self-Reconfiguration

In this section, we discuss the details of Dynamic Partial Reconfiguration of the HTX and a debugging mechanism to read back the configuration memory from the FPGA device. Xilinx provides several components and primitives such as JTAG, ICAP, SelectMAP to reconfigure the FPGA device. In this design we use Xilinx Internal Configuration Access Port (ICAP) to reconfigure the accelerator in the device on the fly. The reconfiguration is called Dynamic since rest of the device is still operational (i.e., components such as DMA unit, Interrupt Handler, Address Translation unit are still functional), to fetch the bitstream from the main memory and feed it to the ICAP. Partial because, we reconfigure only a part of the device. Self, as the device uses its own primitives (ICAP) to reconfigure itself. Hence the name Dynamic Partial Self-Reconfiguration (DPSR) as the definition suggest. In this design we restrict the reconfigurable module to the accelerator, although theoretically any portion of the FPGA can be reconfigured on the fly.

The ICAP Interface is a stripped-down version of the SelectMAP interface [4] [28] of the FPGA in slave mode. The ICAP allows the programmer to read and write into the configuration memory of the FPGA thus allowing self-reconfiguration of the FPGA device. The ICAP interface in Virtex-4 has a 32-bit (4 byte) data ports. Furthermore, we can also configure the data port to be 8-bit (1 byte) wide. However, internally the device will treat it as 4 1-bytes. Special care must be taken to invert the input bits if

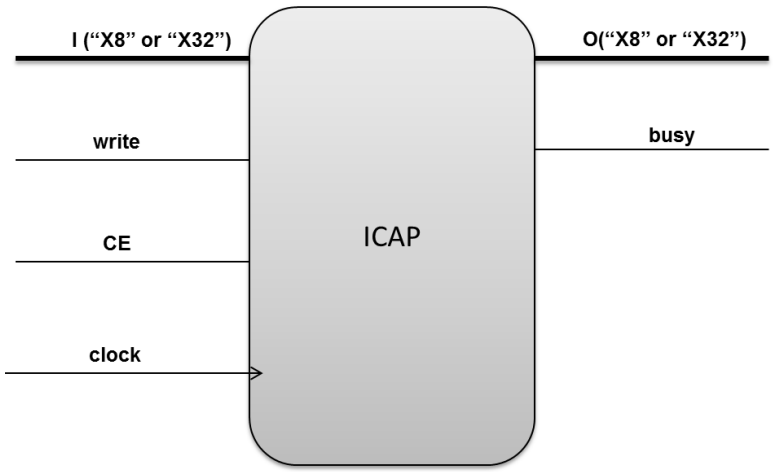

Figure 3.1: Xilinx ICAP primitive

using the 1 byte version of the port in Virtex-4 devices. ICAP has a write and chip enables, a busy signal and a clock inputs. When configuring a certain device via ICAP the user must make sure not to reconfigure the circuitry that controls the ICAP. Figure 3.1 shows a Xilinx ICAP primitive. Details on the Internal Configuration Access Port (ICAP) interface can be found in [4] and [28].

3.1.1 ICAP Controller

In order to fetch the bitstream from the main memory and feed the data to the ICAP, the design has an ICAP controller designed in the hardware wrapper. Figure 3.2 shows the architecture of the system with the ICAP controller.

The Dynamic Partial Reconfiguration of the FPGA device is initiated by the user program. The user program has a call to the wrapper function *molen_set("filename")*. This corresponds to the *ioctl* system call in the device driver with a user defined attribute *SET* i.e. *ioctl(dev,SET,arg)*. The driver then writes a SET command into the memory mapped region of the device. When the control unit detects SET command it asserts a *start_op* signal to the ICAP Controller for 1 cycle. In the meanwhile, the *molen_set()* wrapper function creates a memory map file using *mmap()* function and passes a pointer to *ioctl* call. The *mmap()* function shall establish a mapping between a process' address space and a file. During the *ioctl* call, the driver will also write the starting address of the bit stream to a control register of the FPGA device, which will be later read by the ICAP controller for the starting address of the bitstream. The size of the bitstream is embedded into its header. The first word of the bitstream has the size of the bitstream. Henceforth, the ICAP controller keeps track of the size until the entire bitstream has been read from the memory. During this process, the ICAP controller feeds the ICAP with the bitstream word by word (32 bits) and initiates partial self-reconfiguration. Once all the information has been read, the FPGA notifies

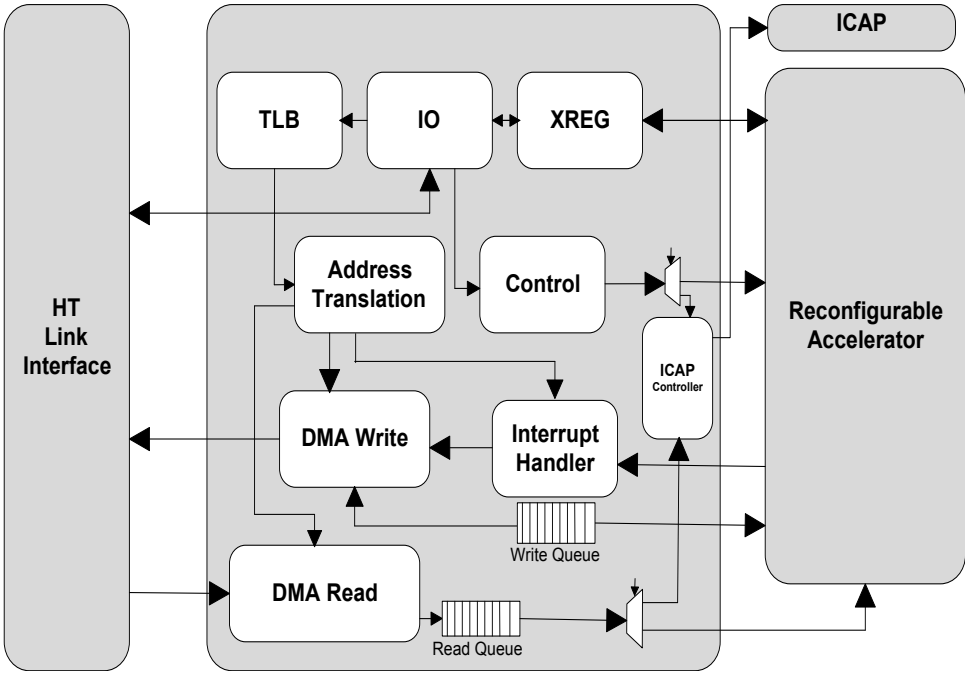

Figure 3.2: Architecture with ICAP controller

the driver by raising an interrupt, which indicates that that the function is available for execution. The wrapper function to initiate DPR is given in Appendix B.

ICAP controller is internally a state machine. Figure 3.3 shows the FSM states of the ICAP controller. The various states are explained as follows:

- **s_idle:** Remains in the idle state until control unit signals the start of execution;
- **s_req1:** Request the starting address of the bitstream;
- **s_wait1:** Calculate the bitstream size from the bitstream header;
- **s_req2:** Request for the rest of the bitstream from the memory;
- **s_wait2:** Wait for data to arrive in the FIFO;
- **s_feed_high:**Feed the upper 32 bits to the ICAP;
- **s_feed_low:** Feed the lower 32 bits to the ICAP;
- **s_done:** Final state when all the data has been fed to the ICAP.

3.1.2 Device Driver Support

IOCTL system call is used in places where the driver need to perform actions more than just read/write, such as lock a process, eject a media, report error, find baud rate etc. In this design, *IOCTL* system call is used in the device driver to support

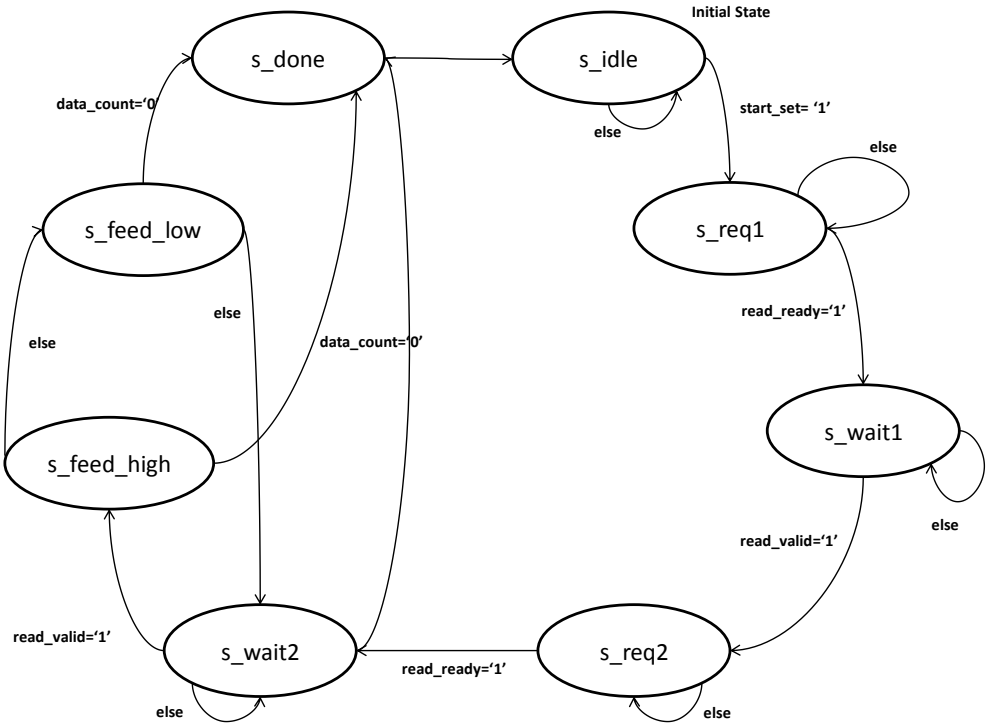

Figure 3.3: ICAP Controller FSM

partial reconfiguration. This implementation supports a special command for partial reconfiguration, the *HTEX_IOSET* command. The driver writes a SET command to the second array location of the second memory mapped region as explained in Section 2.4.2.6. It also puts the calling process to sleep until it receives an interrupt from the device after reconfiguration completion. Furthermore, when the process sleeps, it is put into a wait state which is interruptible, so that the user can cancel the reconfiguration at any point. It is useful for debugging the hardware design, since the controller can enter an infinite loop resulting in the application hanging indefinitely. Further information about prefetching the bitstream, linking the bitstream to the executable, driver support for non-blocking execution during partial reconfiguration, and generating bitstreams for partial reconfiguration can be found in [29].

3.1.3 Partition Pins

Throughout the thesis, PlanAhead tool was used to generate partial bitstreams. Various Partial Reconfiguration terminologies has already been described in Section 2.5. This section describes in detail about the signal interfacing between static and reconfigurable partitions. When performing partial reconfiguration, the design is split into two parts namely static and reconfigurable regions. The static part remains active throughout, and so does the interface between the static and reconfigurable part. The only thing that is allowed to change is the reconfigurable part. Although the designer is free to

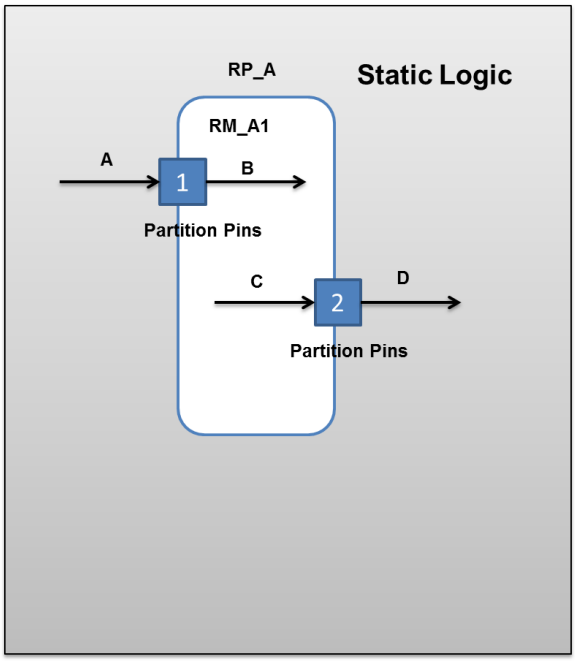

Figure 3.4: Static and Reconfigurable logics with Partition Pins inserted

program any arbitrary design for the reconfigurable partition, the interfaces to the static part has to be consistent for the design to work properly, failing which the design might crash. Xilinx provides special interfaces called Partition Pins which will route the signals between the two regions and remain active throughout and also during reconfiguration.

Partition Pins are special components at the port boundary between static logic and reconfigurable logic. They are required to guarantee that the circuit connections between the static logic and different Reconfigurable Modules (RM) for each Reconfigurable Partition(RP) are identical. The Partition Pins can also be used to set the timing constraints over the nets that pass between the static and the reconfigurable logic. Partition Pin can take one of several forms depending on path and the direction of the signal as given in Figure 3.4. Even though Partition Pins are physically located within the reconfigurable regions, they are logically part of the static logic. Furthermore, Partition Pins can be input or output connections to a reconfigurable region and cannot be bidirectional.

- **A)** Static net input to a Partition Pin;
- **B)** Reconfigurable net output of a Partition Pin;
- **C)** Reconfigurable net input to a Partition Pin;
- **D)** Static net output of a Partition Pin.

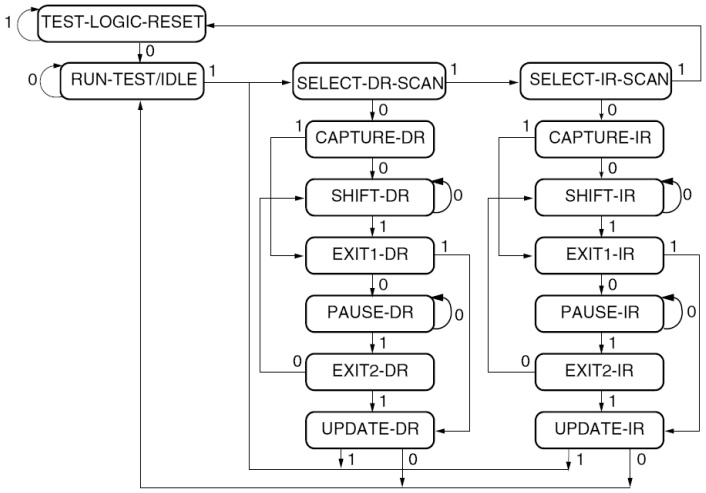

Figure 3.5: JTAG TAP Controller State Diagram. Source Xilinx configuration guide [4]

3.1.4 Configuration memory read back

There are several ways to debug a design after programming it on the FPGA. One of the methods is to read back the configuration memory from the FPGA. Configuration memory is the bit information that has been used to program the FPGA device. By comparing the information obtained from the device, with the original bitstream that has been used to program the device, we can find the missing programing information. Although, the configuration memory thus obtained is not a direct way of debugging, we can use the information to investigate the missing programming information by further investigating the design or by using post-synthesis debugging tools such as ChipscopePro.

For this purpose we use a Serial Vector Format (SVF) file to push the configuration memory out via the JTAG port data registers. SVF is an industry standard file format that is used to describe JTAG chain operations in a compact, portable fashion. SVF files are used to record JTAG operations by describing the information that needs to be shifted into the device chain. All JTAG operations are controlled through Test Access Port (TAP) of the device. The TAP consists of four signals: TMS, TDI, TDO, and TCK. These signals interact with the device through the TAP Controller, a 16-state finite state machine as shown in Figure 3.5.

From any start state, holding TMS High for at least five TCK cycles (five state transitions) leaves the TAP in the Test-Logic-Reset state. All JTAG operations shift data into or out of JTAG instruction and data registers. The TAP Controller provides direct access to all of these registers. There are two classes of JTAG registers: the Instruction register (only one) and Data registers (many). Access to the Instruction Register is provided through the Shift-IR state, while access to the Data Register is provided through the Shift-DR state. To shift data through these registers, the TAP Controller of the target device must be moved to the corresponding state. For example,

to shift data into the Instruction Register, the TAP Controller must be moved to the Shift-IR state, and the data shifted in.

There are several SVF commands. However, the basic idea of most of the commands is as follows. Every command specifies the number of bits to be shifted to the instruction/data register and a scan pattern to be applied to the register. The command also has a data pattern that is expected (mask data) to be pushed out of the data registers when the pattern is applied. If the pattern matches, it moves on to the next instruction, if not, it throws an error and shows the actual data that is pushed out to the data register. This data can be compared with the original bitstream, to find the match between original bitstream and data obtained from the device. It is also worth mentioning that, the data that is pushed out of the JTAG port of the device is nibble swapped and bit reversed. In this way we have the configuration memory from the device. Appendix A lists a small SVF file used to read back the configuration memory from the device after programming the FPGA.

3.2 Multiple Accelerators

This section describes the hardware, driver and compiler support for accelerating multiple software functions concurrently on the hardware. 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 non-blocking execution of multiple software functions, the exact number depending on the number of accelerators supported by the FPGA. This thesis serves as a starting point to accelerate two software functions of a single process simultaneously in combination with non-blocking execution. Allowing multiple accelerators on the same reconfigurable device presents the hardware with some issues, the main ones being:

- How do multiple accelerators access main memory?
- How to schedule accelerators execution in the hardware?

The interface from the wrapper to the reconfigurable accelerator is based on that of the HTX [18] system. Now that there are two accelerators, implemented to accelerate different kernels simultaneously, there has to be a proper arbiter in place for the CCUs to access the resources of the hardware wrapper. For this purpose, all the buses and signals from both the accelerators go through an arbiter as shown in Figure 3.6. Furthermore, there are also signals from the accelerators to the arbiter, to synchronize and pipeline their execution phases. The arbiter decides which accelerator should get the access of the wrapper resources dynamically during execution, based on the following factors. Also the accelerators are implemented based on the following criteria:

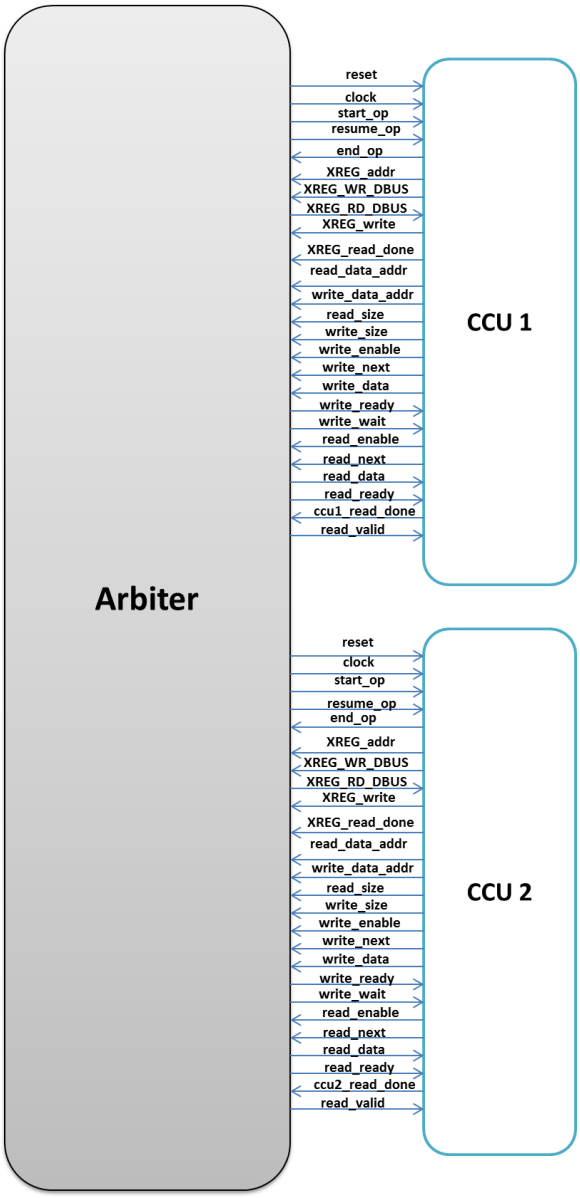

Figure 3.6: Multiple accelerators with arbiter

- The first software function that is annotated for acceleration will be first on the hardware, and the subsequently annotated software function will be the second;
- The execution phases of the accelerators have been identified into three pipeline stages namely, read, execute, write. Moreover, the read phase comprises of memory reads and XREG reads and write phase consists of memory writes and writes to XREGs;
- When the host processor writes to the second memory mapped IO region to signal the start of execution, a command and data packet arrives on the input queue.

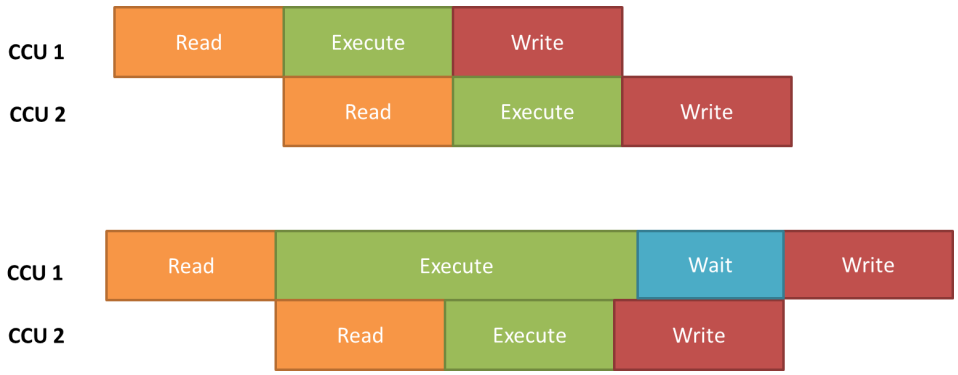

Figure 3.7: Execution phases of CCUs without data dependencies

Since we have two accelerators , the host driver writes into the memory mapped IO two times, to indicate the start of both the accelerators. Internally, the control unit signals the start of the accelerators to the arbiter;

- When the arbiter receives the start signal for the first CCU, it gives the resource access for reads for the first CCU. In the meanwhile, the control unit signals the arbiter to start the second accelerator. The arbiter checks to see if the first accelerator is still accessing the resources for reads, if it does, it makes the second wait until the first completes all the reads;

- Once the accelerators have completed the read phase it goes into execution phase. Upon completion of execution phases, the arbiter gives the write resources to whichever accelerator makes a request first. In this way, arbiter makes sure that writes happen on a First Come First Serve(FCFS) basis;

- Finally, once both the accelerators have signaled the execution completion, the arbiter detects them and signals the interrupt handler(IH) once in the end. The IH raises an interrupt finally to signal the execution completion to the driver. Thus there is a pipelined order of execution as shown in Figure 3.7;

- Furthermore, the arbiter also checks for data dependency between both the accelerators by comparing the write addresses of the first accelerator with the read addresses of the second accelerator. If it detects any dependencies it switches to sequential execution of the accelerators as shown in Figure 3.8. Thus the main task of the arbiter is to manage the access of resources and to schedule the execution of the accelerators based on dependencies. It is also noteworthy that, since the granularity of reads/writes of each accelerator is coarse (i.e. each accelerator completes all its reads/ writes before the read/write of another starts), it is sufficient to check for the address dependencies between writes of first accelerator and reads of second accelerator;

- Finally, since the hardware design has two accelerators, it is essential that two corresponding software functions are annotated on the software in the proper se-

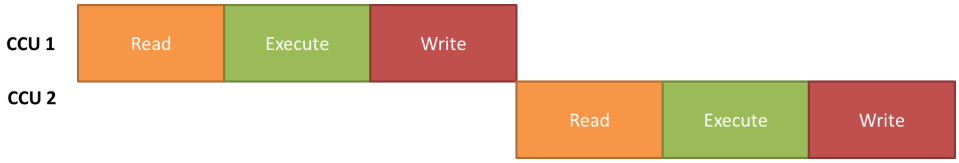

Figure 3.8: Execution phases of CCUs with data dependencies

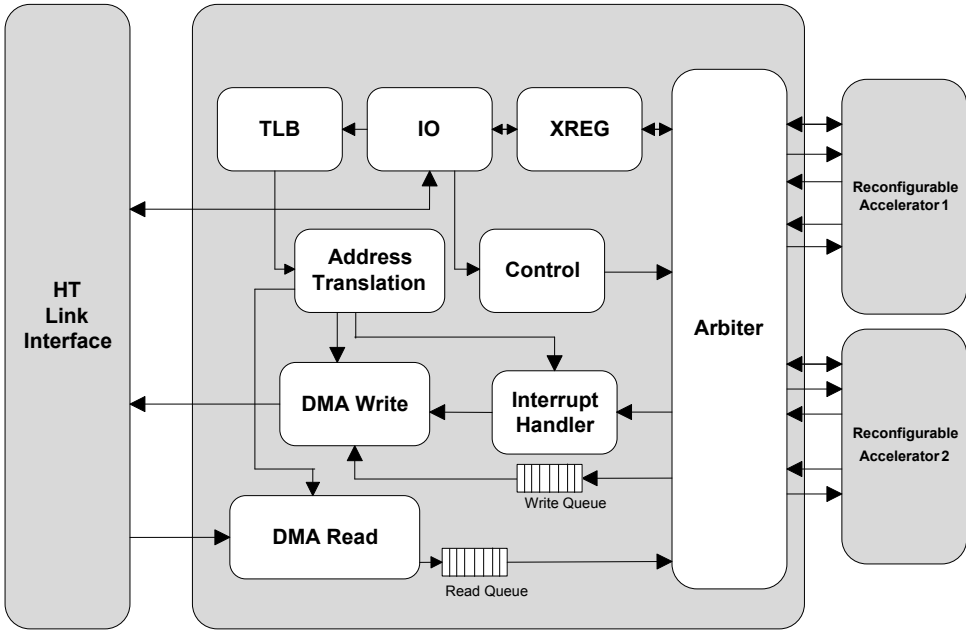

Figure 3.9: Wrapper with multiple accelerators

quence in order for the system to produce the correct results. Failing which, the system might produce unexpected outputs from the accelerators.

3.2.1 Wrapper Support

The various components in the hardware wrapper that helps in the integration of the FPGA in the system has been explained in Section 2.4.1. However as mentioned before, running multiple accelerators requires proper management of resources. Therefore, we have included an arbiter module inside the wrapper. Figure 3.9 shows the overall architecture of the system after inclusion of the arbiter.

3.2.1.1 The arbiter

The arbiter module is responsible for three main tasks. First, it is responsible for managing the resources in the static region. Since we do not have separate resources for each accelerator in the static part, we need to share the resources between the accelerators equally making sure that neither of them doesn't have to wait too long for any resource. Next, the abiter is responsible for detecting any address dependencies between accelerators. The accelerators receive their read / write addresses during the read phase as

explained in 3.2. After this phase, the arbiter checks for data dependencies, by comparing the write address of CCU1 and read address of CCU2. Depending on which their execution is scheduled to be either sequential or pipelined as explained above. Finally, the arbiter is also responsible for creating a synchronization between the accelerators and making sure that the reads of CCU2 never occurs before the writes of CCU1 when where is a data dependency. This kind of check is very useful when accelerating two streaming applications simultaneously with TLB locks as explained in Section 3.2.1.3.

3.2.1.2 Control Unit

The memory mapped IO regions has been explained in Section 2.4.2.6. In case of multiple parallel accelerators, the read-write unit receives multiple *EXECUTE* commands from the driver, one for each accelerator. The control unit in this case, when it receives multiple *EXECUTE* commands in address 0, it signals the arbiter for one cycle, to start a corresponding CCU. The arbiter then asserts the start signal for an accelerator. The control unit is a state machine, which has a counter to keep track of the number of execute commands. Depending on the value of the counter, it signals the start of a corresponding accelerator. After signaling the start of both the accelerators, it resets the counter. However, due to some unknown reasons, if it does not receive the second execute command it resets the counter. It is decided upon the earliest of the two signals i.e the second execute command or the end_op signal of the first accelerator. Since the design currently has only two accelerators, the control unit receives execute commands only for two CCUs. However, if it was to receive additional execute commands, the signal is simple discarded. Nevertheless, the design is always scalable to support additional accelerators.

3.2.1.3 TLB Support

Section 2.4.2.3 describes how TLBs are used by the device to translate virtual address to physical address. This section describes an enhanced TLB support for streaming applications. It has already been explained in Section 3.2 that the arbiter schedules the execution of the accelerators sequentially, whenever there is a data dependency between them. In these cases, instead of making their execution completely sequential, it is still possible to pipeline the execution of the CCUs. The main idea is as follows. CCU 1 writes data which has to be read by CCU 2 immediately. Therefore, the arbiter prefetches the read addresses of CCU 2 (write addresses of CCU 1 as both are the same) and places a lock on them to avoid CCU 2 reading the address locations until they have been updated by CCU 1. Once the first CCU starts writing to the address locations, the locks on the address locations are removed and the arbiter is notified. The arbiter then gives read access to CCU 2 for those locations. Therefore CCU 2 reads the updates values as and when CCU 1 writes. Furthermore, we also have a synchronization mechanism such that, the reads of CCU 2 never occurs before the writes of CCU 1 because, we have a lock over the read addresses for CCU 1. So, if a CCU encounters a lock on an address, it registers the address with the address translation unit, notifies the arbiter, and waits until the lock has been released by the other CCU. When the lock has been released by another accelerator, it checks if there are any outstanding addresses registered. If

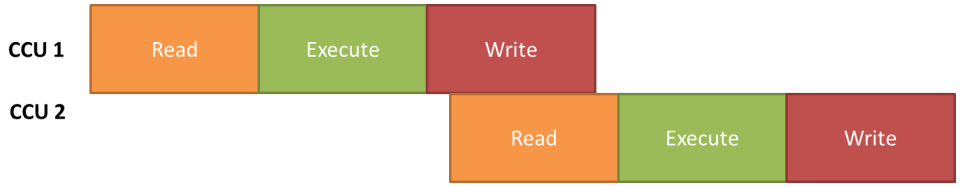

Figure 3.10: Pipelined sequential execution with TLB Support

so, it notifies the arbiter that the lock has been released. The arbiter then notifies the accelerator that is waiting. After which it can again continue to read. The case explained in sequential execution as shown in Figure 3.8 becomes like the one shown in Figure 3.10.

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 other flags such as read and write protection. After the CCUs get the read and write addresses, the arbiter checks the addresses for data dependencies. If there are dependencies, the arbiter schedules the sequential execution of the accelerators. The arbiter also prefetches the TLB entries with the write addresses as explained in the next paragraph. The reason for doing this is as follows. The TLB entries are filled up by the driver whenever there is an address translation interrupt. However, address translation occurs only during read / write phases. This means that, accelerators will get a TLB entry only when an address is to be read or written. Hence it is not possible place locks in advance and pipeline the sequential execution.

When the arbiter detects data dependency, it schedules the prefetch of the TLB entries. The arbiter sets the address and size signals to the starting address and the total number of 64 bit values being written and asserts the prefetch signal to a TLB prefetch unit. The TLB prefetch unit calculates the number of pages and translates the virtual address into physical addresses using the address translation unit as explained in Section 2.4.2.3. If it finds a physical address entry already, it places a two bit lock. One to indicate the CCU ID and the other to indicate if the TLB is locked or unlocked. If there are no entries found, it notifies the interrupt handler to raise an address translation interrupt. In software, driver's interrupt handler work queue determines that there is an address translation request. Next the address is also read from the device and is then translated, and the result is written to the TLB. Once the TLB entry is written, the TLB prefetch unit places a 2 bit lock over it. We use a two bit lock because we have only two accelerators. Although theoretically the number of bits can be extended according to the number of accelerators in the design. In this way, we can prefill the TLB, place locks, synchronize the phases of CCUs, and pipeline the execution of the CCUs.

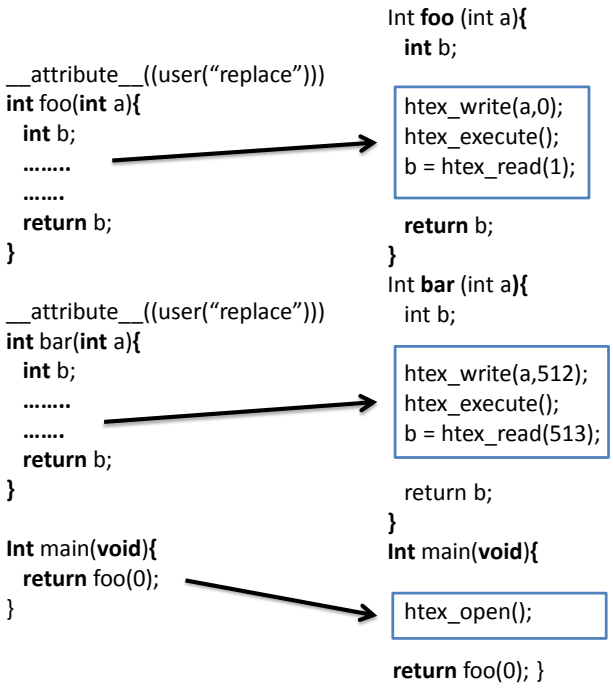

Figure 3.11: Proper arguments inserted to htex_write() wrapper function

3.2.2 Compiler Support

The compiler support for the HTX platform has been explained in Section 2.4.2.2. Whenever the replacement pass is called, the compiler plug-in replaces body of the function with the following wrapper functions as shown in Listing 3.1. When providing compiler support for accelerating multiple functions, the wrapper function of concern is the *htx_write(uint64_t arg , uint64_t index)*. Where *arg* is the argument to be written and *index* is the location in the XREG where the argument has to be written to. Every time replacement pass executes, the compiler plug-in inserts the the function with a reference to the same index location. Therefore, if there are N annotated functions, the same locations in the XREGs gets overwritten every time.

```
uint 64_t htx_read(uint64_t index) ;
void htx_write(uint64_t arg , uint64_t index ) ;
void htx_execute( );
```

Listing 3.1: Function prototypes of wrapper functions inserted by GCC plug-in

In order to avoid the same locations being overwritten when the source file has multiple annotated functions, the compiler plug-in scans the number of annotated functions in the source program, and inserts the system calls accordingly. Figure 3.11 shows the compiler replacing the body of the function with calls to wrapper functions.

3.2.3 Driver Support

During the the execution of the accelerator, the driver provides support for address translation, memory protection and memory paging by putting the calling process into sleep. By doing so, driver marks the process in a special state, removes it from the scheduler's run queue and adds it to a special queue called wait queue. It remains in the state until a certain event occurs. During this, the driver also pins down the relevant user pages in the memory so that they do not get swapped to the disk. Therefore, when the driver encounters the first *ioctl()* system call in the running process, it puts the process to sleep. The above properties are implemented in the device driver using the following kernel functions. The function *get_user_pages()* provides memory protection and paging. In addition, *get_user_pages()* locks all pages used by the device in memory. The function *pci_map_page()* is used by the driver for translating a virtual address and maintaining the host-cache coherent. Finally the process is put to sleep by the function*wait_event_interruptible(wait_queue, wait_for_event != 0)*.

However, this type of blocking execution will not be suitable when accelerating multiple functions concurrently. During the acceleration of multiple software functions, the process is not put to sleep when the driver encounters of the first *ioctl()* system call. This is due to the fact that, once the process is put to sleep, the flow will not reach the second function and the device will never obtain the parameters of the second function to be accelerated. It is also note worthy that, never putting the process to sleep makes the addresses translation and extremely difficult for the driver since the user pages cannot be locked in memory and therefore address translation fails in the driver. Therefore when the first function call is encountered the driver continues execution and the process is put to sleep only when the second function call is encountered. During this time driver has to ensure that the any address interrupts from the device are still served.

The driver will start serving interrupts from the device for address translations after the process goes to sleep state. However, to provide physical address entries until the process sleeps, the driver pre-fills the TLB of the device with (physical) address entries from the process's virtual address space by converting them into physical entries. This is done by locking process's virtual address space in RAM using *mlock()* system call and passing its pointer to the driver. This makes sure that the process's address space does not get swapped to disk and make sure the entries supplied are still valid. The driver then starts translating the virtual addresses to the physical addresses and writes them to the TLB. This is done before the device starts the execution of the first accelerator. Therefore, the device will already have the physical addresses for the address translation request from the first accelerator which has just started execution. This means that, address translation requests will never be raised until the process sleeps. Listing 3.2 shows the driver module that pre-fills the TLB with the physical addresses using the stack and heap entires of the process.

```
void prefill_tlb(struct htex_dev_t *htex_dev)
{
  int heap_count, stack_count;
  unsigned long start_heap, start_stack, end_heap;

  //Prefilling tlb with stack and heap addresses
  struct mm_struct *mm = htex_dev->tsk->mm;
  start_heap = mm->start_brk;
  end_heap = mm->brk;
  start_stack = mm->start_stack;
  stack_count = mm->stack_vm;

  //prefill stack
  start_stack = start_stack & PAGE_MASK;
  start_stack = start_stack - ((stack_count-1)*PAGE_SIZE);
  translate_range(start_stack, stack_count, htex_dev);

  //prefill heap
  heap_count = (end_heap - start_heap) >> PAGE_SHIFT;
  translate_range(start_heap, heap_count, htex_dev);
}
```

Listing 3.2: Driver Support for non-blocking execution

3.2.4 Memory Consistency

When multiple accelerators read and write simultaneously, it is important that we provide memory consistency. Meaning that, when reads or writes of an accelerator follows reads or writes of another accelerator, it is necessary to make sure that the individual accelerators get only the data that it had requested for. In other words, the requests and responses of accelerators should not get mixed up. This type of consistency is ensured as follows.

First, the HT core in the device provides separate queues to handle read/write requests. Furthermore, we also have separate queues to handle the data packets of reads and writes in the wrapper. Therefore, read and write requests/responses of different accelerators never gets mixed up. However, the next point to be addressed is, read request/responses or write requests from different accelerators should be consistent.

When there are read requests from the first accelerator followed by read requests from the second accelerator, consistency can be addressed in two ways i.e. either by coupling or decoupling the request and response. It is note worthy that, although theoretically read-read is not a problem, since we are expecting a response packet for each read request, we have to make sure that response packets for different accelerators do not get mixed up. By decoupling the request and response, we need tags in the DMA unit to id each request packet, and push the response packets into individual read queues of each accelerator. Moreover, giving an ID for each request packet becomes more complicated because, this requires an extensive restructuring of the command packets, since the design already uses the maximum 96 bit command packet size. However, this

CCU1	CCU2	Consistency
Read	Read	Coupling requests/responses from each accelerator
Read	Write	Different queues in HT core and individual FIFOs in Wrapper
Write	Read	Different queues in HT core and individual FIFOs in Wrapper
Write	Write	Grouping requests from each accelerator and performing sequentially

Table 3.1: Memory Consistency for request/response from different CCUs

might also have an overhead in the DMA unit for every response packet received since we have to compare the packets with tags and push it in the corresponding queues. Therefore, we address read-read consistency by coupling the requests and responses. Read requests are made and the response packets are pushed in a single read queue until the accelerator that requested the packets reads them from the queue.

For write requests followed by write requests from different accelerators there are no response packets involved. Therefore, the writes of individual accelerators are grouped together and done one after another sequentially. By this way, we solve the read-write consistency issue from different accelerators. Table 3.1 summarizes the above.

3.2.5 Partial Reconfiguration

Partial reconfiguration support can be added to the design with multiple CCUs can be added. This can be done by using PlanAhead tool to generate partial bitstreams. The design must be split into static and dynamic parts, with partition pins 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. Reconfigurable partition is assigned by setting the accelerator as the reconfigurable module (RM) in the PlanAhead design project. In this case, there are multiple RPs and RMs since the design has two accelerators. The partition pins along with the enable signals for each pin is 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.

3.3 Beamforming Audio Processor

The application that was used to test the performance of our system was a beamforming audio processor as proposed in [30]. However, we have adapted the beamforming processor to suit the architecture of the system. Beamforming is a signal processing technique that improves the signal strength received from a specific location. It is already used for many years in areas of RADAR, SONAR etc. Over the last few years it has been adopted by audio research society to enhance speech recognition. The audio domain uses the microphone (instead of antennas in signal processing domain) array that captures the audio environment. Generally, there are two different types of

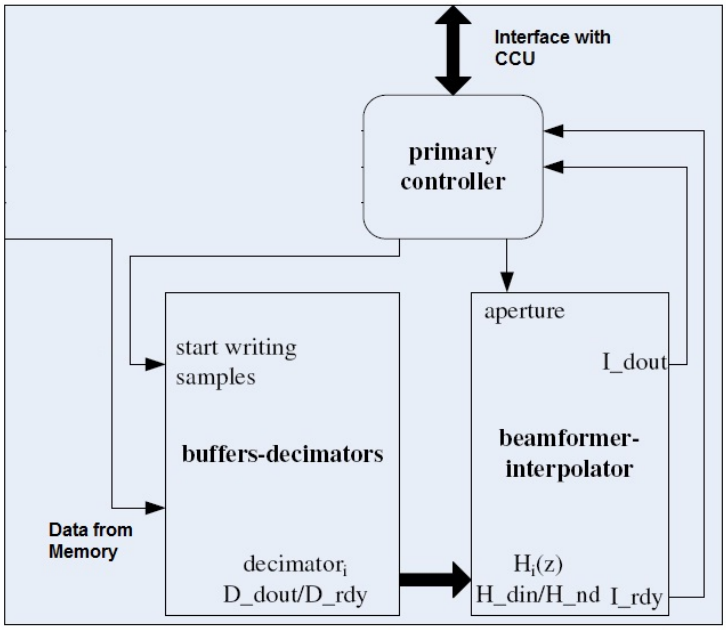

Figure 3.12: Beamforming Fabric Co-processor (BF_FCP) Organization

Beamforming, non-adaptive (or time-invariant or non-blind) and adaptive (or blind) [31], [32]. Non- adaptive methods are based on the fact that the spatial environment is already known and tracking devices are used to enhance speech recognition. In contrast, adaptive approaches do not utilize tracking devices to locate the sound source. The use the received signals from the microphones to calibrate the beamformer. The adaptive Beamforming techniques do not converge fast enough and requires more processing power and hardware. In this thesis we use a non-adaptive beamformer, that can be used in small hand-held devices and complete 3-D audio systems. All possible filter coefficients are pre-calculated [33] and stored in an on-chip memory to save hardware and processing power. We also further simplify the design by storing the input samples received from a microphone source in the main memory, which is later read through the HTX bus.

3.3.1 The Design

The kernel is a Beamforming Fabric co-processor module (BF_FCM) which internally has a primary controller and two other modules namely buffers decimator (BD) and Beamforming interpolator (BI) as shown in Figure 3.12. The primary controller (C1) is a FSM. It initiates the reading and writing of the input samples from and to the main memory. It reads input samples from 16 input channels (1024x16 bit inputs for each channel) and stores them into the internal buffers. Once the input is stored in the internal buffers all the processing happens concurrently for all input channels. The inputs are sampled from the source at 48 KHz.

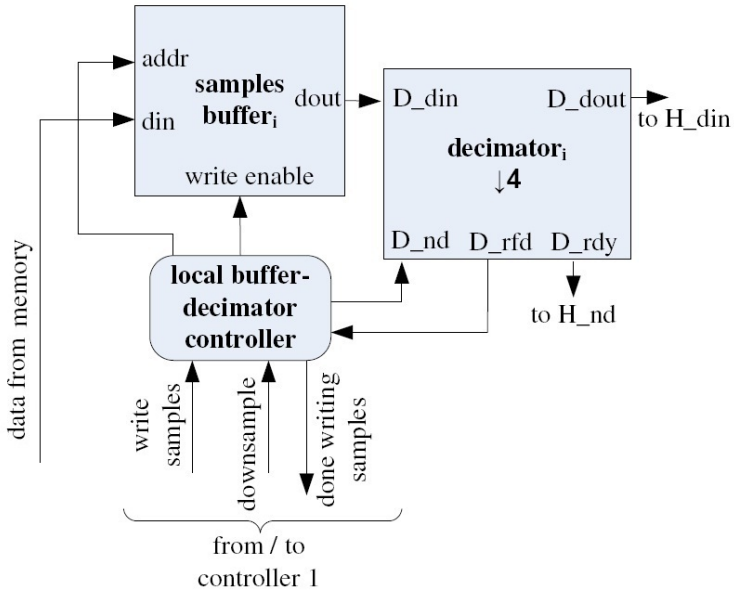

Figure 3.13: Buffer-decimator module

3.3.2 Buffers Decimator (BD)

Each input channel is down sampled from 48 KHz to 12 KHz. For this reason, the buffer decimator (BD) module consists of the samples buffer (SB), a decimator by 4 and the local buffer-decimator controller (BDC). The latter is responsible for storing 1024 16-bit signed audio samples to the SB that was captured by the corresponding microphone. Figure 3.13 shows a Buffers Decimator module with its Sample Buffers (SB) and Figure 3.14 shows an external controller with Buffers Decimator modules. External controller (C1) initiates the copying process to SB. Once samples are copied, the BDC acknowledges C1. The C1 instructs BD to start downsampling. The downsampling is performed concurrently for all the 16 input channels. Once decimator generates the output it is forwarded to the corresponding H(z) beamsteering FIR Filter.

3.3.3 H(z) beemsteering FIR filter

Figure 3.15 shows the H (z) beemsteering FIR filter. It takes as input the aperture value recorded from the source tracking device, and the signals from the decimator. The FIR filter uses the appropriate set among N available 128-coefficient sets to filter the down sampled signals. The filtered signal is forwarded to the external controller2 (C2) and acknowledges that new data is available.

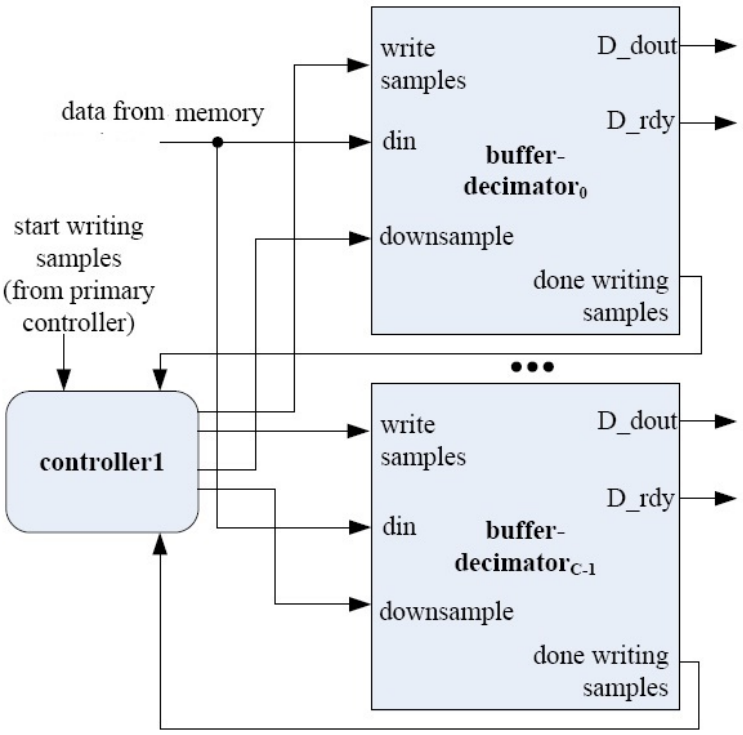

Figure 3.14: Controller1 and buffer-decimator (BD) modules

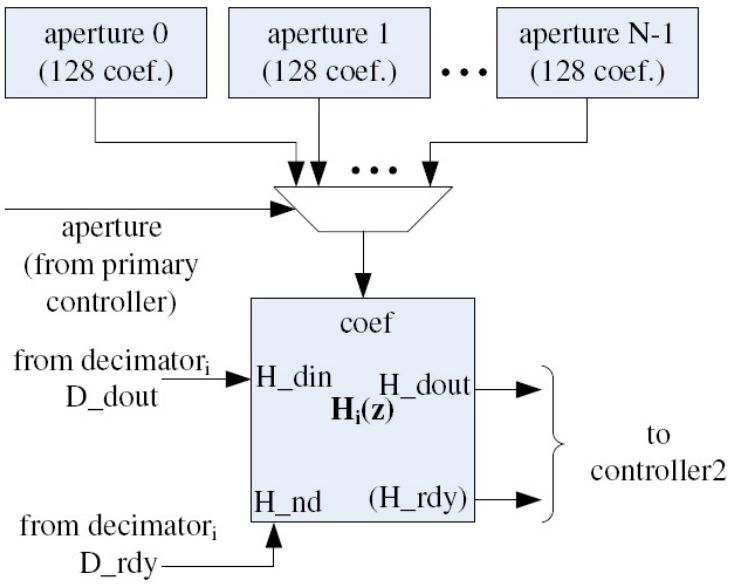

Figure 3.15: A beamsteering FIR filter

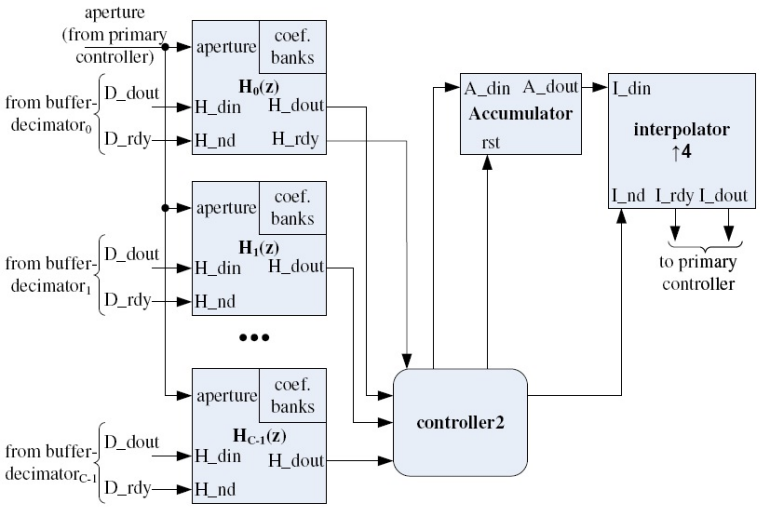

Figure 3.16: Controller2 and the accumulator

3.3.4 Beamforming Interpolator (BI)

Once the source is extracted from the filtered signals, it is again up sampled from 12 kHz to 48Khz. Based on that Beamforming Interpolator integrates all outputs from FIR filters, an accumulator and an interpolator. Beamforming Interpolator has been shown in Figure 3.16. The controller C2 can connect C number of H(z) FIR filter signals that process concurrently the down sampled signals. All FIR outputs are forwarded to the accumulator. Once all samples are accumulated, C2 forwards the results to the interpolator. Which in turn acknowledges the primary controller and the up sampled signal is stored back on the main memory through external primary controller (C1).

3.4 Conclusion

This chapter has described the details of Dynamic Partial Reconfiguration, having multiple accelerators and an audio beamforming application. Section 3.1 has discussed the details of the hardware ICAP controller that has been designed to read bitstreams from the memory and feed it to the ICAP, and thus initiating DPR. Further, we discus briefly about the driver support and how DPR is initiated through a user application on-demand. Finally it describes about the Partition Pin interface between the static and reconfigurable hardware areas.

Section 3.2 has presented the various wrapper modifications required to support multiple accelerators in the design. It also described the various mechanisms provided by the arbiter to schedule the execution of the accelerators depending on the data dependencies. Further, we also presented a TLB locking mechanism for streaming applications to immediately consume the data that has been produced. Next it describes the device

driver support for non-blocking execution of multiple software functions of a same process. It concluded with compiler changes to insert wrapper calls properly in the source program when annotating multiple software functions to be accelerated. Finally section 3.3 described the architecture of a beamforming audio processor that is used to evaluate the design. We have discussed the main blocks, various stages of processing and the data flow involved in the beamforming audio processor.

4 Evaluation

This chapter evaluates the proposed system using various performance metrics. During the initial stages, an array copy kernel was used to test the DPR functionality. A minimal kernel was used to keep the size of the bitstream as small as possible. Different debugging mechanisms were used to make it functional on the HTX platform. Once DPR was fully functional, a beamforming audio processor as described in [30] was ported into the design and was used to evaluate the performance. Furthermore, the platform is evaluated using a number of other criteria such as implementation area, speedup compared to software, reconfiguration latency, execution time and power as described below.

The Chapter is further organized as follows. Section 4.1 evaluates the results of the proposed system in terms of total area and resource consumption on the FPGA for both static and reconfigurable logic. Section 4.2 measures the execution times of different applications on the hardware, and calculates the speedup achieved compared to the software implementation. Next, Section 4.3 evaluates the overhead of DPR reconfiguration and the latency caused due to different bitstream sizes. Section 4.4 shows the final floorplan of the design with various modules locked to specific regions on the FPGA. Finally, Section 4.6 shows some statistics about power that is consumed by the design with single and multiple accelerators.

4.1 Area Utilization

This section presents the resource requirements for the implementation of the design with a single and multiple CCUs on a Xilinx Virtex-4 FPGA. Table 4.1 offers the area requirements of the FPGA modules with single CCU. The HTX interface occupies about 5,000 slices and 36 BRAMs; that is due to the multiple queues needed for the communication with the HyperTransport bus and the tables required to keep track of the ongoing transactions. The wrapper needs more than 2,500 slices and 69 BRAMs; it is worth noting that address translation and DMA manager occupies the largest part of the wrapper, mostly due to its large queues and memory blocks that are needed for storing the TLB entires for address translation. The maximum size of the PR region consists of 10,000 slices and about half of the BRAMs in the FPGA (144 BRAMs). The Audio processing accelerator requires 9,000 slices and 143 BRAMs due to the logic that buffers the input data before processing it, while the AES accelerator covers 5,000 slices.

Table 4.2 shows the area requirement of the design with multiple CCUs. The resource consumption of the HTX link interface remains the same while the wrapper uses 4,500 slices due to the logic utilized by the arbiter. There are separate PR partitions defined for

	Slices	FFs	LUTs	BRAMs	ICAP
HTX Interface	5,369	4,961	7,347	36	0
Wrapper	2,519	1,426	4,007	69	1
Max PR Region	10,368	20,736	20,736	144	0
Audio Design	9,247	10,490	11,123	143	0
AES Design	5,225	2,010	9,196	0	0
Total	18,256	27,123	32,090	249	1

Table 4.1: Area Utilization of the FPGA Design Blocks with single CCU

	Slices	FFs	LUTs	BRAMs	ICAP
HTX Interface	5,369	4,961	7,347	36	0
Wrapper	4,519	6,227	9,606	69	1
Max PR Region-Audio CCU	10,368	20,736	20,736	144	0
Audio Design	9,247	10,490	11,123	143	0
Max PR Region-AES CCU	5,624	11,248	11,248	30	0
AES Design	5,225	2,010	8851	0	0
Total	25,880	43,172	46,937	279	1

Table 4.2: Area Utilization of the FPGA Design Blocks with multiple CCUs

both the accelerators in the FPGA. The audio CCU uses the same amount of resources as in the previous case. However, the maximum PR region of AES occupies about 5,500 slices and 30 block RAMs while the accelerator itself consumes about 5,225 slices and no block memories.

4.2 Speedup and execution times

This section measures the performance of the design with single and multiple accelerators. The speedup and execution times obtained are from two applications (in three different scenarios). First an audio kernel is accelerated in hardware and the execution time and speedup is compared with that of pure software implementation of the same functionality. Then, the design with multiple accelerators is evaluated by means of an audio encryption application. Two kernels (audio and AES) are accelerated sequentially and in a pipelined fashion and their execution times and speedup are compared with that of pure software implementation of the same. As explained in Section 3.2, for the former, a data dependency is created between both the kernels, i.e. the AES kernel will encrypt the output produced by the audio kernel. Therefore, the arbiter schedules their execution sequentially. For pipelined execution, the audio kernel processes a different dataset, whereas the AES kernel encrypts a different dataset and hence the arbiter schedules their execution in a pipelined fashion.

The theoretical speed up of an application is given by the Amdahl's law $\frac{1}{(1-P)+\frac{P}{S}}$.

Input size (kB)	Software execution time (ms)	Hardware execution time (ms)	Speedup
32	7.27	2.41	3.02
64	14.23	3.11	4.57
128	28.92	3.36	8.6
256	62.33	6.70	9.31
512	112.01	15.81	7.09
1,024	226.81	31.88	7.11
2,048	478.16	53.69	8.91
4,096	933.33	115.76	8.06
8,192	1934.02	214.60	9.01
16,384	3701.33	499.15	7.42
32,768	7938.31	859.82	9.23
65,536	14870.28	1783.95	8.34
131,072	31106.05	3438.60	9.05
262,144	71650.65	6909.16	10.37
524,288	123153.36	11300.18	10.9
1,048,576	262510.51	20315.68	12.92
2,097,152	514873.56	42167.41	12.21

Table 4.3: Execution times and speedup for different input sizes for Audio CCU

Since the entire application the accelerated, the fraction enhanced (P=1) becomes one. Therefore speedup is calculated using the formula software execution time over hardware execution time. In case of multiple CCUs, numerator is the sum of execution times of two software functions, in this case audio and AES applications. Same applies to the denominator. It is note worthy that, when two kernels are accelerated performing a DPR, the denominator will also include the reconfiguration overhead when speedup is calculated. This will increase the value of the denominator and hence reduce speedup, in cases where execution time is comparable to that of reconfiguration overhead i.e. small input sizes. Thus by accelerating multiple kernels concurrently without DPR, the reconfiguration factor in the denominator is effectively avoided and hence have good speedups even for small input sizes.

Figure 4.1 shows the speedup obtained by accelerating the different applications (audio, audio+AES) which is calculated using the numbers in Table 4.3 and Table 4.4. As seen, the audio application when accelerated on hardware produces 3x speedup compared to software, even for small input sizes. For file size between 32-512 kB, the audio CCU has better speedup compared to multiple CCUs because, the hardware execution time of the second CCU is increasing the overall hardware execution time and hence pulling down the speedup in case of multiple CCUs. However, for these file sizes, sequential execution is better than pipelined execution. In the sequential execution of hardware, the second CCU will not produce address translation interrupts for reads because, the first CCU would have already requested those addresses during its writes

Input size (kB)	Software execution time (ms) (Audio+AES)	Sequential execution time (ms)(HW)	Speedup	Pipelined execution time (ms)(HW)	Speedup
32	7.68	2.601	2.953	2.791	2.752
64	14.997	3.300	4.545	3.500	4.285
128	30.414	3.871	7.856	3.940	7.719
256	65.30	7.608	8.583	7.943	8.221
512	117.935	16.977	6.947	17.120	6.889
1,024	238.652	37.962	6.287	32.319	7.384
2,048	500.234	58.582	8.539	55.537	9.007
4,096	980.711	128.386	7.639	120.473	8.140
8,192	2030.707	239.136	8.492	223.697	9.078
16,384	3891.539	530.764	7.332	511.420	7.609
32,768	8319.571	900.480	9.239	875.619	9.501
65,536	15630.408	1845.732	8.468	1807.984	8.645
131,072	32628.444	3543.935	9.207	3479.621	9.377
262,144	74861.987	7095.650	10.550	6981.829	10.722
524,288	129432.291	11656.227	11.104	11439.00	11.315
1,048,576	274887.141	21005.002	13.087	20584.46	13.354
2,097,152	540387.833	43515.570	12.418	42693.12	12.657

Table 4.4: Execution time and speedup for multiple concurrent CCUs

and hence effectively reducing the time needed for execution. On the other hand, in pipelined execution of the CCUs, since both the accelerators would produce interrupts for address translation, they will have large IO time and spend less time in execution. Hence the speedup of pipelined execution is less than that of sequential execution for these input sizes.

The speed up gradually increases with the input size. This is because, the IO time of the application is very much less compared to the large execution time on hardware for bigger file sizes. From file sizes 1 MB to 2 GB, the pipelined execution produces better speedup compared to sequential execution. Naturally, this is because, each CCU spends more time in execution and thus every phase gets nested within an execution phase of the previous CCU, thus forming a neat pipeline. With sequential execution, although the second CCU still doesn't raise any address translation interrupts, the execution time becomes quite large and the overall execution time is larger than when compared to pipelined execution. Furthermore, it can be seen that accelerating both the applications concurrently or sequentially in hardware produces up to 13x speedup which is 1x more speedup when compared accelerating only one application (audio) on hardware. It is also seen that the speedup of different acceleration strategies follows a similar trend. This is because, the audio CCU is the most dominant and compute intensive application of the two and it consumes 90%-95% of the execution time.

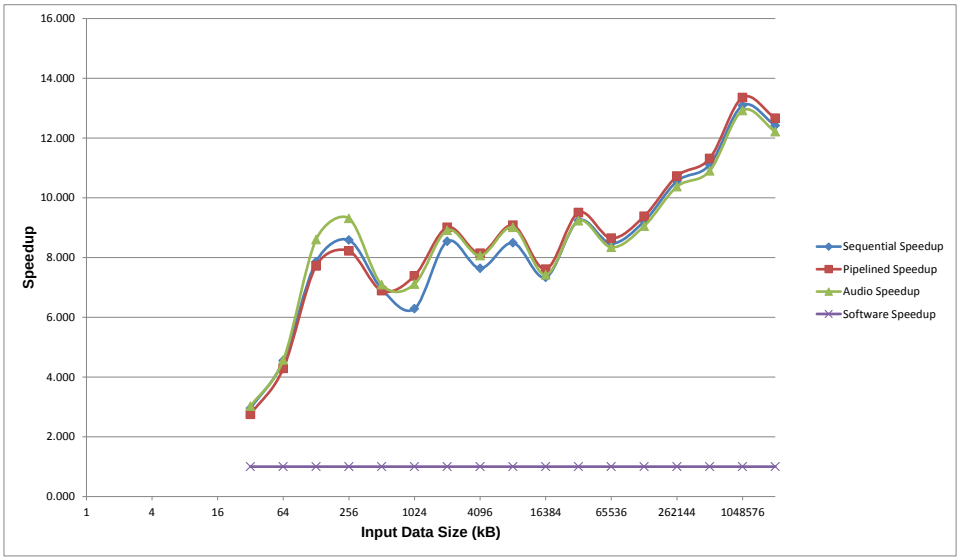

Figure 4.1: Input size versus speedup

Therefore, the speedup trend of all acceleration looks similar to that of the speedup given by audio CCU. Finally, we can also see some oscilations in the speedup graph. This is due to two reasons as speed up is a ratio of software to hardware execution time. First, from a software standpoint, the execution of the software is scheduled by the OS. Therefore the execution time of software is subject to vary for different input sizes. Second, from a hardware standpoint, an address translation interrupt will cause the driver to provide the physical address to the TLB. For smaller input sizes, the address translation interrupt itself might have its toll on the hardware execution time. However, for larger input sizes, when the TLB entries exceed 512, few entries has to be overwritten again. When this happens, the existing pages locked by the driver in memory, might be swapped to the disk. Therefore this would have its overhead in the hardware execution time. Combined effect of both the above mentioned factors result in oscilations.

The execution time of pipelined and sequential execution in hardware is greater than that of the audio kernel. This however, cannot be seen in the graph since the difference is very less in the logarithmic scale. Figure 4.2 shows that all the kernels have quite linear increase in their execution time with respect to the input sizes. In such cases, one may estimate at runtime the overall execution time of tasks running in software and in reconfigurable hardware, based on the input set. This can be achieved using some profiling information. In doing so, it can be chosen dynamically at runtime whether or not dynamic self-reconfiguration and hardware acceleration of a function would deliver speedup. More information on prefetching the bitsreams and inserting system calls for reconfiguration during compilation can be found in [29].

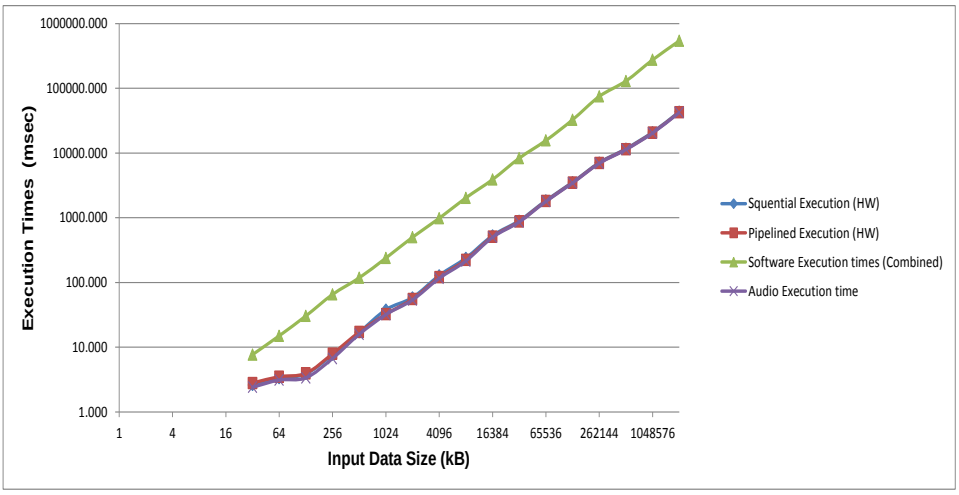

Figure 4.2: Input size versus execution times

Thus, in summary it can be concluded that accelerating multiple kernels sequentially or concurrently will provide better speedup than accelerating a single kernel, for larger input sizes. Also the reason for getting higher speedups for bigger input sizes is due to the fact that the overhead of system calls, and IO time is getting averaged out due to the large amount of hardware execution time. In this study, this is the case for input sets larger than 1 MByte.

4.3 Bitstream size vs Reconfiguration Latency

This section shows the measurement of the reconfiguration latency for different bitstream sizes. The main factor that determine how fast a new accelerator can be dynamically reconfigured and will be available for execution depends on the memory access latency, i.e. the time spent waiting for data to arrive to the ICAP controller. The latency of Dynamic Partial Reconfiguration is measured in our system, to evaluate the time taken to configure bitstreams of different sizes. The size of the bitstream is directly proportional to the area that has to be reconfigured and has a direct effect on reconfiguration latency.

Figure 4.3 shows the reconfiguration latency versus the bitstream size and Table 4.5 shows the corresponding numbers. Naturally, the larger the bitstream the longer the reconfiguration latency. Once the reconfiguration process has been initiated by the ICAP controller, the bitstream needs to be fed to the ICAP without any interruptions. Furthermore, we need a large DMA read queue (128KBytes) to buffer a large part of the bitstream. This is due to the fact that he HT link can push more data into the queue than what the ICAP controller can read at a given time. This theoretically could

Bitstream size (kB)	Reconfiguration latency (msec)
29.10	0.15
100.50	3.39
143.30	5.01
200.19	7.52
300.29	12.17
400.39	17.02
500.48	21.28
600.58	24.14
700.68	27.49
800.78	28.47
900.87	29.43
1000.97	29.52
1085.30	30.76

Table 4.5: Bitstream size versus reconfiguration latency

cause an overflow as explained in Section 4.5. Bitstreams of a few tens of Kbytes require less than a millisecond to be configured, a 256 KByte bitstream roughly needs about 10 milliseconds, while bitstreams larger than a MByte have reconfiguration latency of more than 30 milliseconds. It is also interesting to note that, the graph deviates from the theoretical linear pattern for file sizes greater than 800kB. This is because, for larger file sizes, the time for address translation and IO is being averaged out by the time take to read the bitstreams from the queue. In other words, the controller spends more time in reading the bitstream from the queue and reconfiguring the device.

4.4 Floorplaning

This section presents a floorplan that has been adapted for the design. In this thesis, PlanAhead software was used as a flow management tool for floorplanning, MAP, PAR and generation of partial bitstreams. Thorough analysis, floorplanning, physical constraints are applied to help control the timing of the implemented design. Trial and error approach was used to fix various blocks to specific regions of the FPGA device. After few runs and a good analysis of the timing report produced by the software tool, various modules were fixed to specific regions of the device so as to achieve the required timing.

Figure 4.4 illustrates the floorplan of our design on the Virtex4-100 FPGA. The blocks that compose the HTX-core interface are distributed in the FPGA in order to fit the pinout restrictions of the device. The ICAP controller and ICAP block as well as the various wrapper blocks are shown in the figure. The partially reconfigurable region is the largest rectangle available in the FPGA, which fits the area as well as

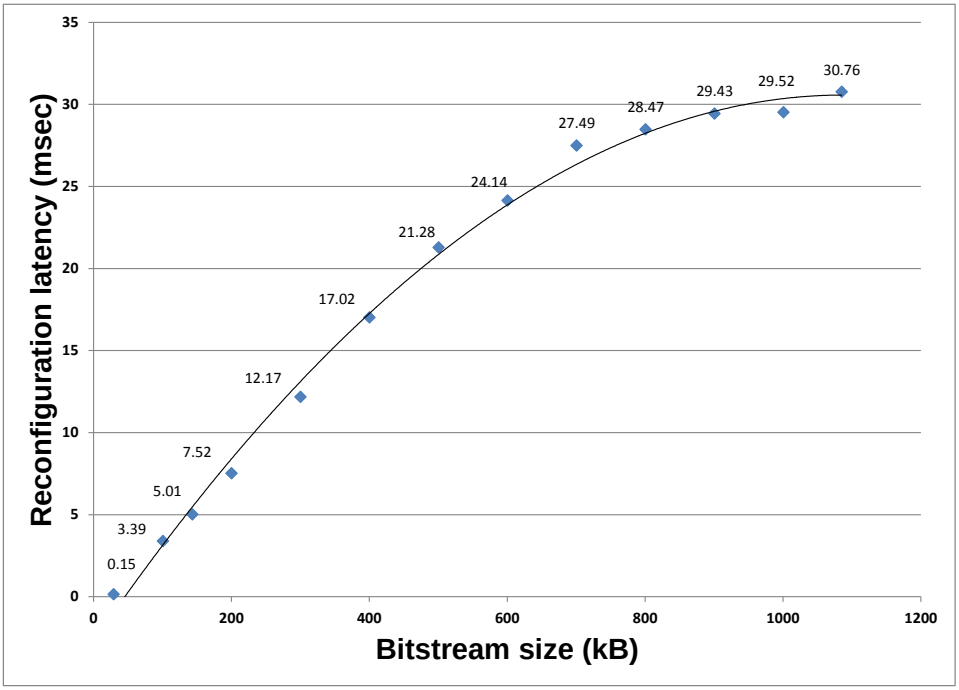

Figure 4.3: Reconfiguration latency versus bitstream size

the timing constraints of our design. In order to meet our timing constraints, the partially reconfigurable area needs to be close to the DMA read and write FIFOs, to the Interrupt Handler and ICAP blocks, as well as to the control and XREGs. In the floorplan of a design with multiple CCUs, all blocks of the static region are fixed in the same positions, except that there are two PR regions defined for the reconfigurable modules so as to fit two accelerators at the same time.

Although the rest of the design can operate at 200MHz, the frequency of the entire FPGA device is limited by the accelerator to 100 MHz. This is mainly due to the PlanAhead tool used for PR. The reason for such drop in the design frequency is as follows. Audio kernel is quite resource consuming and therefore has to be constrained to a PR region such that, it has sufficient resources. This implies that the AES kernel also has to be placed in the same PR region. this leaves, just enough resources for the placement of modules in the static regions and has to be constrained to those particular locations on the FPGA. Therefore, the tool has to do number PAR optimizations to route all signals properly. This affects the operational frequency of the design considerably. Furthermore, it is also noteworthy that in most cases, even without floor planning, the post-PAR frequency of the PR design is considerably lower than that of post-PAR frequency of a flat design, due to the insertion of partition pins in a PR design by the tool.

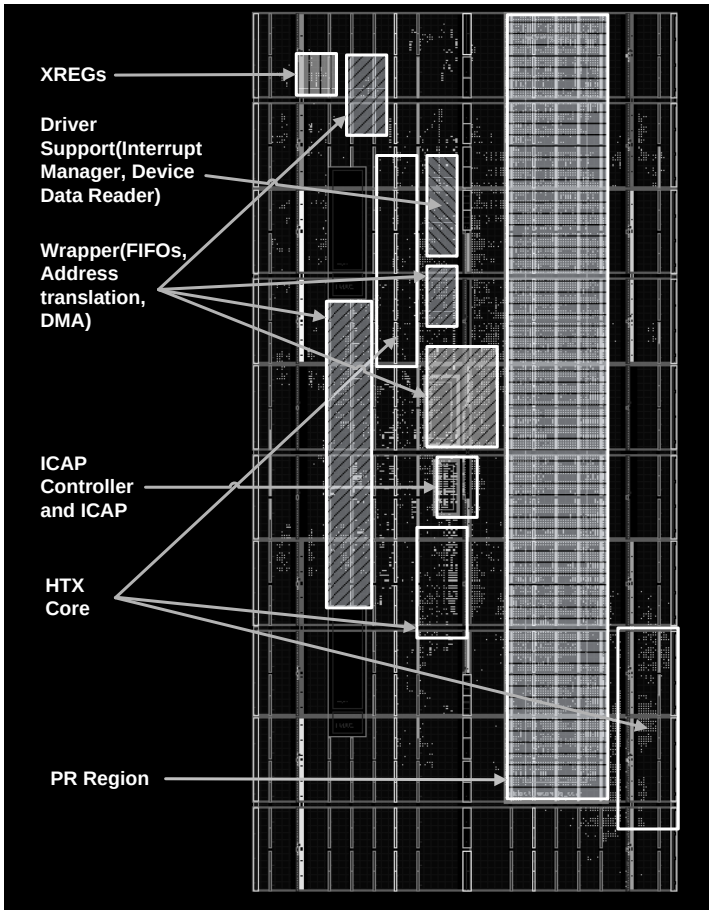

Figure 4.4: Floorplanning with single CCU

4.5 Bandwidth of HTX and Queue size

The HTX board has a 16bit wide bidirectional HT link interface that can be plugged to any AMD Opteron processor node with HTX, e.g., the Iwill DK8-HTX dual processor HyperTransport-enabled server board. The HT DDR interface operates at a frequency of 400MHz. Since HT interface does a DDR transmission the maximum operation frequency is 800 MHz. Therefore, from the given data it can be found that the HT link interface can deliver a peak of 1.2 GBps unidirectionally. Which implies that it can deliver a peak bandwidth of 3.2GByte/s and also at a very low latency [19]. Furthermore, the link can process up to 32 outstanding requests.

Section 4.3 evaluates the reconfiguration latency for different bitstream sizes and states that, a large part of the bitstream (128kB) has to be buffered before it is fed into the ICAP. In theory, having small FIFO sizes will cause a buffer overflow. This is due to the fact that the operational frequency of the HT link is different from that of the design. As explained above, HT link operates at 400 MHz with DDR transmission with a bus width of 16 bits, but the design operates at 100 MHz. A bitstream size of

Design (Wrapper + CCU)	Power Consumption (Watts)
Audio	2.224
AES	3.192
Audio + AES	4.037

Table 4.6: Design and power utilization

128kB (131,072 bytes) has to be read from the memory and fed to the ICAP. The ICAP controller reads 4 bytes of data every cycle and feeds it to the ICAP, during which the HTX bus would have already written 16 bytes of data. At this rate, when the HTX link has written 131,072 bytes of data approximately, the ICAP controller would have just read 32,768 bytes. Therefore, the queue size has to be a minimum of 96kB to buffer the remaining data, in order to avoid buffer overflow. Therefore, a queue size to the nearest power of 2 is chosen and hence 128 kB.

4.6 Power Consumption

This section shows the power consumption by the design with different CCUs. Although power consumption is not a main factor of concern for general purpose computing applications, the audio application that has been used is originally targeted for embedded platforms [30]. Therefore, it would be interesting to measure the power consumption of our design with audio application and compared it with the design that has other kernels. In this thesis, Xilinx XPower Analyzer is used to analyze the power consumption of the design. There are two factors that determine XPower Analyzer's accuracy. The accuracy of data within XPower Analyzer and the accuracy of the stimulus provided by the user. XPower Analyzer relies upon stimulus data to estimate power consumption for internal components. Accurately entering valid input frequencies and toggle rates is necessary to generate a proper power estimate. In order to ensure this, a signal dump file for each design was generated from the simulation and was given as an input file to the tool for power analysis.

Figure 4.5 shows the power consumption by each design. The design with the audio CCU consumes the lowest power as it was originally targeted for embedded platforms. Naturally, the one with both the kernels consumes the highest power. Furthermore, a design with only the AES kernel consumes 45.5% more power than the design with just the audio kernel.

4.7 Conclusion

This chapter has presented the results of various experiments performed on the proposed design. Different metrics such as area, execution time, speedup and power were measured. Two different applications were used to evaluate the proposed design. First, the hardware execution time of a beamforming audio processor was measured and its

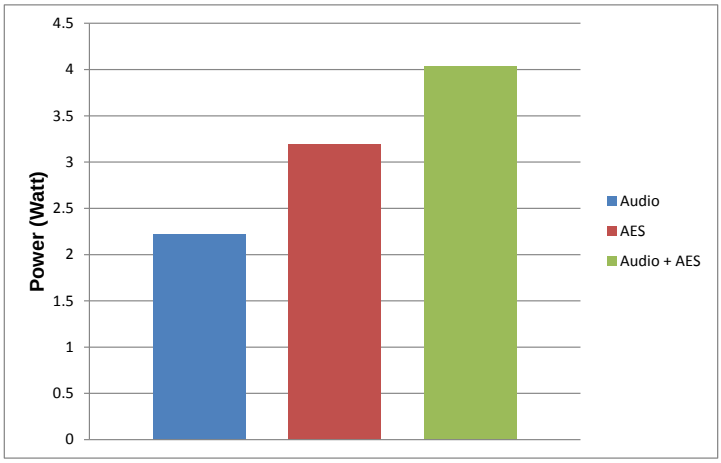

Figure 4.5: Design and power consumption

speedup as compared to a software execution was calculated. Later an audio encryption application (audio + aes kernels) is accelerated in hardware, and its performance was measured. This application was accelerated in two ways. First, both the kernels were executed in hardware sequentially with data dependencies and later, in a pipelined way without data dependencies. This approach gave up to 13x speedup with pipelined execution when compared to a pure software implementation, which is 1x more speed up than accelerating the audio kernel separately on hardware.

The amount of resource utilized by the design on the FPGA was calculated. It is found that the design with multiple CCUs consumes more resources than the one with the single CCU. This is due to fact that, the wrapper occupies more resource due to the arbiter logic in place and the PR region consists of two separate partitions one for each accelerator. Furthermore, the design was floor planned to fix various modules of the wrapper and CCU to specific regions on the FPGA, so as to meet the timing requirements for the design.

Further, the reconfiguration latency for different bitstream sizes were also measured. Reconfiguration latency depends on the size of the bitstream, which in turn depends on the area of the FPGA to be reconfigured. Larger bitstream size also demands a larger buffer size to store the bitstreams. This is because, the HT Link operates at a higher frequency than the design and hence it writes into the buffer more data, than what the design can actually read and feed into the ICAP at any given time. Bandwidth calculation of the HT link interface is also done. Finally the chapter is concluded by measuring the power consumption for each application and comparing them to find the most energy efficient one.

5 Conclusion

This thesis has described a generic approach to integrate a dynamically reconfigurable device into a general purpose computing platform, to install and execute hardware instances of software functions on-demand. Further, it has also served as a starting point to accelerate multiple software functions concurrently on the reconfigurable hardware. Dynamic Partial Reconfiguration is achieved by inserting system calls into the original program. The bitstreams are read from the main memory, loaded to the configuration memory using the ICAP and thus performing Dynamic Partial Reconfiguration. Multiple functions have been accelerated by implementing multiple accelerators in the design, providing wrapper support for proper management of the resources and having memory consistency between reads and writes of different accelerators. During which, support for non-blocking execution of multiple software functions was given by the driver.

This chapter presents a short summary of the work described in this thesis, and the various objectives that has been met. Finally, future work can be done to improve on the existing design is also discussed.

5.1 Summary

Chapter 2 started with a small introduction to the various commercial platforms available for hardware acceleration. The HTX reconfigurable platform was also studied in detail. It has been developed as an open source solution for addressing the various known issues in integrating a reconfigurable platform into a general purpose machine. The platform served as a starting point for this thesis work due to several reasons. First, it is a generic solution to accelerate software functions in hardware. Second being, it uses the existing off-the-shelf components to integrate a reconfigurable device into a general purpose system. Finally, its an open source solution for providing hardware acceleration. To conclude, a short introduction to the Xilinx Partial Reconfiguration terminology was provided to understand the basic PR flow and terminology.

Chapter 3 started with presenting the various architectural modifications to make the platform dynamically reconfigurable. First, the architecture of the system for DPR was explained. In order to read the bitstream from the memory, and feed it to the ICAP on the FPGA, an ICAP controller was designed. Further, the design was partitioned into static and reconfigurable regions to generate the partial bitstreams. Further, to provide software support for the system, additional system calls were introduced that to create a memory map IO file and pass the pointer to the driver. The driver then wrote the starting address to the control register of the device and thus initiating Dynamic Partial Self-Reconfiguration. Next, partition pins were described in detail. They were

used to interface the static and reconfigurable regions of the design while using a PR flow. The next half of the chapter started with explaining the various architectural changes to support the integration of multiple accelerators. An arbiter was designed to manage the access of the resources of the static region, provide synchronization and memory consistency. Finally the compiler and the driver were modified to support the annotation of multiple software functions and non-blocking execution respectively.

After making the necessary architectural changes, the design was evaluated by using two different applications to compare the various performance metrics. Chapter 4 explained the the results that were obtained by accelerating two different applications on the hardware. The first application was a beamforming audio application, and the second application was an audio encryption application which had to accelerate two kernels simultaneously (audio + AES). The experimental results showed that accelerating a single kernel(audio) in hardware gave up to 12x speedup and accelerating two kernels simultaneous gave up to 13x speedup compared to software. Next, time taken to configure the device with bitstreams of different sizes was evaluated. It was found that, the reconfiguration time varies linearly with the bitstream size, which is in direct correlation with the area of the FPGA to be reconfigured. Finally the system was also evaluated in terms of area and power. It was finally concluded that, accelerating multiple kernels sequentially or concurrently will provide better speedup than accelerating a single kernel, for larger input sizes.

5.2 Contributions

The main objectives of the thesis were the following. The first aim was to establish a generic approach to integrate a dynamically reconfigurable device into a general purpose system with a high-speed interconnect. Furthermore, it was also intended to serve as a starting point for accelerating multiple software functions of a single process, in combination with non-blocking execution. In order to achieve above goals, the following contributions were made to improve the existing design of the HTX reconfigurable platform:

- **Dynamic Partial Reconfiguration:**

  The primary objective of the thesis was to make the HTX platform dynamically reconfigurable. An ICAP controller was designed to read the bitstreams from the main memory and feed it to the ICAP. The design then had to be adapted to the Partial Reconfiguration flow to generate the partial bitstreams. The ICAP controller along with the wrapper and the HT-link interface was placed in the static part and the accelerator was placed the reconfigurable region. In order to provide software support for DPR, the device driver was extended to create a memory map IO file and the pointer of the file was written into the control register of the device, so that the controller could read the starting address of the bitstream. Finally, when a function call is made for DPR, the driver puts the process to sleep, writes the starting address of the bitstream into the device, and starts the

execution, thus performing DPR. Furthermore, in order to debug such a design after programming the FPGA, a debugging methodology had been implemented to read-back the configuration memory through the JTAG port registers;

- **Multiple accelerators:** The next objective of the thesis was to provide a starting point to accelerate multiple software functions of a single process, in combination with non-blocking execution, concurrently on the hardware. In order to achieve this, several modifications were made in the hardware wrapper, device driver and the compiler plug-in. As a first step, multiple accelerators were implemented in the design to execute multiple kernels concurrently. Further modifications were made in the wrapper in order to manage the access to the resources in the static part;

- **Wrapper modifications:** As mentioned above, the first modification to the existing wrapper was to include proper management of resources in the static part. Therefore, an arbiter module was designed in the hardware wrapper. All the signals and buses that interface the static part and the CCU had to go through the arbiter module. It is responsible for the following. First, it checks for data dependencies between the accelerators and schedules their execution accordingly. Second, it serves to provide memory consistency between the read and write requests from different accelerators. Next, it provides a synchronization mechanism when accelerating two streaming applications simultaneously on the hardware. Finally, it detects the end_op signals from both the accelerators and signals the interrupt handler to notify the device driver. Furthermore, modifications were made to the other modules in the wrapper to support the integration of multiple accelerators in the design;

- **TLB support for streaming applications:** The existing TLB is used to translate virtual address to physical addresses. To add to this, additional support for accelerating multiple streaming applications on the hardware is provided. In scenarios where there is a data dependency between the two functions that are being accelerated, instead of scheduling their execution sequentially, their execution can still be pipelined. For doing this, a lock is placed on the address locations that has to be updated by CCU1 before CCU2 can read them. These locations are immediately accessible for reading as soon as the lock on them is released. Hence there is a pipelined execution;

- **Driver support for non-blocking execution:** In order to accelerate multiple software functions concurrently on the hardware, driver had to allow the non-blocking execution of multiple functions in a single process. However, the problem with allowing non-blocking execution is that, the driver cannot translate virtual addresses to physical addresses when the process is not put to sleep. Therefore, to supply the device which physical address (in case any address translation interrupts are raised) before the process is put to wait state, the TLB of the device is pre-filled with the physical address entries from the stack and heap addresses of the process's address space. This allows the device to find all the addresses in the TLB without having to raise an interrupt until the process goes into a wait state. Thus providing the non-blocking execution for software functions;

- **Compiler support for accelerating multiple functions:** The system calls were inserted in the original code using a modified GCC compiler plug-in after the function to be accelerated was annotated. However, additional support was provided to the compiler plug-in to identify multiple annotated functions and insert system calls accordingly. This includes additional condition in the plug-in to check the number of times the replace pass has been called and inserting system calls accordingly;

- **Performance Evaluation:** The final contribution in the thesis was to test the performance of the system on the FPGA. In order to do this, two applications were ported to the platform and various performance metrics were evaluated. The first application was an audio beamforming application. The next one being an audio encryption application that executed two kernels (Audio and AES) kernel concurrently. To test the second application, it was executed with and without data dependencies. On executing the application with different input sizes, our method provided up to 12x speedup when accelerating a single application and up to 13x speedup when accelerating two kernels in a pipelined fashion, compared to software. The platform was also tested by reconfiguring it with bitstreams of different sizes and checking it against the reconfiguration latency. Finally, the power consumption for various designs were measured.

5.3 Future Work

The HTX platform described in the thesis work is able to dynamically install and execute hardware instances of software functions on demand through system calls. It also served as a starting point for accelerating multiple software functions on hardware concurrently. However, there are still areas which are not yet addressed in this thesis work which would improve the usability of the HTX platform. The main areas of interests and improvements are, providing compiler support for non-blocking execution, a more robust driver support for non-blocking execution, allowing multiple processes to access the device at the same time and finally, a hardware software mapping to identify which function maps to which accelerator in hardware. They are explained in detail as follows:

- **Compiler support for non-blocking execution:**

  Currently the driver provides support for non-blocking execution and data dependencies are checked in hardware during runtime. However, data dependencies can be checked during the compilation phase. In order to do this, additional system call should be inserted when there are any variables that are dependent on the return value of the first software function that is being accelerated. The compiler plug-in support for this would consist of following functionalities. The compiler has to check for any variables that are dependent on the return results of the first function that is being accelerated, and it has to insert a new function call if there are any dependencies. Putting the process to sleep, and freeing the pages locked by the device should be moved to this system call. 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;

- **Software hardware mapping:**

  In this thesis, the software function that has been annotated first will result in the execution of the first accelerator in hardware and so on. However, when there is compiler support for non-blocking execution, there may be a delay in the execution between the accelerators due to dependencies in the code. In such cases, in order to to start the execution of the correct accelerator, there has to be some kind of mapping between the software function that has been annotated and the hardware accelerator that has to be started. This can be accomplished by including an extra argument in the *IOCTL* system call. Therefore, the driver will not only write an execute command to an array location of the device's memory mapped region, but also an ID associated with it to another location in the array. The device will then detect the start command as well as the ID associated with it, and signal the start of the corresponding accelerator. The compiler plug-in support for this would consist of inserting an argument to the *EXECUTE* system call depending on the number of times the replace pass has been called;

- **Device access by multiple processes:**

  In addition to allowing a single process to accelerate multiple functions, multiple processes can be allowed to access the device to accelerate multiple functions. The existing modifications in the wrapper and the driver support for non-blocking execution can be used to achieve this 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. The interrupt handler would need to identify which interrupt belongs to which process. The driver and the device also needs to know to which process the results has to returned to and to which address space the pages belong.

Bibliography

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

[2] SGI. Reconfigurable application-specific computing user's guide, 2006.

[3] Convey. http://www.conveycomputer.com/Resources/ConveyArchitectureWhiteP.pdf.

[4] Xilinx. Xilinx virtex4 configuration guide.

[5] P. Bertin and H. Touati. Pam programming environments: practice and experience. In *FPGAs for Custom Computing Machines, 1994. Proceedings. IEEE Workshop on*, pages 133 – 138, April 1994.

[6] Assaf Shacham O.L.Ladouceur. The data vortex optical packet switched interconnection network. *Journal of Lightwave Technology, Vol. 26, Issue 13*, pages 133 – 138, April 2008.

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

[8] J.R. Hauser and J. Wawrzynek. Garp: a mips processor with a reconfigurable coprocessor. In *FPGAs for Custom Computing Machines, 1997. Proceedings., The 5th Annual IEEE Symposium on*, pages 12 – 21, April 1997.

[9] S. Hauck, T.W. Fry, M.M. Hosler, and J.P. Kao. The chimaera reconfigurable functional unit. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 12(2):206 – 217, February 2004.

[10] Rahul Razdan and Michael D. Smith. A high-performance microarchitecture with hardware-programmable functional units. In *Proceedings of the 27th annual international symposium on Microarchitecture*, MICRO 27, pages 172 – 180, New York, NY, USA, 1994. ACM.

[11] R.D. Wittig and P. Chow. Onechip: an fpga processor with reconfigurable logic. In *FPGAs for Custom Computing Machines, 1996. Proceedings. IEEE Symposium on*, pages 126 – 135, April 1996.

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

[13] AMD. Hypertransport bus. www.hypertransport.org.

[14] SGI. Altix sgi machines. http://www.sgi.com/products/servers.

[15] Clear speed. Clear speed. http://www.clearspeed.com.

[16] Convey Computer. Convey computer. http://www.conveycomputer.com.

[17] Intel. Intel atom processor with built-in altera arria fpga. www.eejournal.com/archives/on-demand/2011012001-arrow-ct.

[18] A.A.C. Brandon. General purpose computing with reconfigurable acceleration. Msc thesis, Delft University of Technology, November 2010.

[19] Hyper Transport consortium. Latency comparison between ht and pcie in communication systems. http://www.hypertransport.org/.

[20] Maxeler Technologies. Max3 acceleration. http://www.maxeler.com/content/hardware.

[21] DRC Computers. http://www.drccomputer.com/pdfs/DRC_Accelium_Coprocessors.pdf.

[22] SRC Computers. http://www.srccomp.com/.

[23] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, and E.M. Panainte. The molen polymorphic processor. *Computers, IEEE Transactions on*, 53(11):1363 – 1375, November 2004.

[24] A. Brandon, I. Sourdis, and G.N. Gaydadjiev. General purpose computing with reconfigurable acceleration. In *Field Programmable Logic and Applications (FPL), 2010 International Conference on*, pages 588 – 591, September 2010.

[25] Cray xd1 Supercomputers. http://www.cray.com/products/Legacy.aspx.

[26] David Slogsnat, Alexander Giese, Mondrian Nüssle, and Ulrich Brüning. An open-source hypertransport core. *ACM Trans. Reconfigurable Technol. Syst.*, 1:14:1 – 14:21, September 2008.

[27] Intel. Quick path bus. www.intel.com/technology/quickpath/.

[28] Xilinx. Xilinx virtex4 user guide.

[29] A.Nandy. System level software support for dynamic partial reconfiguration. Msc thesis, Delft University of Technology, September 2011.

[30] D. Theodoropoulos, G. Gaydadjiev, and G. Kuzmanov. A reconfigurable beamformer for audio applications. In *Application Specific Processors, 2009. SASP '09. IEEE 7th Symposium on*, pages 80 – 87, July 2009.

[31] B.D. Van Veen and K.M. Buckley. Beamforming: a versatile approach to spatial filtering. *ASSP Magazine, IEEE*, 5(2):4 – 24, April 1988.

[32] B. Sallberg, N. Grbic, and I. Claesson. Online maximization of subband kurtosis for blind adaptive beamforming in realtime speech extraction. In *Digital Signal Processing, 2007 15th International Conference on*, pages 603 – 606, July 2007.

[33] L. Parra. Steerable frequency-invariant beamforming for arbitrary arrays. *Journal of the Acoustical Society of America*, pages 3839 – 3847, June 2006.

SVF file to read back configuration memory

A

```
TRST OFF;
ENDIR IDLE;
ENDDR IDLE;
STATE RESET;
STATE IDLE;
FREQUENCY 1E6 HZ;
TIR 14 TDI (3fff) SMASK (3fff) ;
HIR 8 TDI (ff) SMASK (ff) ;
HDR 1 TDI (00) SMASK (01) ;
TDR 1 TDI (00) SMASK (01) ;
//Loading device with 'idcode' instruction.
SIR 16 TDI (00fe) SMASK (ffff) ;
SDR 32 TDI (00000000) SMASK (fffffff) TDO (f5059093) MASK (0fffffff) ;
//Boundary Scan Chain Contents
//Position 1: xc4vfx100
//Position 2: xcf32p
//Position 3: ISPPAC_POWR607_XXN32
SIR 38 TDI (3fffffff) SMASK (3fffffff) ;
SDR 3 TDI (00) SMASK (07) ;
TIR 0 ;
HIR 24 TDI (ffffff) SMASK (ffffff) ;
HDR 2 TDI (00) SMASK (03) ;
//Loading device with 'idcode' instruction.
SIR 14 TDI (3fc9) SMASK (3fff) ;
SDR 32 TDI (00000000) SMASK (fffffff) TDO (f1ee4093) MASK (0fffffff) ;
//Boundary Scan Chain Contents
//Position 1: xc4vfx100
//Position 2: xcf32p
//Position 3: ISPPAC_POWR607_XXN32
HIR 24 TDI (ffffff) SMASK (ffffff) ;
HDR 2 TDI (00) SMASK (03) ;
//Loading device with 'idcode' instruction.
SIR 14 TDI (3fc9) ;
SDR 32 TDI (00000000) TDO (f1ee4093) ;
//Loading device with 'bypass' instruction.
SIR 14 TDI (3fff) ;
```

// reading the status register contents.
// Loading device with a 'cfg_in' instruction.
// Loading device with a 'cfg_in' instruction.
SIR 14 TDI (3fc5) ;
SDR 320 TDI (0000000000ac41200060014000000008004000c200000008001000c0000000
466aa9955ffffffff)
SMASK (ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff);
// Loading device with a 'cfg_out' instruction. REad out FDRO
SIR 14 TDI (3fc4) ;
SDR 5401087 TDI (000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000) SMASK (ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff) TDO (000000000000000000000000000000000000000000000000000000000000000000000000
00000f00000000000000000000000000000000000000000000000000000000000000000000000
00000f00000000000000000000000000000000000000000000000000000000000000000000000) MASK (ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff) ;
STATE RESET;
STATE IDLE;

C Program to initiate DPR

B

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include "../molen_htx.h"
#include <time.h>
void molen_set(const char* filename);

int main (void)
{

  molen_htx_init();
  molen_set("ccu");
  printf("\nPartial reconfiguration complete.");
  return 0;
}

void molen_set(const char* filename)
{
  int fd = open(filename, O_RDWR);
  int length = 0;
  char *arg;
  struct stat buf;

  if (fd < 0)
  {
    perror("molen_set : Filename Open");
    return;
  }
  fstat(fd, &buf);
  length = buf.st_size;
  if(length == 0)
  {
    fprintf(stderr, "molen_set : Empty input file\n");
    return;
  }
  arg = mmap(NULL, length, PROT_WRITE | PROT_READ, MAP_PRIVATE, fd, 0);
  printf("molen_set : Bitstream = \p", arg;
  printf("molen_set : Bistream size = \d", length;

  printf("molen_set : ioctl called\n");
  ioctl(htex_handle, HTEX_IOSET, arg);
  printf("molen_set : ioctl returned\n");
  munmap(arg, length);
}
```

Curriculum Vitae

Venkatasubramanian Viswanathan (Venkat) was born in Chennai, Tamilnadu, India on 8th October 1987. He received his Bachelor of Engineering (B.E) in Computer Science and Engineering from Anna University, Chennai, India. His project during his bachelors was on "Analyzing the Software Architecture using Innovative Patterns"'. He was awarded a degree with a First class with distinction.

His passion towards Computer Architecture and High Performance Computing motivated him to do a masters degree in Computer Engineering specializing in General Purpose and High Performance Systems in TU Delft, Netherlands. His research interests in the field of Computer Architectures has been growing ever since. He is currently working on his Master thesis topic on "Hardware support for Dynamic Partial Reconfiguration". During his thesis, he has worked on Dynamic Partial Reconfiguration of the HTX reconfigurable platform and support for multiple concurrent accelerators. His work during the thesis also led to a publication in the International Conference on Field-Programmable Technology 2011, New Delhi, India.

Venkat is also a very motivated and enthusiastic person who always looks forward to new challenges. He considers challenges as an opportunity to explore and improve himself. Furthermore, he is highly passionate about philosophy, cosmology and astronomy. Amongst all the other work in a day, he always finds some time to explore the above mentioned fields. Finally, Venkat loves to explore new places, meet new people and spend time with friends.

Venkatasubramanian Viswanathan