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

2011

MSc THESIS

System Level Support for Dynamic Partial Reconfiguration

Abhijit Nandy

CE-MS-2011-24

Abstract

In this thesis a generic approach for integrating a dynamically reconfigurable device into a general purpose system interconnected with a high-speed interconnect, is described. The system dynamically installs and executes hardware instances implementing functions to accelerate parts of a particular workload. The hardware descriptions of the functions (bitstreams) are inserted into an unified executable running on the host. This is achieved through an extension to the GCC compiler which in addition inserts system-calls to the device driver controlling the reconfigurable device. Thereafter, the general purpose host-processor manages the hardware reconfiguration and execution through a Linux device driver. The device has direct access to the main memory (DMA) operating on virtual addresses; it further supports memory mapped IO for data and control, and is able to interrupt the host for synchronization. The above system is implemented on a general purpose AMD Opteron-244, and 1 GB of DDR memory providing a HyperTransport bus to connect a Xilinx Virtex4-100 FPGA. Moreover to facilitate automatic generation of hardware, an open source C to VHDL compiler is used. Finally, our proposal is evaluated using a secure audio processing application. This is done through acceleration of the audio processing kernel in hardware and subsequently an AES encryption function is configured via dynamic partial reconfiguration. Experimental results with up to 2GB of data show that our solution is up to 12 times faster than pure software execution.

Faculty of Electrical Engineering, Mathematics and Computer Science

System Level Support for Dynamic Partial Reconfiguration

Addition of Partial Reconfiguration support and automatic hardware generation from C code

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Abhijit Nandy
born in Jamshedpur, India

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

System Level Support for Dynamic Partial Reconfiguration

by Abhijit Nandy

Abstract

In this thesis a generic approach for integrating a dynamically reconfigurable device into a general purpose system connected with a high-speed interconnect, is described. The system dynamically installs and executes hardware instances implementing functions to accelerate parts of a particular workload. The hardware descriptions of the functions (bitstreams) are inserted into an unified executable running on the host. This is achieved through an extension to the GCC compiler which in addition inserts system-calls to the device driver controlling the reconfigurable device. Thereafter, the general purpose host-processor manages the hardware reconfiguration and execution through a Linux device driver. The device has direct access to the main memory (DMA) operating on virtual addresses; it further supports memory mapped IO for data and control, and is able to interrupt the host for synchronization. The above system is implemented on a general purpose AMD Opteron-244, and 1 GB of DDR memory providing a HyperTransport bus to connect a Xilinx Virtex4-100 FPGA. Moreover to facilitate automatic generation of hardware, an open source C to VHDL compiler is used. Finally, our proposal is evaluated using a secure audio processing application. This is done through acceleration of the audio processing kernel in hardware and subsequently an AES encryption function is configured via dynamic partial reconfiguration. Experimental results with up to 2GB of data show that our solution is up to 12 times faster than pure software execution.

Laboratory	:	Computer Engineering
Codenumber	:	CE-MS-2011-24

Committee Members :

Advisor:	Ioannis Sourdis, CE, TU Delft
Advisor:	Georgi N. Gaydadjiev, CE, TU Delft
Chairperson:	Koen Bertels, CE, TU Delft
Member:	Henk Sips, PDS, TU Delft

To my loving family to whom I owe all that I am today and my respected professors who have taught me to always keep pushing the limits.

Contents

List of Figures

List of Tables

Acknowledgements

This thesis represents not only my work at the keyboard but is also the result of research in Reconfigurable Computing by many distinguished professors and students at the Computer Engineering Lab of TU Delft.

I would like to begin by thanking my supervisors Ioannis Sourdis and Georgi Gaydadjiev for guiding me throughout this thesis. Your constant support and faith in me was most encouraging and allowed me to achieve much more. It was an honor working with you and I hope we do so again in the future.

I am grateful to my family: My mother Ila, my father Amit and my brother Apratim for their constant support and love. I hope I did you all proud.

I would like to thank all the professors at TU Delft who have taught me in the last two years. I learnt a lot from each of you and from my stay in the Netherlands. This thesis is the culmination of all the knowledge I gained from you. Furthermore I am thankful for the help provided on multiple occasions during my thesis by Lidwina Tromp and Monique Tromp of Computer Engineering for various administrative activities.

My very special thanks to Anthony Brandon for being always available to help me understand the finer details of our work. I hope I improved upon and did justice to the efforts you put in to the HTX Platform.

I would like to express my gratitude to Google for funding this project through the Faculty Awards Program. I hope my work contributes to their efforts in this direction.

Finally for all my friends, thank you for always standing by me and making me feel at home. I would like to especially mention Venkat, Vishwas, Dhariyash and Ravi. We had some truly wonderful times together and this thesis would not have been possible without you all.

Abhijit Nandy
Delft, The Netherlands
October 3, 2011

Achievements

The following conference paper was accepted as a result of this MSc thesis:

- Ioannis Sourdis, Abhijit Nandy, Venkatasubramanian Viswanathan, Anthony Brandon, Dimitris Theodoropoulos and Georgi N. Gaydadjiev, *"Reconfigurable Acceleration and Dynamic Partial Self-Reconfiguration in General Purpose Computing"*, Int. Conf. on Field-Programmable Technology (FPT'11), December 2011, New Delhi, India.

1 Introduction

Computers allow us to solve computational problems in two different ways, either in hardware or in software. Hardware solutions are preferred when price and performance is favorable. Such a case exists for Graphical Processing Units(GPUs) in video cards whose performance has increased and prices have fallen in response to widespread demand and technical innovation. When solving problems in hardware we can use a device like an application-specific integrated circuit (ASIC) which has the advantage of being fast and optimized for a particular task. However its structure is tied to a single application in spite of long development times and effort. Software is comparatively cheaper and can be modified far more easily to tackle the problem at hand. Yet this adaptability comes at a cost, with several orders of magnitude slower execution speed compared to a hardware implementation. An equivalent ASIC implementation provides far more efficient usage of silicon area and power.

Reconfigurable computing provides an attractive compromise between hardware and software. It allows hardware to be reconfigured for different applications but under software control. Designers can then implement performance critical and parallel logic in hardware while parts which are sequential can stay in software. Therefore an application can reap the benefits of the performance offered by hardware while still allowing the software part to be modified quickly as needed. Field Programmable Gate Arrays (FPGAs) are often the primary devices used to realize such systems. FPGAs can serve as application accelerators and provide a customized match for the specific requirements of the computing problem. A more general purpose computer such as Linux clusters or Massively Parallel Processing systems based on CPUs or GPUs alone may not provide a comparable boost. However the use of such an accelerator must be abstracted from a general programmer who may be unaware of hardware design and hardware design languages such as VHDL. The system must be able to transparently use the accelerator in the background to provide higher performance and lower power. Furthermore the system should be able to deliver increased throughput with larger workloads when hardware is added.

Integrating Reconfigurable Computing with traditional computing systems has been a challenge since the early days of FPGAs; several machines have been designed in this direction, such as PAM [5], Splash [6], and DISC [6]. The primary drawback of all these attempts was the limited bandwidth and increased communication latency between the host general purpose processor and the reconfigurable device. In order to overcome these bottlenecks, other solutions proposed a more tightly coupled reconfigurable unit integration, e.g. Garp [7], Chimaera [8], and OneChip [9]. Recently, however, several components have been released which support standard high-speed communication between general purpose processor(s), memory and other peripheral devices. These solutions use high-speed on-board links such as the Intel QuickPath [10] and the AMD

HyperTransport bus [11] and provide multi-GByte/sec low latency communication.

A number of commercial solutions have been developed based on these high speed links. DRC Computers [12] provides a solution based on multiple HyperTransport links that fits into a standard x86 processor slot and seamlessly works with the CPU. SRC Computers [2] makes the Series H MAP Processor which is a commercial FPGA based reconfigurable processor. They describe the computational logic running in the FPGA as Direct Execution Logic (DEL) which is executed by a DEL processor. The DEL processor is realized using the resources of an FPGA. XtremeData makes a module, called the In-Socket Accelerator (ISA), which is pin compatible with a CPU socket from Intel or AMD. Their product sits in a CPU socket which allows the module to be a peer to the CPU and handle interrupts. Their approach is based on the loosely coupled massively parallel processing (MPP) approach and also uses HyperTransport for high bandwidth access to the host's main memory.

The above developments offer a new opportunity for integrating reconfigurable computing in general purpose systems. Rather than suggesting fundamental architectural changes, it is better for performance and certainly cost-efficient to propose a generic solution that uses existing off-the-shelf components with only software (and configuration) modifications. Furthermore to encourage the rapid adoption of such a generic solution it is important to make it easily accessible through a known and simplified software development environment (such as C or C++). Such a system lies between a software developer and the underlying hardware accelerator thus helping to abstract away hardware complexities.

To realize an environment which can insulate a programmer from the complex nature of hardware, an automatic hardware generator is desirable. Significant work has been done in automatic software to Hardware-Description-Language translation in order to simplify the use of reconfigurable acceleration by a programmer. A few examples are DWARV [1], ROCCC [13], Catapult-C [14], Handel-C [15] and others. In this thesis I demonstrate the integration of the ROCCC C to VHDL compiler to make it easier to write programs for our selected reconfigurable platform. Furthermore I address many unsolved issues which prevent a smooth integration of reconfigurable acceleration in a general purpose machine.

The remainder of this introduction is organized into four sections. Section 1.1 describes the problems with current platforms for combining general purpose computing with reconfigurable computing. In Section 1.2 I describe the features already existing in our selected reconfigurable platform and the proposed extensions to facilitate better integration of reconfigurable computing with a general purpose computer. Finally, Section 1.3 gives an overview of the remaining chapters of this thesis.

1.1 Problem Statement

Current general purpose computers which support hardware acceleration of a given workload have several drawbacks which has prevented their widespread usage. The primary stumbling block to higher acceptance among commercial and general users has been the highly focused nature of these machines. For example the inherent fixed function nature of the XtremeData DISC with very little support for quick modification for an entirely new problem has restricted it's usage beyond the data warehousing market. Other platforms like Convey suffer from high latencies while attempting to access host memory and while reconfiguring the FPGA. Some architectures such as GARP, Chameleon, Chimaera, OneChip and PRISC attempt to solve this by integrating the reconfigurable accelerator in the same chip as the general purpose processor; however, this requires redesign of the chip, and a different fabrication process making it expensive. However, the latency and bandwidth issue can be alleviated to a large extent through the use of a standard high speed bus such as Quickpath or HyperTransport, which are specifically developed for high speed connections between processors, main memory and peripherals. This helps reduce the bottleneck presented by data access issues and from application as well as reconfiguration data riding on the same bus.

Furthermore modern FPGAs support Dynamic Partial Reconfiguration (DPR). The scope of application of a FPGA based system can be increased by incorporating this in the design. Allowing the application specific accelerator to be reconfigured at runtime enhances the functionality of a single FPGA as completely different designs can be time multiplexed into a single device. However to use DPR transparently a considerable amount of system level support is required. The following components play an especially large role in providing this support.

- the compiler should detect and generate code that reconfigures the FPGA during runtime as needed;
- a software driver is needed to provide access to the underlying design as well as support it's reconfiguration;
- hardware support must be present in the static part of the FPGA design to reconfigure the device.

Another drawback of many reconfigurable computing systems is in general the programmability. A common solution for programming these systems is either through a new or extended programming language which is then transformed by the compiler into an unified executable or additional function calls which manage the reconfigurable device. Particularly important is an unified executable which contains both the software and hardware implementations of a software function within the same binary. This allows the compiler to choose the correct implementation to use at runtime. The ideal situation for the programmer is of course to use an existing language such as C or C++ which is learnt by most programmers early in their careers. The burden of hardware acceleration should thus shift to the compiler where possible.

The final drawback of many platforms is similar to vendor lock-in. Most applications developed for a certain platform are tied to the specific hardware or development tool-chain used for that platform and therefore to the hardware's producer. This implies that the work needed to port either the application or accelerator (or both) to a different and possible better implementation of the same platform is non-trivial at best. Some platforms such as Molen and Convey address this by specifying a fixed interface to the accelerator and software. However even these platforms have some drawbacks. The next drawback is that most of these systems are not suitable for general purpose computing. Convey is aimed at the high performance computing market, meaning supercomputers and clusters used for computationally intensive tasks such as simulations in scientific research. SRC and DRC computers are meant for the same market and for data intensive applications. Thus their platforms are not ideal for general purpose computing. Furthermore each of these platforms use their own specific development environment and operating system with custom libraries and system calls. The operating systems are often tuned to the specific application to which that particular machine is targeted thus foregoing the benefits of a more general purpose system. While it is certainly possible to port an existing operating system to a platform such as Convey or Molen the preferred way would be to add reconfigurable computing support to an existing platform which runs a general purpose operating system without modifying the operating system extensively.

A platform that meets the above challenges should therefore have the following requirements.

- the reconfigurable device used as the accelerator should have high bandwidth access to the main memory of the host;
- the system should be easy to adapt to a new problem with a radically different workload through modern techniques such as dynamic partial reconfiguration;
- it should use a general purpose operating system running on common off-the-shelf hardware thus making it an easy target to port applications;
- the platform should be easily programmable in a familiar programming language.

1.2 Objectives

As discussed in the previous section there are many important requirements for a platform independent solution that utilizes reconfigurable computing. In this section I explain how this thesis attempts to meet those requirements. Our selected reconfigurable system is shown in Figure 1.1 and is the basis of this thesis work.

As shown in the figure, it consists of a hardware platform comprising of the following:

- AMD Opteron-244 1.8 Ghz 64-bit processor;
- 1 GB of DDR memory connected to an IWILL DK8-HTX motherboard;
- a AMD Hypertransport Bus on the motherboard which also provides an interface to this bus called as the HyperTransport eXpansion(HTX);
- a Xilinx Virtex 4-100 FPGA chip placed on the HTX Board developed at the University of Mannheim [16];
- a HTX module used by the FPGA to interface to the host.

The software used with this hardware platform are as follows:

- Gentoo Linux Kernel 2.6.30;

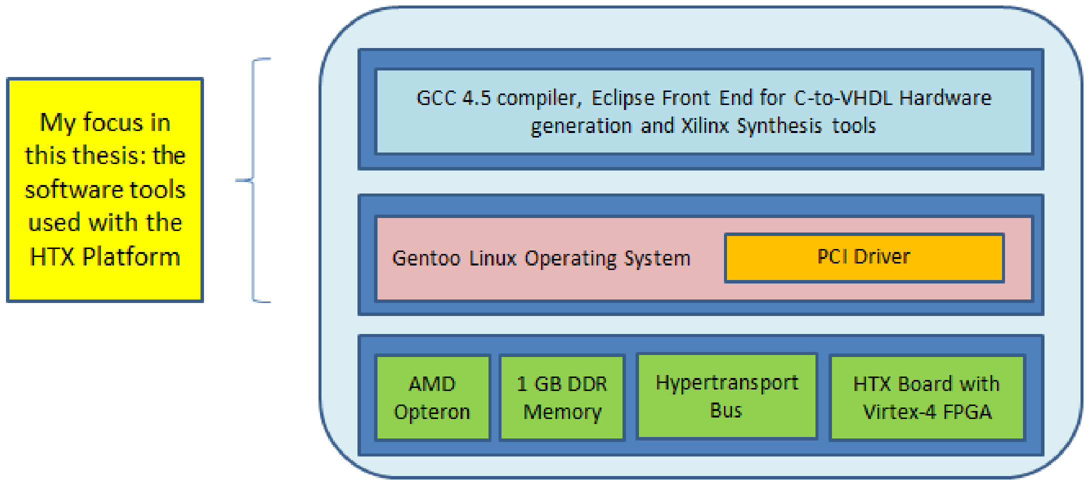

My focus in this thesis: the software tools used with the HTX Platform
GCC 4.5 compiler, Eclipse Front End for C-to-VHDL Hardware generation and Xilinx Synthesis tools
Gentoo Linux Operating System
PCI Driver
AMD Opteron
1 GB DDR Memory
Hypertransport Bus
HTX Board with Virtex-4 FPGA

Figure 1.1: The HTX Platform indicating it's various components as well as my area of focus.

- Linux PCI driver
- a modified GCC 4.5 compiler

The above software and hardware components together comprise the HTX Platform. This term will hereafter refer to both these components. The points listed below are the enhancements done to this AMD Opteron Hypertransport based machine. My work mainly involved the tools and operating system and resulted in a more flexible and programmable platform that can be used for a large number of applications.

- **Dynamic Partial Reconfiguration Support**: This is the primary enhancement done to the system and is implemented via the Partial Reconfiguration feature of the Virtex 4 device. It uses the ICAP component to reconfigure hardware instances;
- **Driver Support for Partial Reconfiguration**: Non-blocking execution support is added to allow a user process to initiate configuration of the FPGA and continue execution without waiting for reconfiguration to complete. When the FPGA is needed by the process which initiated it's reconfiguration then it's ready for immediate use. This is used for the prefetching option of the system;
- **Compiler Enhancements**: The GCC 4.5 compiler is modified to insert code at the appropriate places to support partial reconfiguration. Compiler logic has also been inserted for efficient use of the prefetching feature through proper scheduling of partial reconfiguration;
- **Automatic Hardware Generation**: The ROCCC C-to-VHDL compiler is integrated into the tool-chain to allow complete hardware to be automatically generated starting from C or C++ code for a function.

1.3 Overview

This thesis consists of four chapters after this introduction. In the next chapter, Chapter 2, I discuss our selected reconfigurable platform and how it offers a new approach towards a general-purpose system with reconfigurable acceleration. Some of the novel features offered by this platform are discussed along with the mechanism used to integrate the FPGA and control it. This chapter also provides an introduction to the basic premise of dynamic partial reconfiguration and the adaptations required in a typically flat design to accommodate it. The Hierarchical Design Methodology and some of the terminology associated with this flow is presented. I then move on to describe some existing commercial and research machines existing today that have similarities to our Platform. This includes the Convey Computer, the XtremeData DISC, SGI Altix, SRC Computers and machines from DRC. Finally I touch upon the state of current research in C-to-VHDL compilers and a few existing solutions. These are the DelftWorkBench [1], Handel C [15], Catapult C [14], SRC's Map compiler [2] and a few others. I conclude with a discussion of the ROCCC compiler and the reasons for integrating it into our tool-chain.

Interested readers already familiar with the background work, may want to skip directly to Chapter 3 where I present the modifications done to enable partial reconfiguration on our selected reconfigurable platform. I begin with a discussion about the interfaces provided by Xilinx to configure their FPGAs and go into particular detail about the Internal Configuration Access Port(ICAP). Subsequently I discuss modifications that were applied to the Xilinx bitstream format before it could be used to configure the FPGA through the ICAP. I also discuss the development of the ICAP Controller and how the size of the Read FIFOs affected our design. Then I focus on floor planning of components for our platform and the final plan I settled on for getting optimum timing results. This is followed with a discussion about compiler support for partial reconfiguration and driver modifications done for non blocking execution of the calling process. I finally talk about hardware generation using the ROCCC compiler and wrapping ROCCC systems and modules through the *wrappergen* program. The chapter concludes with a discussion of the optimizations done to extract the best possible performance from our system.

Ultimately in Chapter 4 I describe the secure audio processing application used to evaluate my work. I discuss the Audio and AES cores and the resources occupied by them. Then I move on to a discussion about the execution time obtained for different input data sizes and the variation of speedup with the same sizes. I also compared five different prefetching techniques and the performance each of them deliver. Towards the end I delve on the function used to test the CCU wrapper generated by *wrappergen* around the ROCCC output modules. I conclude with a comparison of the execution time for each kernel that is used to benchmark our design.

I conclude this thesis in Chapter 5 where I indicate it's contributions and provide directions for future research.

Background 2

In the previous chapter I discussed how applications can be accelerated in hardware and the challenges involved. This chapter discusses the HTX Platform as proposed in [17] and how it offers a new approach towards a general-purpose system with reconfigurable acceleration. Some of the novel features offered by this platform are discussed along with the mechanism used to integrate the FPGA and control it.

The HTX Platform is an implementation of the Molen polymorphic processor paradigm [18] utilizing an AMD processor and the HTX high speed interconnect. Therefore I will first present a brief discussion on it's features. Molen uses a PowerPC(PPC) processor connected to a Virtex-2 FPGA to accelerate functions. The PPC and the reconfigurable device can communicate data through a set of Exchange Register(XREGs). The Molen Polymorpic Processor is shown in Figure 2.1. It extends the instruction set of the PPC with eight instructions which are sufficient for accelerating applications on this platform. The instructions are internally executed using microcode. The system supports partial reconfiguration but does not support direct access to the PPC's main memory. Programs intended to execute on the Molen architecture are written in C. The functions that are accelerated in hardware are marked with a pragma user directive as shown below. This is based upon the technique used in the DelftWorkBench:

```
__attribute__((user("replace")))
void rijndaelEncryptData(..)
{
...
  ...
}
```

The HTX Platform is an implementation of the Molen architecture however it uses an AMD Opteron processor instead of a PowerPC. Furthermore the platform adds support for high bandwidth memory access using the HTX Bus. Common design issues in the design of Polymorphic Processors are also addressed such as:

- Instruction Set Extension: While Molen does a one time architectural of a given ISA comprising of 8 instructions, the HTX Platform does not require it. The functions for managing reconfiguration are handled through commands sent to the FPGA by the driver

- Parameter Passing: Our platform allows any number of parameters to be passed to the FPGA device through addresses in the virtual address space of a process running in the host. This is further explained in Section 2.1

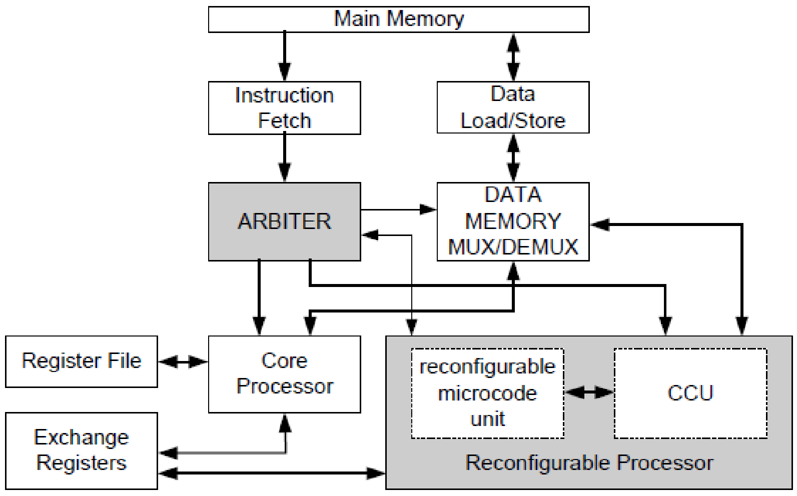


Figure 2.1: The Molen Polymorphic Processor.

- Modularity: The HTX Platform allows independently designed FPGA modules to be inserted through Custom Configured Units(CCUs). These are designed specifically for an application and can be easily inserted into the system.

The HTX Platform already supports acceleration of software functions through reconfiguration of the FPGA device. However it lacks support for changing the hardware configuration at runtime such that the functionality can be changed while a process is using it. This allows the process to utilize the FPGA for accelerating a new function without the need for halting it or restarting it. Thus the HTX platform was extended with support for dynamic partial reconfiguration. This required adaptations to the initial flat design in conformance with hierarchical design using the Hierarchical Design Methodology. The Hierarchical Design Methodology(HDM) is the most prevalent methodology used with Xilinx boards today and it replaced the Early Partial Reconfiguration flow used with earlier tools. HDM is used in our platform as it leads to a very clear cut hierarchical structure and allows automatic insertion of hardware logic to handle reconfigurable regions. This eliminated the need for explicit specification of bus macros. It's also required for specifying reconfigurable partitions which is needed to use DPR with Xilinx tools. Some of the terminology associated with the HDM flow is also presented.

This chapter is split into four sections. In Section 2.1 I discuss the HTX Platform and some of it's novel features. Then in Section 2.2 I provide an introduction to the basic premise of dynamic partial reconfiguration and current design flows associated with it. Subsequently in Section 2.3 I attempt to put my work in perspective of current developments in other parts of the world by describing some existing commercial and research machines. I particularly discuss the machines from SGI Altix, the Convey Computer, XtremeData, SRC Computers and DRC Computers. Finally I touch upon the state of current research in C-to-VHDL compilers and a few existing solutions in

Section 2.2. These include the DelftWorkBench, Handel C, Catapult C, SRC's Map compiler and especially the ROCCC compiler. I conclude this discussion on background work in Section 2.4.

2.1 The HTX Platform

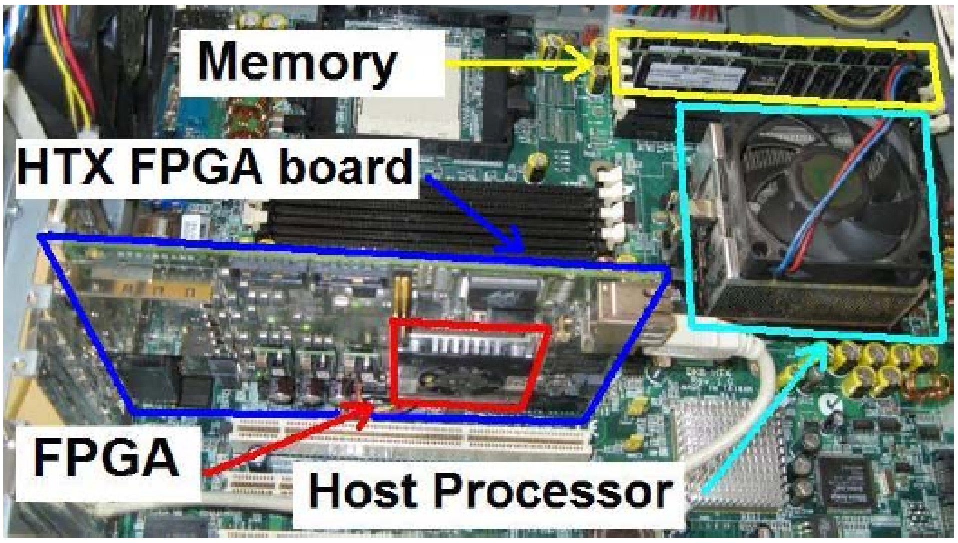


Figure 2.2: The HTX Platform implementation.

The HTX Platform shown in Figure 2.2 consists of an AMD Opteron-244 1.8 Ghz 64-bit processor and 1 GB of DDR memory connected to an IWILL DK8-HTX motherboard. The motherboard has an HTX bus with an HTX connector which is used to connect a Xilinx Virtex 4-100 FPGA device. For interfacing the FPGA with the HTX bus the HTX module developed at the University of Mannheim [16] was used. It allows 3.2 GBytes/sec throughput(unidirectional, 16-bit link width) with up to 32 concurrent DMA requests. The system runs on a Gentoo Linux Kernel 2.6.30. The FPGA contains the accelerator for computationally intensive functions in hardware. Such functions are annotated in the code by the user beforehand to indicate that they should be executed in hardware. The compiler is then responsible for emitting the proper binary with the hardware configuration (bitstream) of the respective function embedded within it. The configuration bitstream and the function arguments are sent to the reconfigurable device, which subsequently performs the reconfiguration, executes the function and returns the result. Figure 2.3 shows an overview of the HTX Platform.

The Reconfigurable device (FPGA card) is connected to the host general purpose processor and to the main memory through the Hyper Transport bus with a bandwidth of 3.2 GBytes/sec throughput(unidirectional, 16-bit link width). On the host side, a Linux driver has been developed to support the communication of the FPGA device with the host and the main memory via system calls and interrupts. On the reconfigurable

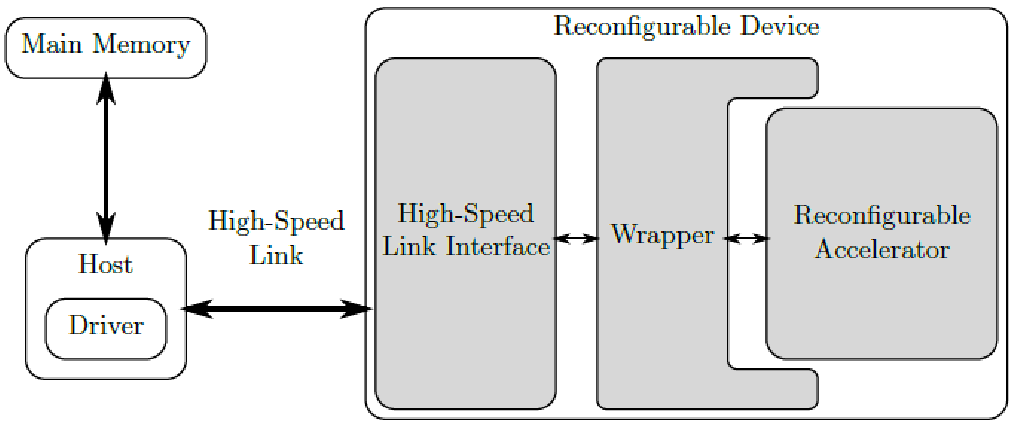


Figure 2.3: Overview of the HTX Platform.

device a module is used to interface with the high-speed link. The reconfigurable device further contains the reconfigurable accelerator intended to speed-up a software function. Various accelerators can be designed depending on the application running in the host. Finally, a wrapper has been designed around the reconfigurable accelerator to support the integration of the FPGA device in the platform.

Figure 2.4 offers a more detailed view of the modules designed in the FPGA device and in particular the internals of the hardware wrapper. The IO module controls all the memory-mapped IO of the device while DMA reads and writes are managed in the wrapper through the address translation module. Address translation uses a TLB copy to translate virtual addresses to physical before each DMA request. The ICAP controller handles dynamic reconfiguration of the accelerator. It receives a bitstream through the HTX bus and feeds it to the ICAP to perform partial reconfiguration. Finally, the Interrupt Handler (IH) manages the interrupts raised by the device. Therefore the wrapper does the following:

- controls the reconfigurable accelerator; that is, it dynamically reconfigures the accelerator through the ICAP as well as initiates and controls it's execution;
- maintains some memory-mapped IO regions: exchange registers (XREGs), a TLB copy, as well as control and status registers; the control registers are used to pass commands from the host 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 also to return execution results back to the host;
- handles interrupts generated by the FPGA device for various reasons, via the interrupt handler (IH);

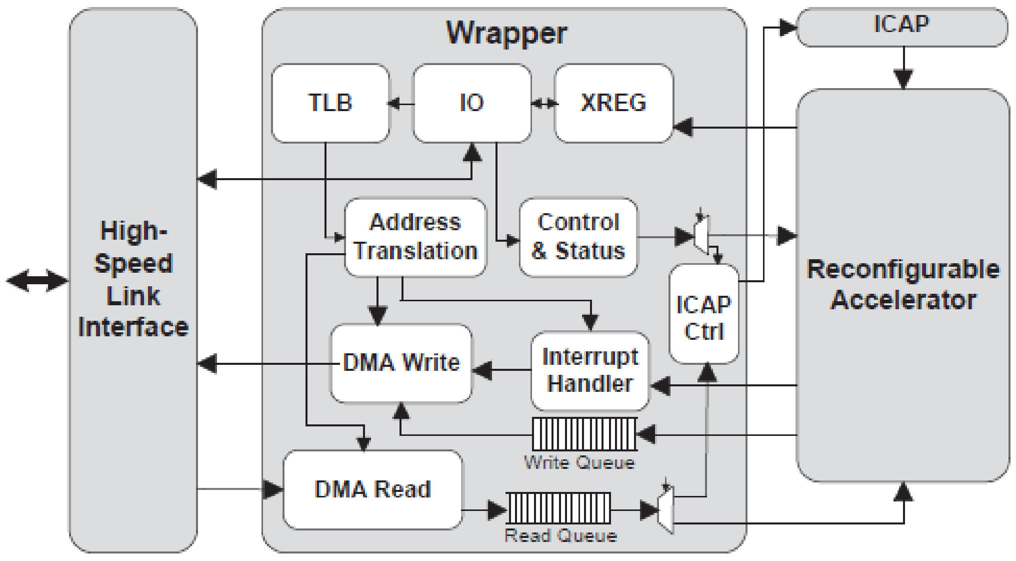


Figure 2.4: Modules implemented in the FPGA.

- performs Direct Memory Access(DMA) operations between the FPGA device and the main memory of the host;
- translates virtual addresses to physical using a local copy of the host processor's TLB and handles all TLB misses (by raising an interrupt).

The platform supports three different types of communication between the host, the memory and the FPGA device. The host can write and read data from the FPGA, the FPGA can send data to the host, and the FPGA can read and write data to/from the main memory. Their purpose and role in the platform is explained below.

- Host to/from FPGA: The host processor sends the arguments of the hardware-accelerated function to the FPGA and also reads the result of the reconfigurable accelerator. In addition, the host sends control signals to the FPGA in order to initialize and reconfigure the device as well as to start the execution of the accelerator. Finally, the host can write new entries to the TLB copy of the FPGA device to support address translation;
- FPGA to Host: The FPGA initiates communication with the host by raising an interrupt in case of a TLB miss, reconfiguration completion or execution completion;
- FPGA to/from Memory: The FPGA performs DMA reads and writes from and to the shared main memory in order to read the bitstream of a new accelerator, to feed the accelerator with input data and to write output data back.

In the following subsections, I first present the mechanisms used to control and integrate the FPGA device into the General Purpose System. Subsequently, I give an overview of the functionality of our platform and describe the way device initialization and hardware execution is supported.

2.1.1 Integration and Control of the Reconfigurable Device

The FPGA device is integrated in the platform through a Linux device driver. The driver provides an Application Programming Interface for software to interact with the device, it also performs the initialization of the device and handles interrupts caused by the device. The FPGA device is controlled via system calls, automatically inserted by a modified GCC compiler, while the configuration bitstreams are embedded in the binary code running in the platform. In addition, the hardware accelerated functions have direct access to the main memory, managed by the FPGA device.

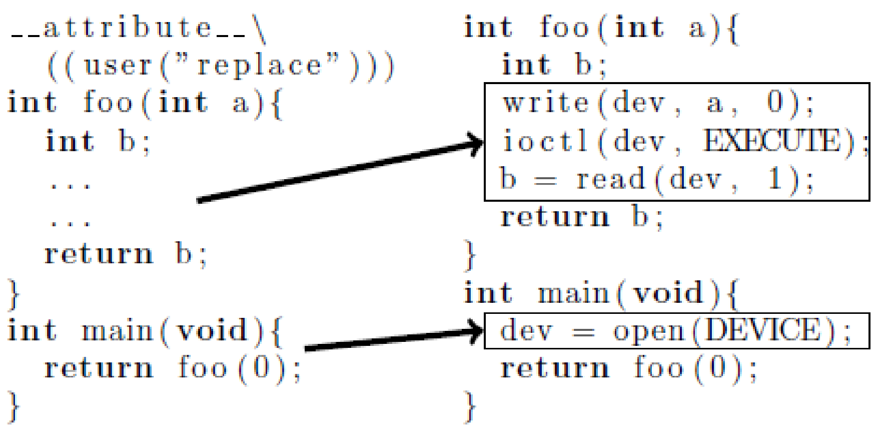
Controlling the FPGA device via System Calls: System calls are used to control the FPGA device from the host via the Application Programming Interface provided by the Linux driver. The following system calls are used to control the functionality of the FPGA device:

- int open(const char *pathname, int flags);
- int close(int fd);
- ssize_t read(int fd, void *buf, size_t count);
- ssize_t write(int fd, void *buf, size_t count);
- off_t lseek(int fd, off_t offset, int whence);
- int ioctl(int fd, int request, ...).

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. It is used to dynamically reconfigure the accelerator area of the FPGA (ioctl(SET)) as well as 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 [14], provides a more generic approach applicable to any system that supports a connection to an FPGA device.

Compiler Support: The programmer indicates the functions to be executed in hardware, by annotating them with GCC attributes: __attribute__((user("replace"))). The compiler can then automatically insert the appropriate system calls to the driver's API in order to achieve the hardware execution of the function. To do so, a GCC 4.5 compiler was extended with a plugin which scans the source code for such annotated functions. The body of such a function is then replaced with the required system calls.

Figure 2.5 illustrates an example of the system calls automatically inserted by the compiler to the original code in order to allow the hardware reconfiguration and execution of a function in the FPGA.



```
__attribute__\
  ((user("replace")))
int foo(int a){
  int b;
  ...
  ...
  return b;
}
int main(void){
  return foo(0);
}
```

```
int foo(int a){
  int b;
  write(dev, a, 0);
  ioctl(dev, EXECUTE);
  b = read(dev, 1);
  return b;
}
int main(void){
  dev = open(DEVICE);
  return foo(0);
}
```

Figure 2.5: System calls inserted.

First, the device is initialized (dev =open(DEVICE)), then, the hardware accelerator is configured (ioctl(dev, SET)),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)). It is noteworthy that the function arguments passed to the FPGA device cannot be more than 64-bit numbers, since this is the size of each FPGA XREG. This does not constitute a limitation however since the FPGA can directly access larger data structures in the host memory. All that is required is to pass the pointer or address of the structure as a function argument through the XREGs.

DMA: Direct Memory Accesses: The FPGA device accesses the main memory using DMA reads and writes. The Reconfigurable accelerator places memory requests and subsequently these are handled by the DMA manager located in the wrapper. The DMA manager consists of two parts, one for reading and one for writing data to the main memory. Read data are queued before being fed to the accelerator in order to concurrently serve multiple outstanding requests. Similarly write requests from the accelerator are queued before being served. Multiple requests can be active at the same time while the reconfigurable accelerator continues processing. Furthermore, multiple memory accesses to consecutive lines can be merged into one request reducing the packetization overhead. The DMA manager then needs to ensure that all memory requests are page-aligned; a single memory access cannot cross the page boundaries since this would cause a problem in the address translation. Using DMA with queues results in better utilization of the high-speed link and hides more efficiently the memory latency.

Address translation: The accelerated function operates in virtual addresses. Consequently, in order to retrieve the physical address of a DMA request, address translation is required. Address translation is performed in the wrapper of the FPGA. The virtual addresses are translated into physical using a local copy of the host CPU's TLB kept in the FPGA. The wrapper maintains a TLB copy of 512 entries. TLB misses are handled by interrupts raised by the device. In such a case, 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 then locked by the driver in memory, while those pages whose entries were replaced should get unlocked. During this process the driver should support memory protection. The reconfigurable device inherits the memory access permissions of the program that called it. The driver ensures that the FPGA device receives new TLB entries only for pages that it has permissions to read or write. Moreover, read-only pages are protected from write accesses. Another task of the driver is related to memory paging. The driver checks whether a page accessed by the FPGA device exists in the memory; in case a page is swapped out to disk, the driver loads it back in memory. Finally, the driver maintains cache coherency after the completion of a hardware-accelerated function. The above properties are implemented in the device driver using two kernel functions. The first function is get_user_pages() and provides memory protection and paging. In addition, get_user_pages() locks in memory all pages used by the device; alternatively the FPGA TLB copy would have to be updated each time a useful page were swapped to disk. Locking useful pages to the memory does not limit the total number of pages used by the device. The maximum number of pages locked by the device is equal to the number of FPGA TLB entries; it is however possible to use more pages than that by replacing -and hence unlocking existing old TLB entries. The second function used by the driver is pci_map_page() which is used for translating a virtual address to the current physical address and ensure the host remains cache coherent if a page is modified by the FPGA.

Interrupts: The driver receives interrupts from the FPGA device in three cases: on a TLB miss during address translation, when the reconfiguration of the device is complete, and when the accelerator has completed execution. An interrupt caused by the FPGA device is supported in the Interrupt Handler (IH) located in the wrapper. Since the High-Speed link interface supports only one interrupt identifier, I use a status device register to differentiate between the three types of interrupts above. This is achieved by storing a different interrupt-id in a status register located in the IH. The IH is further responsible for prioritizing multiple concurrent interrupts; this is possible when the device hosts multiple parallel reconfigurable accelerators. Sending interrupts is achieved by writing to a specific memory-mapped address using a write DMA operation. This memory-mapped IO location is setup by the driver during the initialization of the FPGA as a PCI device.

I next move on to the functionality provided by the FPGA.

2.1.2 Functionality of the Reconfigurable Device

The HTX platform uses the above described mechanisms to dynamically reconfigure and execute hardware functions, as well as to initialize the FPGA device.

1. Initialization of the device: The Linux driver is responsible for the initialization of the FPGA-device. When the driver is first loaded into the Linux-kernel, it registers the FPGA device-id. Then, in case the kernel detects the FPGA device, the driver

is notified and initializes the device as well as several related data structures within the driver. First the driver maps the memory-mapped IO regions of the FPGA device into the kernel's address space. This allows the driver to access these regions as if they were arrays in memory. Subsequently, the driver creates an interrupt-id which can be used by the device, and registers the Interrupt Handler with the kernel, providing the device with a memory location to be written to when triggering an interrupt. Finally, the driver creates an entry (device node) in the file system of the machine; this entry is then available to user-space programs in order to exploit the device's capabilities.

2. Dynamic Reconfiguration of Hardware Functions: The Dynamic Reconfiguration of the accelerator FPGA-region is initiated by the program, which locks the device. This is performed through the ioctl(dev, SET) system call. Further details are presented in the next chapter where I discuss the modifications made to the platform to add partial reconfiguration.

3. Execution of Hardware Functions: Execution of a hardware function is initiated by the program through the ioctl(dev, EXECUTE) system call. The parameters of the function are passed to the XREGs of the device using write system calls before execution starts. During execution the accelerator can perform DMA transfers and use the XREGs for data exchange. Upon execution completion the device raises an interrupt to notify the host, which then runs one or more read system calls to read XREGs containing results. Additional results of the function may be located already in the main memory, stored by the device through DMA writes.

In this section I discussed the HTX Platform and it's basic functioning. I now move on to dynamic partial reconfiguration. Before delving into the system level changes in the next chapter, an introduction to the term and its use in the real world is presented. Following this introduction, I present a few research and commercial machines which have implemented the concept of FPGAs as a hardware accelerator for applications. The chapter concludes with a discussion on automatic hardware generation from C code and a few existing compilers that support such automatic conversion.

2.2 Introduction to Dynamic Partial Reconfiguration

Modern FPGAs such as the boards from Xilinx and Altera support Partial Reconfiguration(PR). This allows a part of the FPGA design to be changed without affecting the operation of the remaining part. A design which allows a part of it to be reconfigured must therefore have at least two parts, a static part which is never changed and a reconfigurable part which will be altered. Furthermore Xilinx boards also support *dynamic* partial reconfiguration which allows the reconfigurable part to be altered *without halting* the static part of the design. The basic idea behind partial reconfiguration can be shown by Figure 2.6

The figure shows the static part of the design in white and the reconfigurable region A in black. Such reconfigurable regions can be defined by using tools such as PlanAhead. The design in the reconfigurable region can be modified by using either of the

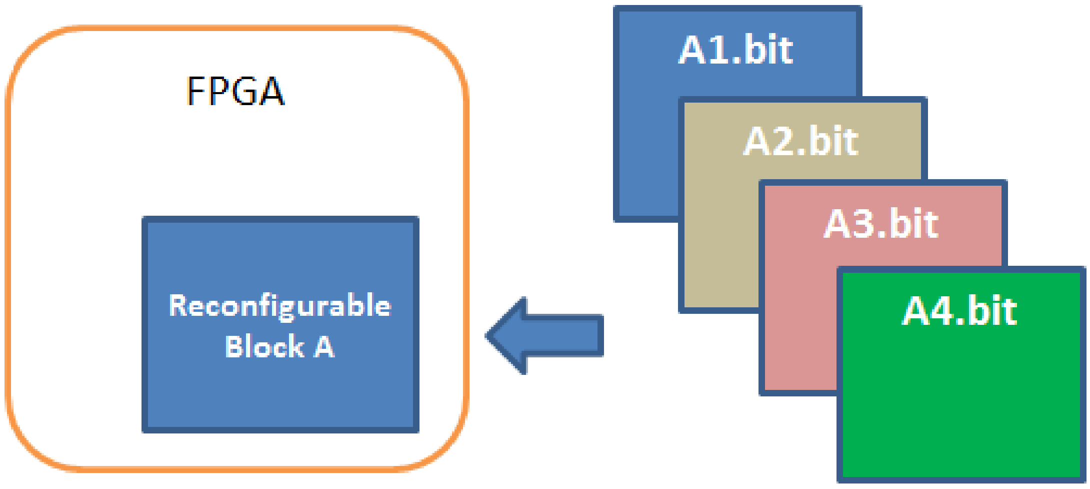


Figure 2.6: The basic premise of Partial Reconfiguration.

bitstreams shown to the right which are called *partial bitstreams*. Thus the functionality implemented in Reconfigurable Block A is modified by downloading one of several partial bit files, A1.bit, A2.bit, A3.bit or A4.bit which in hardware terms represent independent circuits. They can be created using appropriate vendor tools specific to the board. It is important to note that the static logic remains functioning and is completely unaffected by the loading of a partial bit file. The reconfigurable logic is replaced by the design specified in the partial bit file.

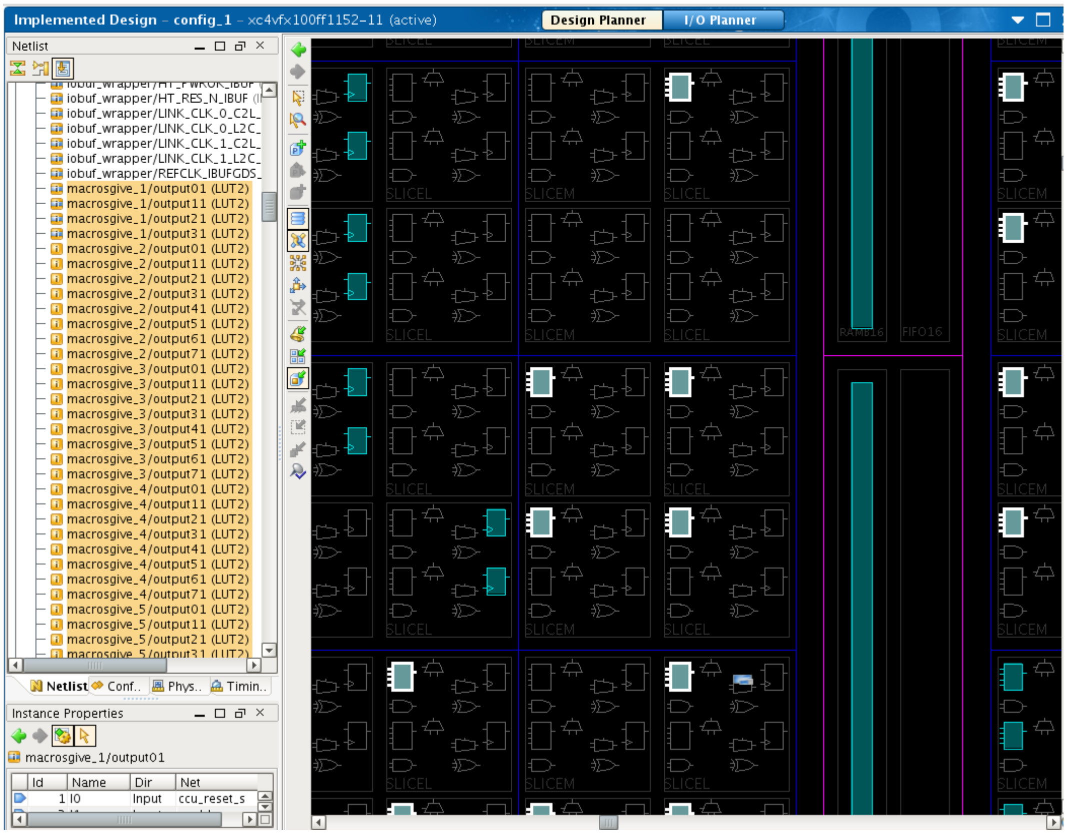
It is also worth mentioning that the design technique I have used with the HTX board is known as the Hierarchical Design Methodology(HDM) [19]. HDM is a design approach that leverages the natural logical hierarchy of a design to overcome the restrictions imposed by a typical flat design flow. Simply put, it means that HDM enables the designer to break up the design into smaller, logical blocks, allowing each major function to be worked on independently. These blocks are referred to as Partitions and they can be created using floor planning tools such as PlanAhead.

Therefore a Partition is a logical section of the design, defined by the user at a hierarchical boundary and which can be considered for design reuse. This allows a complex design to be broken up into smaller, more manageable pieces. Partitions create boundaries or insulation around the hierarchical module instances, isolating them from other parts of the design. At a hardware level this allows the circuits in each partition to be shutdown, removed and replaced with another circuit without affecting neighboring partitions.

Partitions are essential for a design that allows partial reconfiguration. To enable PR a design partition is marked as reconfigurable and is thereafter referred to as a Reconfigurable Partition(RP). Such a RP has a specific interface to the static part of the design which implies that the ports crossing the PR boundary are fixed. There can be multiple designs that can be accommodated in this RP and each such design must respect the port interface to the rest of the design. These designs are called Reconfigurable

Modules(RMs). Thus a single RP can have multiple RMs. A more complex design could have multiple RPs each with its own set of RMs.

To allow signals to cross the partition boundary, Xilinx synthesis tools insert Partition Pins. Partition Pins are the logical and physical connection between the static logic and the reconfigurable logic. They are automatically created for the ports present in the RP. In hardware the Partition Pin uses a single LUT1 element. In earlier uses of Partial Reconfiguration this logic which allows signals to cross the partition boundary in either direction; from RP to static or vice versa, was called as a Bus Macro. Its now been changed with automatic insertion of the so called Proxy Logic.



```
iobuf_wrapper/HT_PWROK_IBUF
iobuf_wrapper/HT_RES_N_IBUF (I
iobuf_wrapper/LINK_CLK_0_C2L_
iobuf_wrapper/LINK_CLK_0_L2C_
iobuf_wrapper/LINK_CLK_1_C2L_
iobuf_wrapper/LINK_CLK_1_L2C_
iobuf_wrapper/REFCLK_IBUFGDS_
macrosgive_1/output01 (LUT2)
macrosgive_1/output11 (LUT2)
macrosgive_1/output21 (LUT2)
macrosgive_1/output31 (LUT2)
macrosgive_2/output01 (LUT2)
macrosgive_2/output11 (LUT2)
macrosgive_2/output21 (LUT2)
macrosgive_2/output31 (LUT2)
macrosgive_2/output41 (LUT2)
macrosgive_2/output51 (LUT2)
macrosgive_2/output61 (LUT2)
macrosgive_2/output71 (LUT2)
macrosgive_3/output01 (LUT2)
macrosgive_3/output11 (LUT2)
macrosgive_3/output21 (LUT2)
macrosgive_3/output31 (LUT2)
macrosgive_3/output41 (LUT2)
macrosgive_3/output51 (LUT2)
macrosgive_3/output61 (LUT2)
macrosgive_3/output71 (LUT2)
macrosgive_4/output01 (LUT2)
macrosgive_4/output11 (LUT2)
macrosgive_4/output21 (LUT2)
macrosgive_4/output31 (LUT2)
macrosgive_4/output41 (LUT2)
macrosgive_4/output51 (LUT2)
macrosgive_4/output61 (LUT2)
macrosgive_4/output71 (LUT2)
macrosgive_5/output01 (LUT2)
macrosgive_5/output11 (LUT2)
macrosgive_5/output21 (LUT2)
macrosgive_5/output31 (LUT2)
```

macrosgive_1/output01

Id	Name	Dir	Net
1	I0	Input	ccu_reset_s

Figure 2.7: Proxy Logic is realized using LUT elements as shown in this implementation view of our design in PlanAhead.

The HTX Platform was upgraded to use the newer Hierarchical Design Method. This allowed easier creation of RPs and more details could be observed during the synthesis process. It also allowed us to experiment with different floor plans for the RP. The details of the changes are presented in the next chapter. Now, I move on to discussing some real world machines which have integrated FPGAs successfully as hardware accelerators.

2.3 Reconfigurable High-Performance Computers

This section discusses how five existing machines have integrated an FPGA into a general purpose machine. They were chosen on the basis of their similarity with our platform and the various approaches utilized.

2.3.1 Convey

The Convey computer is an integrated system of hardware, software and execution models that accelerate single threads. The Convey HC-1 compute node is an Intel-based x86_64 server system with a dual socket motherboard. One socket contains a dual core Intel Xeon processor while the other is populated with a socket interface to a daughter board. It is shown in Figure 2.8

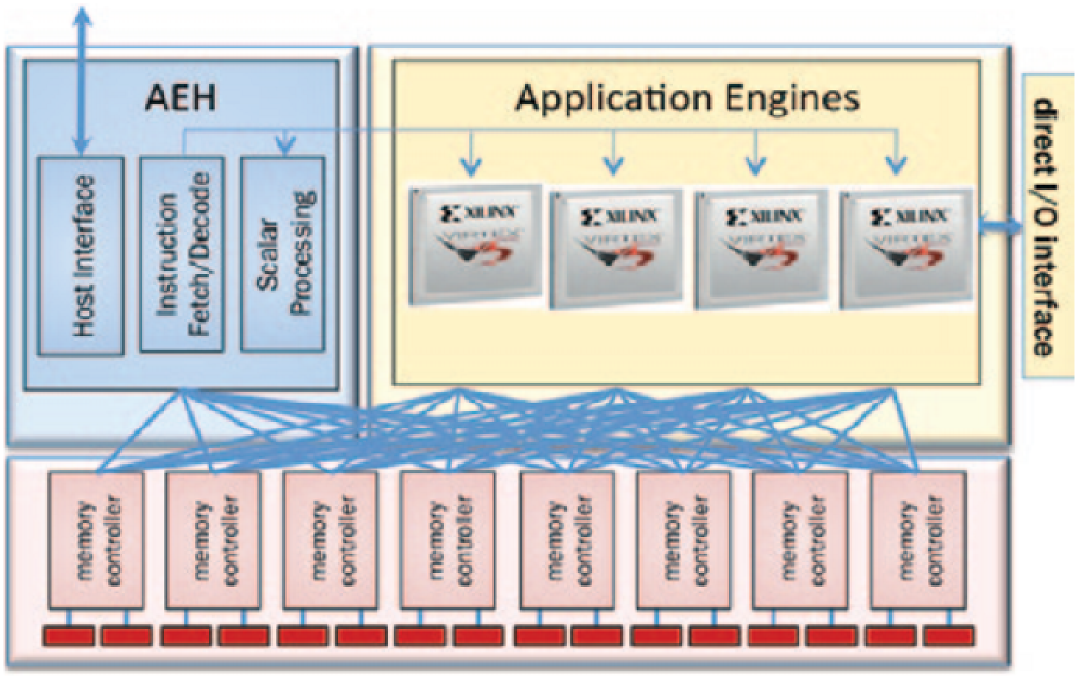


Figure 2.8: Convey Computer. Source: http://www.conveycomputer.com/

This daughter board contains four Xilinx Virtex-5 FPGAs which are the main hardware accelerators and are called Application Engines (AEs). The hardware accelerators are supported by sixteen DDR2 memory cards on the same board. To provide high bandwidth access to this memory from the host microprocessor, eight memory controllers are also integrated into the board. Moreover, a host computer interface, or Application Engine Hub (AEH) provides access to the application engines from the CPU. The AEH has access to a cache which is cache coherent with the host processor, acceleration units (FPGAs), and the Intel I/O hub which has access to all memory on the system. It's also responsible for managing reconfiguration of the AEs and initiating their execution.

The memory on the Convey daughter board along with the memory in the rest of the system exist in the same virtual address space. This eliminates the need for explicit transfer of memory blocks from the host processor to memory that is reachable by acceleration processors. The acceleration processors can reference all the memory on the host processor. Likewise, the host processor can access all memory physically located within the sixteen DDR2 DIMMs on the acceleration board. This requires significant bandwidth and is realized through the eight memory controllers on board which together support up to 80GB/s of bandwidth.

The Convey System extends the x86_64 instruction set with instructions that are implemented in hardware, specifically the Application Engines(AEs). This is achieved through a Open64 based compiler which emits x86_64 code for both the host processor and the instructions intended to be executed by the acceleration processors. At runtime when an application arrives at the first accelerated instruction, the AEH signals the Application Engines to reconfigure themselves into the appropriate state for running that instruction. The next instruction may be a different accelerated instruction and this prompts another reconfiguration. The machine claims that reconfiguration occurs in real time with very little latency to the application. The system is programmed in C/C++ or FORTRAN.

To abstract away the underlying electronic and hardware details, the system provides a series of pre-built, common hardware personalities. These are basically hardware configurations for groups of common instructions that are ready for use by a programmer. They can be programmed into the machine at runtime.

2.3.2 SGI Altix

The SGI Altix is a computing machine targeting high performance computing applications. It provides hardware acceleration using two Virtex-4 FPGAs along with eight Itanium-2 CPUs and 16 GB of main memory in the host. The program that provides access to the underlying hardware is the Reconfigurable Application Specific Computing (RASC) program. Connectivity among the system components is achieved through the SGI NUMALink high speed interconnect which provides low latency and high bandwidth. The link interconnects the FPGAs as a co-processor as well as the Itanium CPUs and Static Random-Access Memory(SRAM) chips. The system provides a device driver to allow accessing the FPGAs through system calls. Furthermore there is a library provided to enable even higher level access from user programs called as RASC Abstraction Layer(RASCAL). The architecture of the SGI Altix is shown in Figure 2.9.

2.3.3 XtremeData DISC

XtremeData makes a module, called the In-Socket Accelerator(ISA), which is pin compatible with a CPU socket from Intel or AMD. Their product is the XD2000i and since it sits in a CPU socket it allows the module to be a peer to the CPU and handle interrupts. The module also uses HyperTransport which offers multiple links which are 16 bits wide and allows transfer rates of 800MT/s(Mega Transfers per second). The platform is based on the loosely coupled Massively Parallel Processing(MPP) approach.

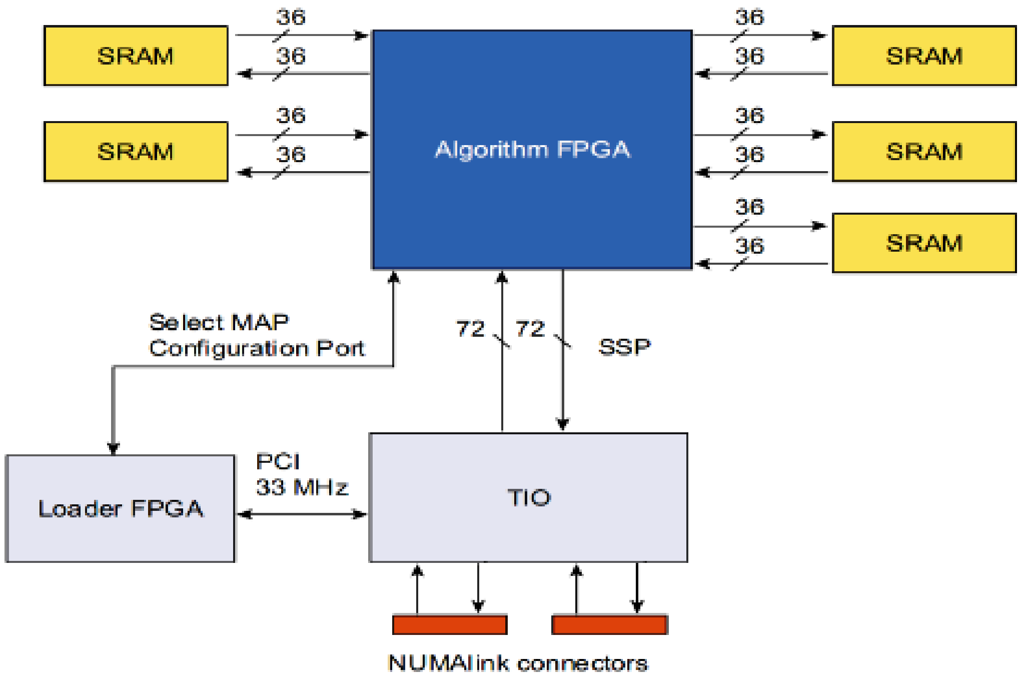


Figure 2.9: SGI Altix architecture showing how an FPGA is connected to the NUMAlink interconnect and SRAM chips. Source: http://www.sgi.com/products/servers/altix/numalink.html

The use of the XD2000i has been explained through the construction of a commercially available Data Intensive Supercomputer, the XtremeData DISC. The primary acceleration component of this system is the XtremeData dbX which is an FPGA-Based device. The FPGAs are re programmable in-system but it is not clear from available manuals whether run-time reconfiguration is available. The system consists of nodes which are the places where computations actually occur. The nodes begin with the Head Node coupled to N data Nodes. The system is scalable by adding more nodes. Each node has its own CPU, FPGA and local storage. There can be up to 1024 nodes when scaling up the system and all these nodes are controlled by a co-ordinator node. There is a high speed InfiniBand interconnect between the nodes. Figure 2.10 shows a block diagram of the XtremeData system.

The XtremeData system is used for data warehousing applications and is augmented with external storage. One such system intended for data warehousing uses 16 nodes with each dbX node having 2 FPGA accelerators. Important SQL operations that take too much time on the CPU are accelerated in the FPGAs. Data movement capabilities are also present in the hardware design along with statistics gathering. A MPP database engine is built on top of this hardware. The system's user interface is through Java Database Connectivity or Open Database Connectivity Drivers. This allows the system to be used without knowing low level VHDL programming. The XtremeData system is a case of an application specific machine giving far higher performance than a more

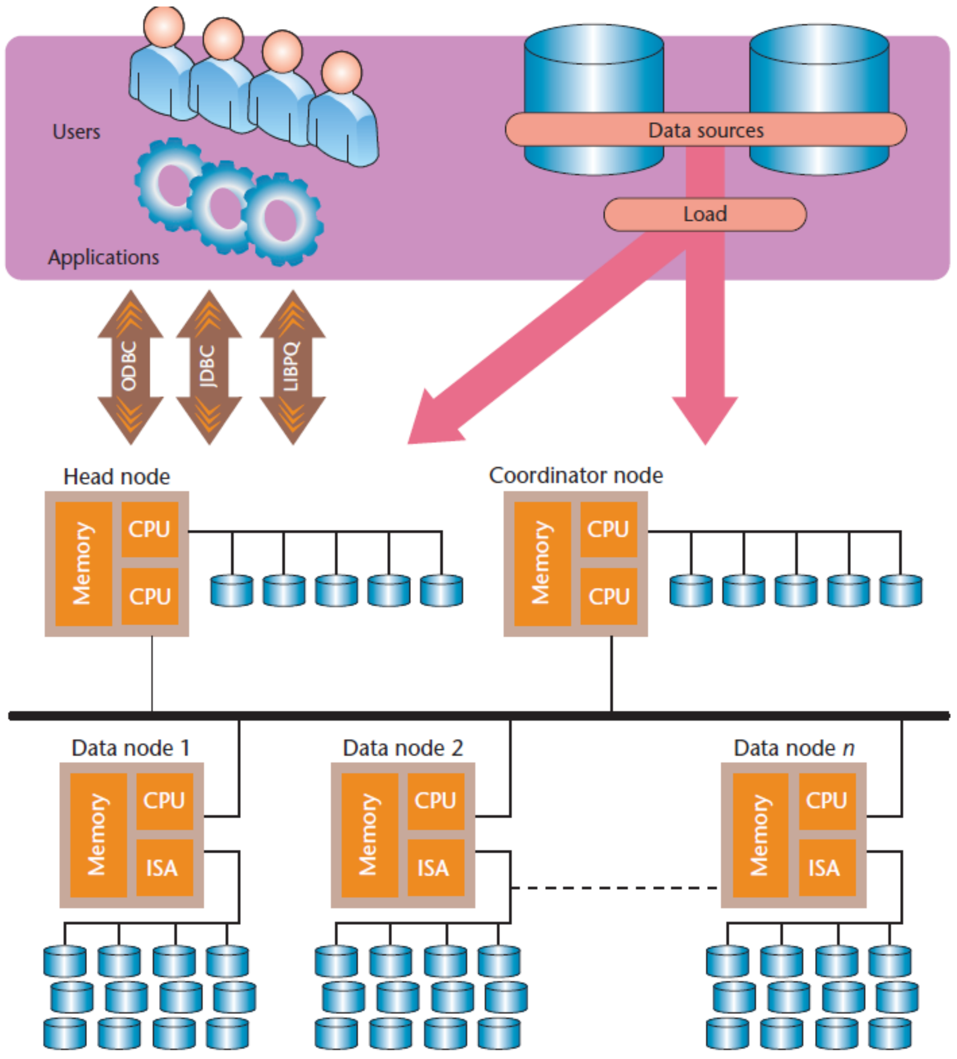


Figure 2.10: XtremeData Architecture. Source: http://www.xtremedata.com

general purpose computing system such as a Linux cluster. In this case SQL was run in hardware. The structure of the system also allows it to be scaled up to tackle much bigger amounts of data than would be possible through software alone.

The XtremeData system is claimed to have a low peak power consumption of 40 watts based on the FPGA in the ISA module.

2.3.4 DRC Computers

DRC Computers provides a commercial solution based on reconfigurable co-processors that can be plugged into standard sockets provided by AMD. The approach can be classified as a very tightly-coupled FPGA to CPU interface. Their machines are constructed to be like a standard PC enhanced with a RPU. The co-processors are designed to be directly plugged into a multi-way motherboard, which are boards that contain either 2 or 4 sockets (referred to as 2 way or 4 way respectively). Their DS2000 family of systems provides a complete computer with 1 reconfigurable processing unit or RPU and 1 CPU plugged into a Tyan 2 way board, or 2 RPUs and 2 CPUs plugged into a 4 way board. Generally the CPU used is the AMD Opteron and there are high speed HTX links to the CPU memory.

DRC's RPU110-L200 module has a tight coupling to the CPU due to it's location on one of the motherboard sockets and an independent memory controller provides fast access to main memory. In fact access to adjacent DDR memory and the Opteron processors are at Hypertransport speeds. They further claim that the RPU110-L200 combines a physical interface with a small footprint and this makes efficient use of space and power while reducing the number of nodes required to do computations.

The RPU is based off a Virtex-4 LX device and contains its own dedicated DDR RAM. The computers run a custom built operating system known as the Milano Hardware OS which enables runtime reconfiguration support without a system reboot. The average reconfiguration time is stated to be in the range of about 2 seconds.

DRC computers are generally targeted towards data intensive applications for the finance industry, time critical applications, oil and gas and biomedical markets. Applications that are search, sort, compress or encrypt intensive have their key algorithms in the RPU. Their aim is to replace Linux cluster environments with a scalable system that can handle high performance computing applications.

2.3.5 SRC Computers

SRC Computers makes the Series H MAP Processor which is a commercial FPGA-based reconfigurable processor. They describe the computational logic running in the FPGA as Direct Execution Logic(DEL) which is executed by a DEL processor. The DEL processor is realized using the resources of an FPGA. Instead of describing an algorithm through instructions intended to be executed in a traditional microprocessor, the DEL processor uses a FPGA's electronic resources directly to execute the logic. Since a FPGA based design is inherently parallel with each functional unit in the design capable of becoming active at every clock pulse, the algorithm can gain immensely from parallel execution without any software overhead. An unique DEL processor can be created for every application and represent the most efficient use of silicon for running that application's logic.

The MAP processors have been designed keeping in mind that it is efficient to run only parts of an application in hardware. Serial code such as those dealing with user interfaces, and file reads are best run in a traditional CPU. Therefore a DEL processor is run in a closely coupled configuration with a Intel Xeon CPU.

SRC systems thus consist of DEL processors and microprocessors. A tool chain is provided to automatically use hardware for a program's logic. The tool chain accepts a program written in a high level language such as C or Fortran and passes it to the MAP compiling system.

The Direct Execution Logic is combined with code running on the microprocessor. Interface logic is generated to co-ordinate the logic running in the MAP processor with logic running on the microprocessor.

Name of Machine	**Focus Area**	**Reconfigurable Hardware**
Convey HC-1	Acceleration of single threads	Dual Socket motherboard with a Intel-Xeon in 1 socket and daughter board with 4 Xilinx Virtex-5 FPGAs
XtremeData DISC	Data Intensive Applications	In-Socket Accelerator(ISA) pin compatible with a CPU socket from Intel or AMD
DRC Computers	High Performance Computing applications that are search, sort, compress or encrypt intensive	RPU110-L200 module based of a Virtex-4 LX device with dedicated DDR RAM plugged into the remaining sockets of a 2 or 4 socket motherboard
SRC Computers	Compute intensive applications in scientific research, oil and gas, defense	Direct Execution Logic processor realized using a commercial FPGA running in a closely coupled configuration with an Intel Xeon CPU.

Table 2.1: Summary of various machines which integrate a reconfigurable device and their focus areas.

In this section I discussed existing computing machines that integrate reconfigurable devices. The machines are summarized in Table 2.1. In the next section I move on to the software aspect of these machines. I discuss automatic hardware generation with a focus on C/C++ as the primary input language.

2.4 Automatic hardware generation

As can be judged from the previous section a large number of machines incorporating microprocessors and FPGAs are now available. The main difficulty lies however in programming them. Hardware is generally programmed in Hardware Definition Languages(HDLs) such as VHDL or Verilog. Software programmers are generally not familiar with these languages and are more comfortable in implementing algorithms in a High Level Language(HLL). Furthermore efficient usage of a FPGA's resources require a detailed knowledge of timing information, the components present and concurrency. These pose hurdles to hardware-software co-design and a steep learning curve to software designers. Yet another issue is that if sections of a large software code base is written in a HDL instead of the HLL in which the rest of the system is written, then there can be difficulties in integrating both parts into a smoothly working solution. Coding using an HDL is generally tedious and error prone to those who are not very proficient in it.

In spite of the above, C is one language which is equally familiar to both software and hardware designers. Also applications which target machines incorporating hardware accelerators, generally get written in C or C++. This has lead to wide exploration of techniques providing automatic conversion of C to VHDL. These techniques have been incorporated into popular compiler frameworks such as SUIF [20] and the more modern LLVM [21]. This has lead to increasing exploitation of hardware resources by software application developers and utilization of the large speedups provided by hardware through it's inherently parallel nature.

In the following subsections I describe a few current C to VHDL conversion tools that have appeared in literature or are currently in use.

2.4.1 DWARV

The DelftWorkBench automated reconfigurable VHDL generator [1] provides a C-to-VHDL generation toolset and was developed in TU Delft. It aims to provide a semi-automatic platform for hardware-software co-design using the DWARV toolset. The code output is meant to run directly on the MOLEN polymorphic processor. The tool accepts annotated C code and imposes a few restrictions on the C syntax. The annotations specify the functions which are intended to be accelerated in hardware. The toolset tries to harness the parallelism in the input C code. The current tool supports memory addressing in one dimension and does not support structures, unions and floating point data types. Iterations and conditions statements are limited to *for* and *if* statements.

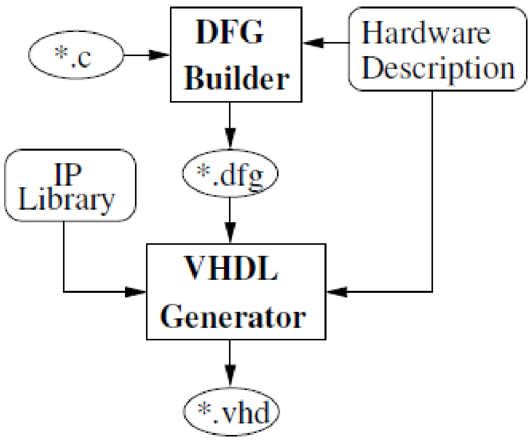


Figure 2.11: The toolset for the DelftWorkBench. Source: [1]

The DWARV toolset has two main modules:

- The DFG Builder and
- The VHDL Generator

The input code first encounters the DFG Builder as shown in Figure 2.11. The DFG Builder performs high level hardware optimizations and converts the code into a intermediate form(IR) that is more suitable for mapping onto an FPGA's resources. The high level optimizations done are simplified scalar replacement, static single assignment, common sub-expression elimination, and dead code elimination. The DFG Builder is implemented as a pass within the SUIF2 compiler framework. The output of the module is an IR in a form such that maximum parallelism is exposed for further exploitation during VHDL code generation. Thus it's output is a hierarchical data-flow graph (HDFG). A HDFG is a directed acyclic graph with two types of nodes: simple and compound. Simple nodes represent arithmetic and logic operations, registers, and memory transfers. Compound nodes represent loops in the input code and contain a sub-HDFG of the loop body. The edges of the graph represent data dependencies and the precedence order between operations as in a dependency graph.

The IR form is further processed using the VHDL generator module. This module carries out As Soon As Possible(ASAP) scheduling of the operations in the graph nodes. Thus it requires the latencies of the operations in the nodes as input. The latencies are in terms of CCU cycles and latencies below this is not currently allowed. The VHDL module also requires the available memory bandwidth and delays for memory operations as input. The output code is based on Finite State Machines(FSMs) and have an interface that matches with the MOLEN CCU interface.

2.4.2 Handel C

Handel-C is a high level HDL and has a lot in common with ANSI-C. Handel-C allows easy conversion of algorithms into a hardware implementation and also allows hardware designers the freedom to easily write functional descriptions of hardware systems. While Handel-C implements only a subset of the ANSI-C standard, it also includes a number of hardware specific constructs to ease development of hardware. Handel-C code can be parallelized by using the *par* construct. The *par* construct is an explicit command to the synthesis tools that all program statements within the par block may be performed in parallel. This allows the compiler to schedule operations in different functional units at the same time step. Handel-C produces code that can be applied to almost any FPGA.

2.4.3 SRC's MAP compiler and Carte

SRC's MAP compiler can accept programs written in a FORTRAN and/or C and produces an unified executable that runs on MAP processors. MAP processors are present in SRC machines and are basically realized from FPGA resources. The compilation system extracts as much parallelism as possible from the program and chooses the parts to be implemented in hardware. It then generates pipelined logic that is instantiated in the MAP processor. Furthermore the compiler generates interface code to co-ordinate data movement to and from the MAP processor and to co-ordinate the logic running in the microprocessor with the logic running in the MAP processor. The MAP compilation system is shown in Figure 2.12.

2.4.4 Catapult-C

Catapult C Synthesis is a commercial high-level synthesis tool by Mentor Graphics. It can generate VHDL or System C RTL descriptions from untimed ANSI C++ and System C code. The software algorithm is written as a bunch of C++ functions. A large

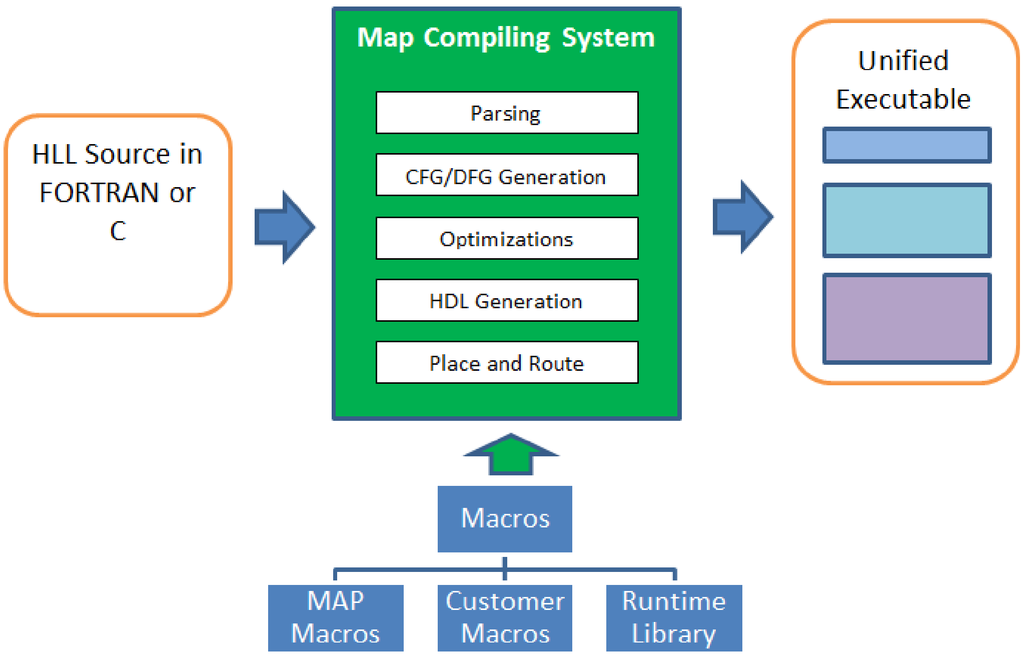


Figure 2.12: The MAP Compiler from SRC Computers. Source: [2]

number of architectural constraints are allowed including target specific input and output delays, power optimizations using clock gating etc. Loops can be unrolled or pipelined to meet timing constraints. The tool is also capable of showing a GANTT chart with the full datapath and times when the various functional unit are active. This allows finer scheduling of the design. Resources allocated to each part of the C code can be viewed or modified by a user before RTL generation. This includes I/O ports synthesized from the parameters of the top level C/C++ function. Other types of resources are channels, constant arrays (ROMs), and local arrays inside subprocesses. The compiler supports pragmas for design specific things that do not change much, such as loop pipelining and unrolling while directives can be used for constraints that change from solution to solution.

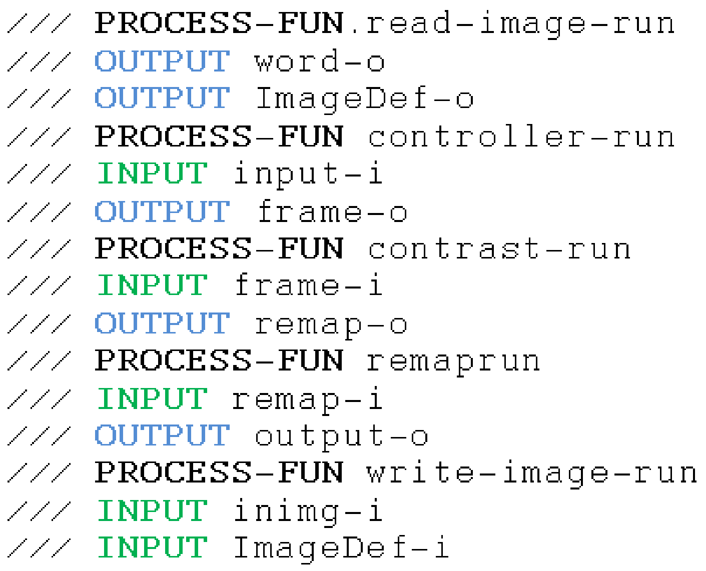
2.4.5 Stream-C

Stream-C is an open source C to VHDL compilation tool. It was developed to support the unique characteristics of stream oriented processing in the Streams-C system and is based on the SUIF C-processing framework. The compiler is thus optimized for the following:

- high-data rate flow of one or more data sources
- fixed size, small stream payload (one byte to one word)
- compute-intensive operations with low precision fixed point and

- access to small local memories holding co-efficients and other constants

The Streams-C compiler targets FPGA-based parallel computers and is capable of synthesizing circuits for multiple FPGAs with a multi-threaded control program that runs on a host processor . The Streams-C language is actually a small set of annotations and library functions callable from a conventional C program. The annotations are used to declare a **process, stream, or signal** and to assign resources on the FPGA board to those objects. A process is basically an independently executing object created by hardware generation from the C code of a subroutine. A process can run on the host processor or on an FPGA chip. For an FPGA process, the process body accesses only local data and is written in a subset of C supported by the Streams-C compiler. The process is allowed to call upon library functions to communicate stream data to other processes or send signals to other circuits. A typical process declaration in Streams-C is shown in Figure 2.13



```
/// PROCESS-FUN.read-image-run
/// OUTPUT word-o
/// OUTPUT ImageDef-o
/// PROCESS-FUN controller-run
/// INPUT input-i
/// OUTPUT frame-o
/// PROCESS-FUN contrast-run
/// INPUT frame-i
/// OUTPUT remap-o
/// PROCESS-FUN remaprun
/// INPUT remap-i
/// OUTPUT output-o
/// PROCESS-FUN write-image-run
/// INPUT inimg-i
/// INPUT ImageDef-i
```

Figure 2.13: Declaration of Process in Stream-C. Source: [3]

The system also provides a functional simulation environment based on POSIX threads, allowing a programmer to simulate the collection of parallel processes and their communication at the functional level. The Impulse C high level compiler is also based on Streams-C.

2.4.6 ROCCC

The Riverside Optimizing Compiler for Configurable Circuits (ROCCC) is a C to VHDL compiler designed at the University of California and supported by Jacquard Computing [22]. It supports modular design and generates code independent of any hardware. Hardware specific optimizations are left up to the user. No extensions are made to the C syntax to support hardware generation. The compiler is based on the SUIF and LLVM infrastructures and is open source.

The latest version of the compiler ROCCC 2.0 stresses on modularity and reusability. It implements these two concepts through the creation and integration of modules. Modules are generated from C functions that do not have use any memory references as one of their arguments. Thus these functions have only the standard C data types such as int, float or double as their parameters. They have a known number of inputs and outputs, some internal computation, and a known delay. However they cannot contain loops. Modules written in C can be kept in their original software function form or converted to a hardware function thus leading to hardware acceleration. These are then integrated directly into larger designs through standard C function calls which connects to the hardware instance.

To allow memory references, ROCCC supports a code structure called as a System. System's code are generally critical loops of applications that perform computations on large streams of data in memory. System code is therefore allowed to contain loops and reference memory through array accesses. Data reuse between consecutive iterations of the loop is detected and used to minimize memory fetches, with the necessary data elements being stored in a smart buffer [23]. If there is no data reuse, FIFOs and internal memories are generated to fetch and store data.

Name of Compiler	**Application Area**	**Target Hardware & Framework used**
DWARV	No restriction of the application domain	Targets the MOLEN platform, SUIF2 Compiler framework
Handel-C	Parallelism, general purpose applications	No specific compiler framework, targets all FPGAs
SRC's MAP compiler	Parallelism, automatic choice of hardware targets	MAP processor
Catapult-C	No specific domain, applications written in ANSI C/C++	Generates RTL(VHDL & Verilog) targeting ASICs and all FPGAs
Stream-C	Streaming data and compute intensive applications	21C
ROCCC	Critical region loop nests	All FPGAs, based on SUIF and LLVM

Table 2.2: Summary of various C to VHDL tools and their focus areas.

Moreover, the compiler also supports external cores to carry out specific functions in hardware. This can include floating point multiplication and division, Discrete Fourier Transform etc. The cores can be generated by hardware vendor's specific tools such as the Xilinx Core Generator. To use such tools inside modules or systems, the user needs to specify the VHDL interface and latency for the core. Thereafter the compiler is capable of automatically using the core whenever the related operation is seen in the C code.

The ROCCC compiler output is very well documented and it is easy to integrate into a hardware platform. Furthermore it generates FPGA agnostic code. These were the main reasons why I chose to integrate ROCCC into our tool-chain. Further details of how this was accomplished is presented in the next chapter.

The compilers discussed in this section are summarized in Table 2.2.

2.5 Concluding the discussion on background work

In this chapter I presented some of the background work and literature surveyed before carrying out modifications to the HTX Platform. I first presented an overview of the platform and it's distinguishing features. The mechanisms used to control the FPGA namely the device driver and system calls were discussed. I then gave a brief introduction to Dynamic Partial Reconfiguration and related terminology. Then I moved on to a discussion on existing machines which support high performance computing. I found that some of these machines also use the HTX bus and use dynamic partial reconfiguration to adapt to different application requirements. I concluded with a discussion on the techniques used by modern automatic hardware generation compilers. I specifically discussed DWARV, Stream C, ROCCC and a few others.

Now I move on to the parts of the system modified in the HTX Platform during the course of this thesis. In the next chapter I go into the details of these changes after a brief diversion to discuss the hardware behind partial reconfiguration in Xilinx boards.

3 System Level Support for Dynamic Partial Reconfiguration

This chapter discusses modifications done to the HTX platform during the course of this thesis. It covers the changes to the design to access the ICAP component on the Virtex 4 board, the ICAP controller, compiler modifications, automatic hardware generation and other optimizations carried out.

The organization of this chapter is as follows, in Section 3.1 I present the hardware available on the Virtex 4 FPGA to support dynamic partial reconfiguration. This section also discusses some of the commands in the Xilinx bitstream and the modifications done to it for use with the ICAP. Next, in Section 3.2, I discuss the modifications done to the GCC compiler via a plugin to support partial reconfiguration. I also explain changes carried out to the hardware PCI driver to accommodate large bitstreams and allow non blocking execution. Subsequently I present automatic generation of CCUs from C code via the ROCCC C-to-VHDL compiler in Section 3.3. I especially focus on the *wrappergen* program which is responsible for correctly wrapping ROCCC output. The chapter concludes with a discussion of the optimizations done in the compiler to get increased performance in Section 3.4 and a summary of the matter presented in Section 3.5.

3.1 Dynamic Partial Reconfiguration on HTX

This section discusses the hardware used to carry out partial reconfiguration on the HTX Platform and floor planning of the basic design. The most important hardware component is the ICAP which supports a standard interface to configure any part of a design which has been marked as reconfigurable. To allow the reconfiguration to occur independently of the rest of the design, hardware logic is inserted automatically at the interface of the reconfigurable part. I now proceed to the operation of the ICAP.

3.1.1 The Internal Configuration Access Port(ICAP)

Virtex-4 devices such as the one attached to the HTX platform are configured by loading an application specific configuration data into an internal configuration memory. This configuration is in the form of a stream of bits. As the configuration memory is volatile, the device needs to be reconfigured each time the board is powered up. Configuration occurs through configuration words which are loaded into configuration memory on the ticks of a Configuration Clock(CCLK). Xilinx provides three configuration interfaces to configure the FPGA, namely:

1. Serial

2. JTAG

3. SelectMAP

The serial configuration involves loading one configuration bit per clock cycle as given in [4]. JTAG configuration overrides all other modes and is always available through a JTAG cable. However the last interface is the one that is most interesting for DPR as it's also used with the ICAP component on the Virtex-4 FPGA.

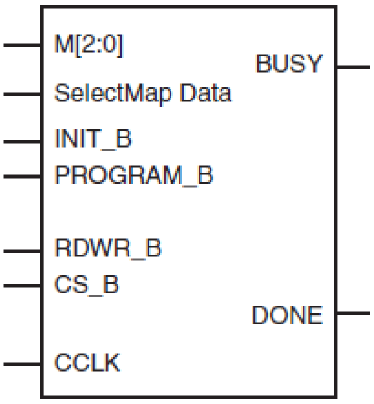


Figure 3.1: The SelectMAP Interface. Source: [4]

The SelectMAP interface is shown in Figure 3.1. It provides a 8-bit or a 32-bit bidirectional data bus. These are respectively called SelectMAP8 or SelectMAP32. It can be used to write to as well as readback configuration memory. CCLK is the configuration clock while M[2:0] is used to set the mode. SelectMap Data is the bidirectional data bus.

The Internal Configuration Access Port (ICAP) allows access to configuration data in the same manner as SelectMAP. ICAP has the same interface signaling as SelectMAP other than the data bus, which is separated into read and write data buses.

The ICAP interface as shown in Figure 3.2 is derived from the SelectMAP interface with a chip-select signal (CS), a read-write control signal (RD), a clock (CLK), a write data bus (DIN), and a read data bus (OUT). Furthermore the ICAP can be configured to two different data bus widths, 8 bits or 32 bits. I used the ICAP in 32 bit mode which is the similar to the SelectMAP32 interface. The ICAP interface can be used to perform readback operations or partial reconfiguration. When using ICAP for partial reconfiguration, it is necessary to avoid changing the logic or interconnect to which the ICAP itself is connected. This is ensured by keeping the ICAP Controller and the ICAP primitive instantiation in the static part of the design. Furthermore in the HTX Platform I needed to add timing constraints for th ICAP which lead to smooth partial reconfiguration.

The ICAP can also be used to read or write to the configuration registers, such as the STAT, CTL, or FAR registers. More details about these registers is provided in the section on the Xilinx bitstream format. This proved to be an indispensable feature early

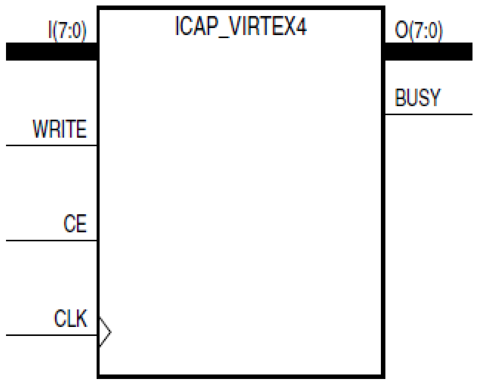


Figure 3.2: ICAP Interface. Source: [4]

on in the thesis when I was trying to achieve reliable partial reconfiguration through the ICAP. Using this, the entire configuration of the device can be read back using the SelectMAP interface through appropriate commands in the bitstream.

A brief summary about the location of the physical component and how it affected our design is in order. The ICAP is located in two sites on the Virtex-4 board with the condition that both cannot be active at the same time. Since the top one is used by default, I placed the ICAP Controller close to the ICAP to eliminate any chances of configuration failing due to timing issues in the data path.

I now move on to the bitstream format accepted by the ICAP.

3.1.2 Xilinx Bitstream Format

The default bitstream format used by Xilinx boards required modifications before it could be used as a partial bitstream for the ICAP. I will first explain the standard bitstream format as generated by the BitGen tool. I then go into the modifications I made to adapt it for use with the ICAP.

Virtex-4 configuration memory is arranged in frames that are tiled about the device. These frames are the smallest addressable segments of the Virtex-4 configuration memory space. I used the XC4VFX100 board which uses frames for various purposes as shown in Table 3.1:

Device	**Non-Config. Frames**	**Config. Frames**	**Device Frames**	**Frame Length**	**Config. Array Size**
XC4VFX100	1,660	25,170	26,830	41	1,031,970

Table 3.1: Virtex 4 frames used for various purposes. Source: [4]

The configuration memory of Virtex boards can be written into using some standard registers. The important registers relevant here are:

1. Frame Address Register;
2. Frame Data Register, Input (write configuration data);
3. Frame Data Register, Output(read configuration data);
4. Command Register;
5. Control Register.

A typical bitstream illustrates the use of these registers for writing to the configuration memory. A description of each command word is shown in Figure 3.3 below. This was generated through a C program written by me to clearly comprehend the format before attempting to change it.

```
Bitstream Header
00000000: 00 09 0f f0 0f f0 0f f0 0f f0 00 00 01 61 00 0a *.............a..*
00000010: 78 66 6f 72 6d 2e 6e 63 64 00 62 00 0c 76 31 30 *xform.ncd.b..v10*
00000020: 30 30 65 66 67 38 36 30 00 63 00 0b 32 30 30 31 *00efg860.c..2001*
00000030: 2f 30 38 2f 31 30 00 64 00 09 30 36 3a 35 35 3a */08/10.d..06:55:*
00000040: 30 34 00 65 00 0c 28 18 ff ff ff ff aa 99 55 66 *04.e..(.......Uf*
.................
Count  |HexCount|   DWORD   | Description
     1 |       1 | ffffffff | Dummy Word
     2 |       2 | AA995566 | Sync word
     3 |       3 | 20000000 | NO-OP
     4 |       4 | 30008001 | Type 1 write 1 word to CMD (Command Register)
     5 |       5 | 00000007 | RCRC command (Read CRC)
....
     8 |       8 | 30018001 | Type 1 write 1 word to ID (Device ID register)
     9 |       9 | 01EE4093 | Device_ID (Device ID register)
    10 |       a | 30008001 | Type 1 write 1 word to CMD (Command Register)
    11 |       b | 00000001 | WCFG command (Write Configuration)
    12 |       c | 20000000 | NO-OP
    13 |       d | 30002001 | Type 1 write 1 word to FAR (Frame Address Register)
    14 |       e | 000000C0 | Data value for Frame Address Register: 192
    15 |       f | 20000000 | NO-OP
    16 |      10 | 30004000 | Type 1 write 0 words to FDRI (Frame Data Register, Input - write configuration data)
    17 |      11 | 50000E41 | Type 2 write 3649 words to FDRI
    18 |      12 | 00000000 | Data word 1 (FDRI)
    19 |      13 | 00000000 | Data word 2 (FDRI)
    20 |      14 | 00000000 | Data word 3 (FDRI)
    ....
  3666 |     e52 | 00000000 | Data word 3649 (FDRI)
  3667 |     e53 | 30002001 | Type 1 write 1 word to FAR(Frame Address Register)
  3668 |     e54 | 000040C0 | Data value for Frame Address Register: 16576
  3669 |     e55 | 20000000 | NO-OP
  3670 |     e56 | 30004000 | Type 1 write 0 words to FDRI(Frame Data Register, Input - write configuration data)
  3671 |     e57 | 50000E41 | Type 2 write 3649 words to FDRI
  3672 |     e58 | 00000000 | Data word 1 (FDRI)
  3673 |     e59 | 00000000 | Data word 2 (FDRI)
  ....
 25590 |    63f6 | 00000000 | Data word 3649 (FDRI)
 25591 |    63f7 | 30008001 | Type 1 write 1 word to CMD(Command Register)
 .....
 25592 |    63f8 | 00000003 | LFRM command(Last Frame)
 25593 |    63f9 | 20000000 | NO-OP
 .......
 25693 |    645d | 20000000 | NO-OP
 25694 |    645e | 30000001 | Type 1 write 1 word to CRC Register
 25695 |    645f | 4786DFEF | Data value for CRC Register
 25696 |    6460 | 30008001 | Type 1 write 1 word to CMD (Command Register)
 25697 |    6461 | 0000000D | DESYNCH command(De-Synchronize)
 25698 |    6462 | 20000000 | NO-OP
 25699 |    6463 | 20000000 | NO-OP
```

Figure 3.3: The Xilinx bitstream format.

To summarize briefly, the bitstream is basically a sequence of 32 bit command words intermingled with 32 bit data words. It begins with a header followed by commands written to the command register to indicate that new configuration is being written. Most of the bitstream after this consists of commands to write frame data to configuration memory. This is achieved by writing the frame address into the FAR register followed by an appropriate number of words in the FDRI register. The bitstream ends with the Last FRaMe , desync and CRC check commands.

My modification to the bitstream was to discard the header and replace it with two 32-bit words. These two words contain the length of the bitstream that follows these first two 32-bit words. This helps the ICAP controller request the correct number of words from the main memory. I also inserted a large number of NOPs towards the end to ensure the bitstream is completely flushed to the ICAP. This was done to ensure that the configuration words preceding the NOPs were definitely loaded into configuration memory as any internal buffers filled up with incoming words. The extra NOPs would help in this and yet not affect or damage the board in any way. Apart from these changes, the partial bitstream was usable for dynamic partial reconfiguration.

The next sub-section deals with bitstream delivery to the ICAP using the ICAP controller developed by us.

3.1.3 The ICAP Controller

An ICAP controller was developed by us. It is capable of reading data buffered in the BRAM or directly through the HTX Bus. It is a simple FSM based controller that accepts the address of the bitstream in the host's main memory and starts fetching the configuration data when it receives a start signal. Figure 3.4 below demonstrates the working of the ICAP controller.

The initial design of the controller was based on configuring the FPGA by fetching the configuration data through the HTX bus and sending it immediately to the ICAP. However this approach led to the FPGA chip entering an error state as the configuration would not complete. I reasoned that this was due to the ICAP not seeing the words in the right endianess or in the correct bit order. However changing the order of the bits or the bytes in each word made no difference. The only other possibility was of the ICAP not receiving all the words to finish the configuration which meant that there was a delay in the delivery of the configuration words through the HTX bus. Since the ICAP requires configuration words at a constant rate, any interruption could easily cause failure in reconfiguration. I finally determined that the problem stemmed from low FIFO sizes of about 1kb used by the DMA Read Manager while fetching the configuration data. This became clear after I inserted the configuration bitstream in a BRAM located on the device and this lead to the configuration proceeding smoothly.

Since the FIFO size used with the DMA Read Manager was initially of the order of a few kilobytes while the bitstream size was considerably bigger at about 1 MB, thus the ICAP would run out of data during the configuration. It may also not receive configuration words for a few clock cycles. These were treated as NOPs in the middle of the bitstream and lead to incorrect configuration of the FPGA. Since FIFOs are also realized through BRAM resources, a simple increase of the Read FIFO to at least 64kb

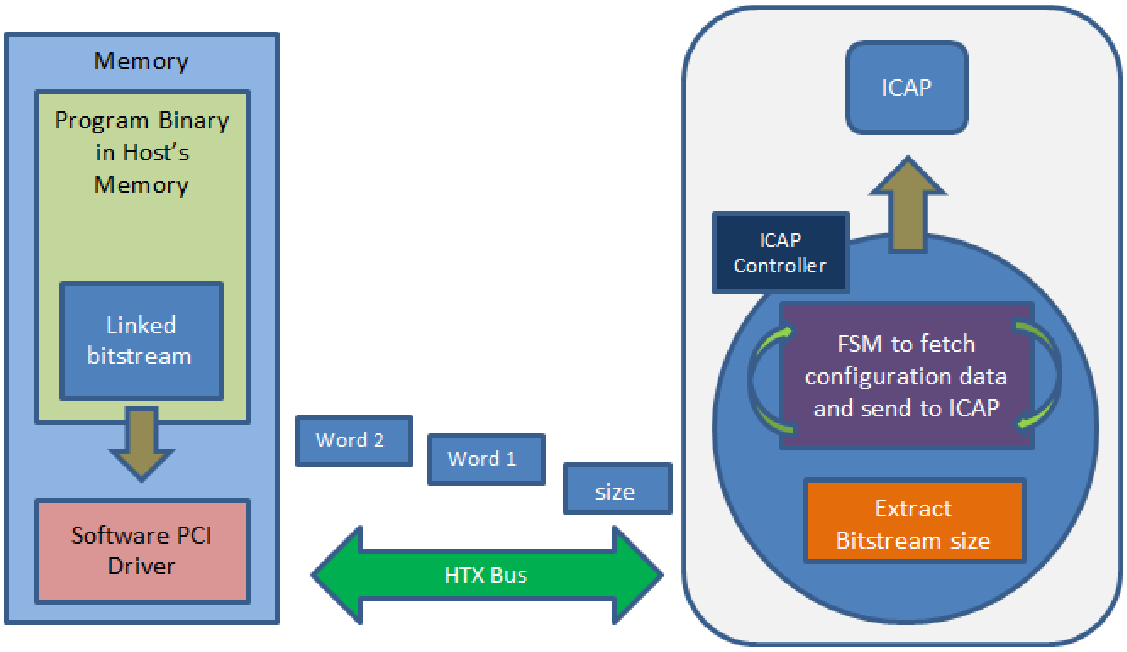


Figure 3.4: The operation of the ICAP Controller.

allowed the entire bitstream in the FIFO. This solved the partial reconfiguration issues I faced early during the project.

3.1.4 Partially Reconfigurable Regions on the HTX board

As discussed in the previous chapter, the Hierarchical Design Methodology(HDM) was employed to setup the HTX Platform for partial reconfiguration. This involved making partitions in the initial flat design with most of the main modules in the wrapper occupying separate partitions. The Custom Computing Unit(CCU) was the only Reconfigurable Partition(RP) in the design. A hierarchical view of the design is shown in Figure 3.5 below.

I designed a number of CCUs during the course of this thesis. These CCUs were the Reconfigurable Modules used for PR and could be swapped in and out of the single RP using the ICAP. PlanAhead 12.2 is the primary Xilinx tool I used for defining partially reconfigurable regions. The tools allows various Reconfigurable Modules(RMs) to be imported as NGC netlists. This implies that individual CCUs should be independently synthesized using XST or similar synthesis tool. A RM combined with the static logic forms a single hardware configuration. Thus there are as many configurations possible as there are combinations of RMs and the static logic. I had primarily two configurations one being the AES and the other the Audio RM. An entire configuration is mapped, place-routed and given to BitGen to generate the bitstream independently of the others.

I also experimented with a number of floor plans when I was trying to make PR

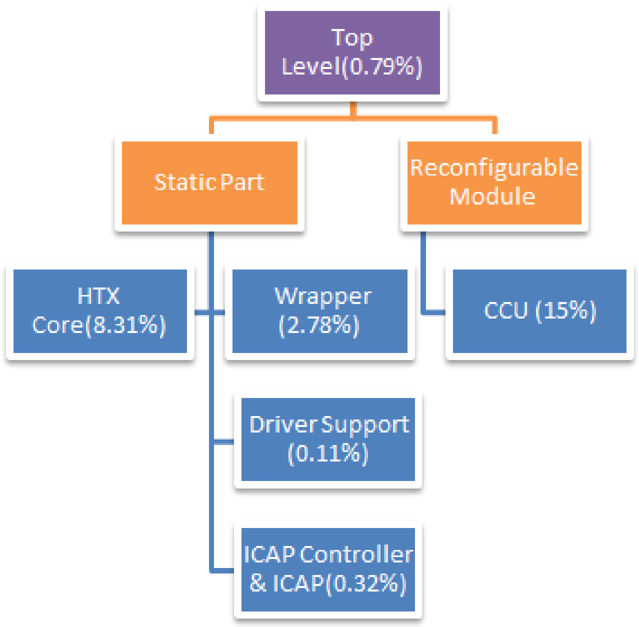


Figure 3.5: The hierarchy in our design.

work on the platform. Most of them involved placing the DMA Read Manager, the Read FIFO and the ICAP Controller as close to the ICAP as possible to alleviate timing issues. A particular challenge was accommodating very large designs like the Audio CCU discussed in the next chapter. This CCU requires significant BRAM resources and I was constrained by the fact that a RP can only be a single rectangular region. Furthermore it was important to not bring critical resources required by the HTX Core under the RP. The final floor plan that I settled upon is shown in Figure 3.6 below.

Another issue that I faced was during synthesis of the PR design. Certain floor plans or excessive floor planning lead to very poor timing scores while others often do not work at all. Certain components cannot be placed at specific places on the board and this is clear only after a failure of the Place-and-Route Process. Converging on the optimum floorplan that works every time is both a time-consuming and laborious process. However PlanAhead did provide a clear cut process with appropriate tools and Tcl/Tk command line support to allow us to change designs rapidly in case things did not work as expected.

This section discussed the changes that I carried out to the hardware design of the HTX Platform to add Dynamic Partial Reconfiguration. The next section delves into the software changes which go with the above to complete the system.

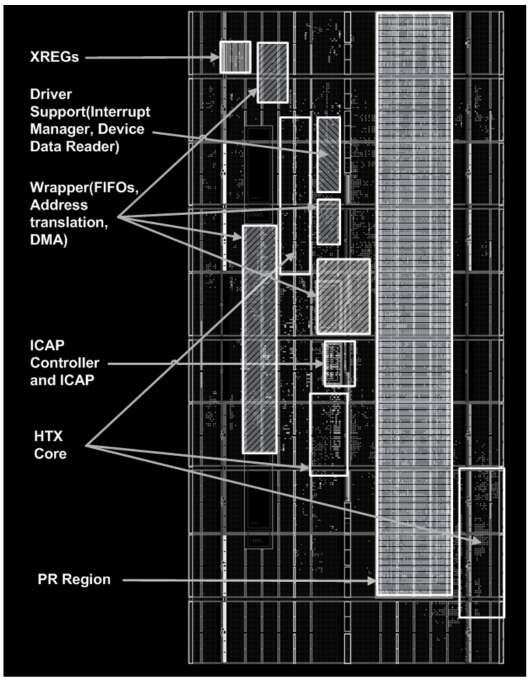

XREGs
Driver Support(Interrupt Manager, Device Data Reader)
Wrapper(FIFOs, Address translation, DMA)
ICAP Controller and ICAP
HTX Core
PR Region

Figure 3.6: The Floorplan for the HTX Platform.

3.2 Compiler Support and Prefetching

As discussed in sub-section 2.1.1, the compiler plays a significant role in completing the picture with regard to PR. An important step in the compilation process is linking the hardware bitstream to the binary executable. This has the advantage that the delay in reading the bitstream from a separate file is avoided as the bitstream gets loaded into the host's memory along with the program. Thus the reconfiguration latency is reduced. I achieved the linking using the objcopy program in Linux.

```
~# objcopy --input binary --output elf64-x86-64 --binary-architecture i386 ccu.bit ccu.o
```

When objcopy does the above conversion it adds the following linker symbols to the created object file: **_binary_ccu_bit_start _binary_ccu_bit_end**

This embeds the bitfile ccu.o inside the executable file which in Linux is in the ELF(Executable and Linkable) format and allows the bitstream to be accessed by the program during runtime to send to the FPGA. However for this to happen the declarations above need to be inserted into the global variable region of the program. Also proper function calls need to be inserted at the appropriate places in the C code. Before discussing the calls themselves I will talk briefly about how they were inserted in the code as it was a part of my thesis.

A GCC 4.5 compiler was modified via a plugin to achieve the modification in the code. To build the plugin I studied the process used by the GCC compiler to take C code text towards RTL code in preparation for conversion to binary. It is shown in Figure 3.7 below:

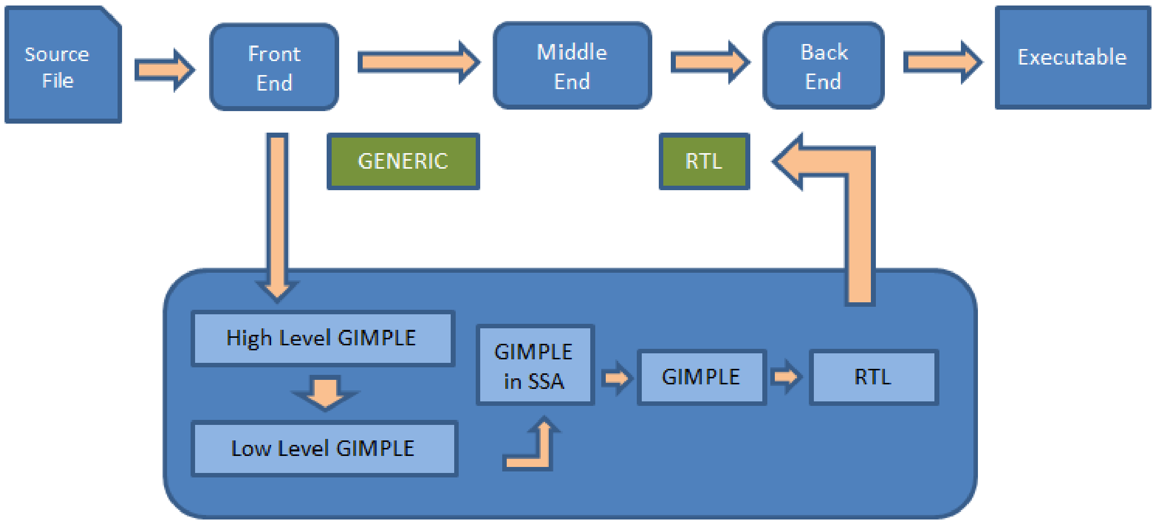


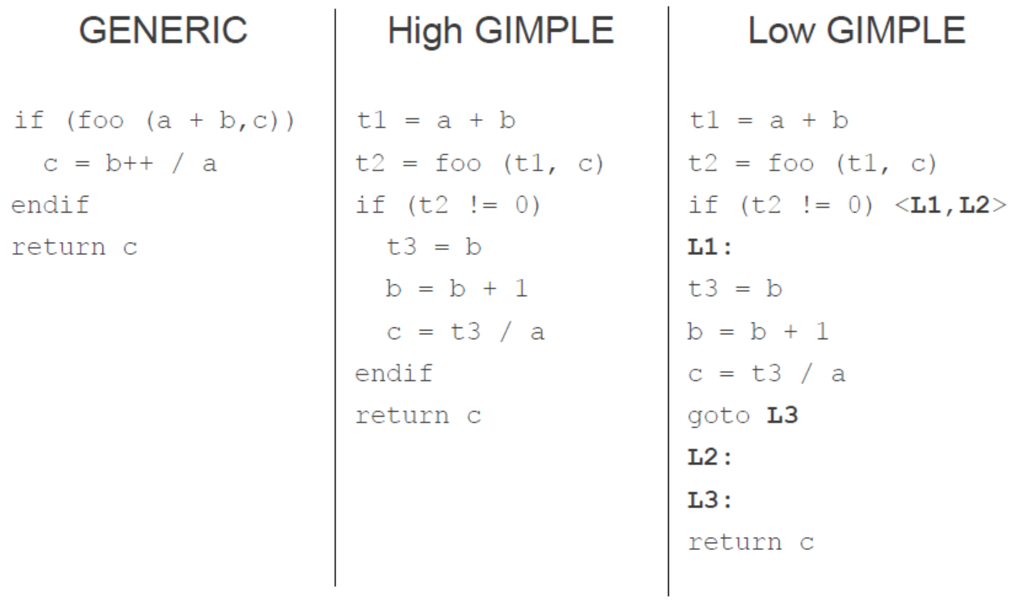
Figure 3.7: The GCC compilation process.

The only two formats that were relevant to me for the plugin were:

1. GENERIC: This is the format emitted by the code text parser. A code text parser may use any internal representation for its own use but at the end it must convert the code text to GENERIC. All GCC front-ends share this representation.
2. GIMPLE: This is essentially a simplified form of GENERIC. A restricted grammar is used in GIMPLE representation to simplify the job of code optimizers. Figure 3.8 shows how code looks like in this format.

The plugin modifies the tree representation used for the GIMPLE format and inserts extra edges to the Control Flow Graph whenever new function calls are added. The most important call required is the ioctl(address, SET) call used to start configuration of the FPGA device. The address parameter refers to the linker symbol _binary_ccu_bit_start whose insertion into the symbol table of the executable was discussed earlier. This ioctl call needs to occur prior to execution of a function in the hardware as the hardware may not be ready. I inserted this call at the top of main() to allow prefetching to start as soon as the process begins executing.

After the ioctl() call occurs at program runtime, control passes to the Linux PCI driver. The memory location where the configuration is stored is received in the address parameter. This address is written to the FPGA using a special register that the FPGA is able to read. This communication is setup using a memory-mapped I/O region at driver initialization. Now *the C program is suspended* and a signal sent to the FPGA using a control register to start configuring the reconfigurable partition. The signal ends up in the ICAP controller and it starts fetching configuration data from the address specified. The length of the bitstream is embedded as a header and is the first 64-bit word read by the ICAP Controller. It decides how many requests for 4KB data pages



GENERIC

```
if (foo (a + b,c))
  c = b++ / a
endif
return c
```

High GIMPLE

```
t1 = a + b
t2 = foo (t1, c)
if (t2 != 0)
  t3 = b
  b = b + 1
  c = t3 / a
endif
return c
```

Low GIMPLE

```
t1 = a + b
t2 = foo (t1, c)
if (t2 != 0) <L1,L2>
L1:
t3 = b
b = b + 1
c = t3 / a
goto L3
L2:
L3:
return c
```

Figure 3.8: The conversion of C code from GENERIC to GIMPLE.

will be made. Thus *the ICAP Controller never reads beyond the bitstream boundaries* in the host's main memory. Once all the pages containing configuration data is read by the ICAP Controller and sent to the ICAP, it sets a DONE signal which causes an interrupt to be sent back to the driver. The driver then wakes up the process running the user program and allows it to use the hardware.

The above approach is sufficient for supporting basic partial reconfiguration. However suspending a process reduces throughput as time is wasted in not executing other instructions which are not dependent on the accelerated function's output. This becomes quickly apparent when two or more functions are accelerated in hardware. It's vital to configure the FPGA in the background while the program is busy running normal software code. Then when control lands on the hardware accelerated function there is no time wasted in configuring the FPGA before it can be used. Hiding this latency is an important stumbling block in getting application acceleration with FPGAs and therefore I proceeded to modify the driver to prefetch the bitstream while the process owning the user program continues to run without blocking.

The basic hurdle in achieving non-blocking execution was locking of user pages in memory before giving their contents to the FPGA. Locking of user pages is important for preventing delays in hardware execution of a function due to the delay involved in reading in pages from secondary storage. It is even more important for partial reconfiguration because the ICAP component on the FPGA requires data at a constant rate especially if insufficient buffer e present on the FPGA. However the pages of a running process cannot be locked in memory or at least I was not able to do so.

My solution to this was to copy the bitstream to kernel space. For this to be possible a region of kernel memory needs to be pre-allocated, that can be directly accessed by PCI devices (recall that the FPGA is treated as a PCI device in Linux). This region is large enough to accommodate the largest bitstream size which was 1.1 MB in my case. The Linux PCI API function pci_alloc_consistent() allocates consistent memory and returns an address through which the CPU can reach this memory. When the user program attempts to configure the FPGA, I copy the bitstream data from user space into this pre-allocated kernel space memory. Then I give the FPGA the CPU address returned by pci_alloc_consistent() and signal the FPGA to start configuring. The signal ends up in the ICAP controller as before, which immediately tries to get the physical address corresponding to the virtual address of the bitstream.

Our design maintains a copy of the TLB in the host as well as in hardware BRAM in the FPGA as has been discussed in the previous chapter. Whenever an address translation is needed, the FPGA looks there first. It's possible to pre-fill this TLB with addresses which will not change during the program's run. Such a case exists for kernel space addresses because kernel pages are never swapped out.

I used this fact to my advantage by prefilling the bitstream addresses in the TLB copy of the FPGA and to compensate for the delay introduced by copying the bitstream from user space to kernel space. This allows the FPGA to immediately know the physical address of the bitstream in memory without incurring a delay in sending an address translation interrupt and having the driver write the correct physical address to the FPGA's copy of the TLB. Finally, the physical address is handed over to the DMA unit which takes care of transfer of data using the HTX bus. This design choice had the added benefit of not having to modify the existing address translation method for accessing data from user space memory.

As a result of the above mechanisms the HTX Platform allows partial reconfiguration in the background while the user process continues running. When running more than one CCU such as the AES and Audio CCU together in the same program, the bitstream of the first CCU can be sent to the FPGA at the beginning of the program before control lands at the hardware accelerated function call. This is called prefetching and it allowed me to hide the latency involved with configuring the FPGA to a large extent. I discuss the use of prefetching in different ways while evaluating it's benefits in the next chapter.

This section elaborated on the modifications made to the compiler and the Linux driver to support Partial Reconfiguration on the HTX Platform. The final section of this chapter follows and it will deal with automatic generation of hardware using the ROCCC compiler.

3.3 Hardware generation using the ROCCC compiler

In this section I discuss the integration of ROCCC into our platform. The goal for integration of ROCCC was to enable generation of a complete CCU from C code thus eliminating the overhead in terms of time and effort required for constructing a CCU by hand. Generation of CCUs for the HTX Platform is possible now through a C program called *wrappergen* written by me. This program generates a CCU wrapper for ROCCC generated VHDL code. The CCU wrapper's job is to read data from the host's main memory and supply it with correct timing to the enclosed ROCCC module. This is elaborated in Figure 3.9 and basically consists of a number of FSMs working in parallel.

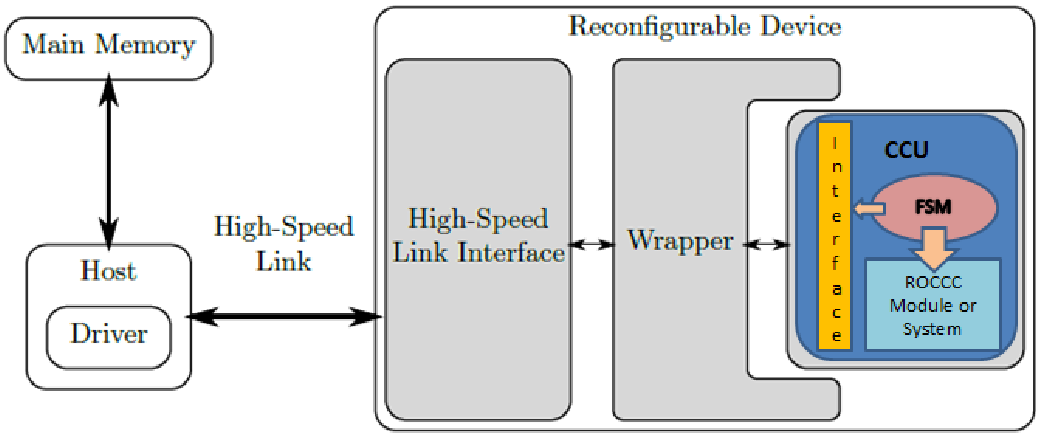


Figure 3.9: The *wrappergen* program generates FSM based VHDL code to manage the ROCCC output within the CCU.

Before discussing the wrapper generation itself I will present some details regarding the phases of compilation in the ROCCC compiler. Following that I will discuss how wrapper generation was added.

ROCCC is split up into a number of phases. These are shown in Figure 3.10 below:

The C code is accepted through the GCC C front end. A modified GCC 4.0.2 compiler then converts this text to the GCC GENERIC representation. A program written specifically for ROCCC then translates the GENERIC representation into SUIF(Stanford University Intermediate Format) as the next step, which does high level optimizations, requires the abstract tree in this format. The SUIF code of the program is then handed over to the Hi-CIRRF passes, where CIRRF stands for Compiler Intermediate Representation for Reconfigurable Fabrics. It is nearly identical to ANSI C with expressions and statements very similar to the original C code. The high level compiler transformations done in this step are the usual:

- Multiply and division by Constant Elimination;
- Loop unrolling;
- Systolic Array Generation;
- Temporal Common Sub-Expression Elimination and others.

At the end of the Hi-CIRRF passes, two files are generated which contain the C code converted to Hi-CIRRF. These are the roccc.h and the hi_cirrf.c files. The next step is the Lo-CIRRF passes which actually generates VHDL code representing the hardware.

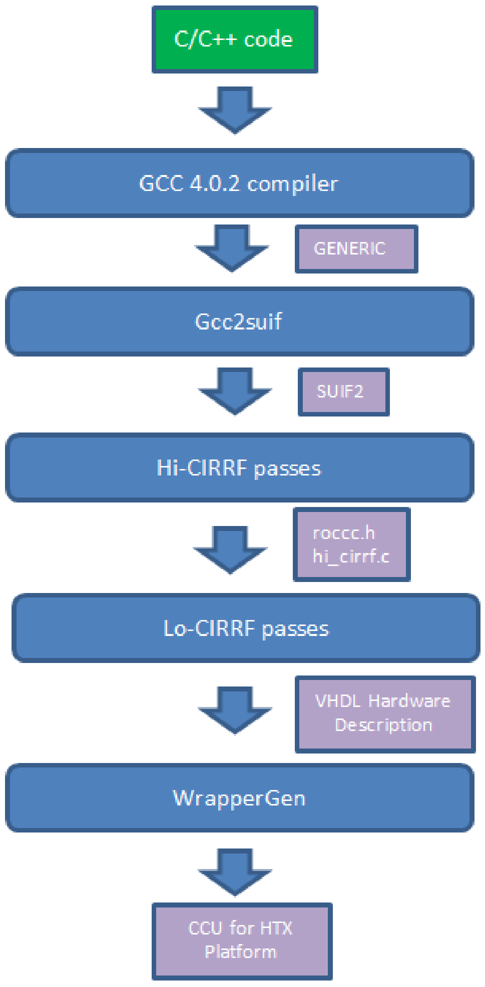


Figure 3.10: The phases in the ROCCC compilation process.

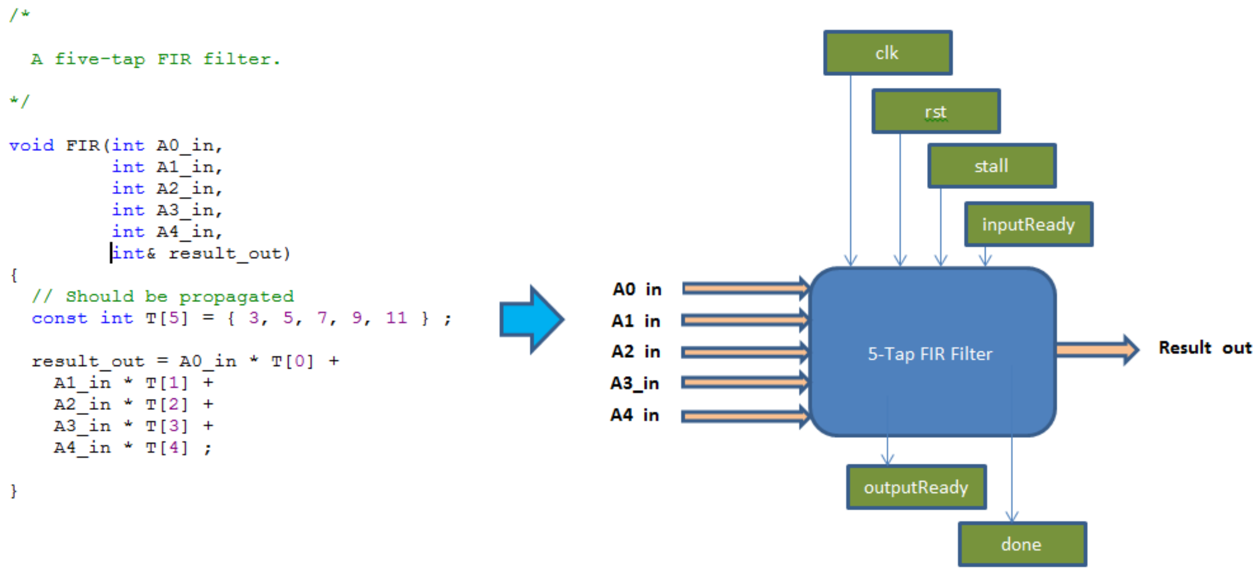
A number of Lo-CIRRF passes are now used to reduce hardware and pipeline the code. The Lo-CIRRF passes are implemented using the Low Level Virtual Machine(LLVM) toolset. Retiming also occurs in this phase. The output of the Lo-CIRRF phase is VHDL entities which can be wrapped appropriately for use in any FPGA based platform. Furthermore I have ensured the creation of a sqlite3 database which contains the details regarding the ports generated and the paths of VHDL files. This is used for generating a wrapper for the HTX platform.

As mentioned earlier in Chapter 2, ROCCC supports the creation of modules and systems. These are created from C or C++ functions only. Thus a complete program cannot be handed over to ROCCC for automatic profiling and hardware implementation of appropriate regions. Modules and systems are generated for two different types of functions distinguished on the basis of whether one of their parameters is a memory pointer or not. Each of these entities require a different type of wrapper and will therefore be discussed separately.

3.3.1 Wrapping ROCCC Modules

A module is generated from a C or C++ implementation function which does not operate on arrays or try to use pointers in any way. The parameters of such a function are passed by value and thus it needs to operate only within its local scope. In response to such code ROCCC generates hardware which always has the following six ports:

- clk;
- rst;
- inputReady;
- outputReady;
- done;
- stall.



```
/*

  A five-tap FIR filter.

*/

void FIR(int A0_in,
         int A1_in,
         int A2_in,
         int A3_in,
         int A4_in,
         int& result_out)
{
  // Should be propagated
  const int T[5] = { 3, 5, 7, 9, 11 } ;

  result_out = A0_in * T[0] +
    A1_in * T[1] +
    A2_in * T[2] +
    A3_in * T[3] +
    A4_in * T[4] ;

}
```

Figure 3.11: The VHDL top level entity generated for a ROCCC module for a 5-Tap FIR filter.

The above ports are shown in Figure 3.11 for a sample module which implements a FIR filter. The most important ports that are generated are the ones on the left and right side of the figure. These are the input and output registers. All single variable inputs in the function parameter list are converted to registers which are loaded through an unique input data port in the top level entity. The same applies for the output variables for whom output data ports are generated. Any internal variables used are converted to internal registers.

However the restriction of being unable to pass memory pointers has been eliminated through the program *wrappergen*. This program generates CCU wrappers for both modules and systems. A FSM based wrapper is created with one FSM dedicated to reading in the values for the ports from the host's main memory and another for writing the output values back to main memory. This wrapper is a part of the CCU in the larger HTX Platform design as mentioned earlier. The ROCCC modules form the other parts of the CCU.

A C program running on the HTX platform and trying to use a design which incorporates a ROCCC module needs to simply pass an array with the values of the function parameters present in it in the correct order. This array is read by the Read FSM in the CCU wrapper and the values are fed to the ROCCC module with appropriate timing to ensure the values get properly clocked into their registers. The output is similarly read back into an output array from the output ports. The only other inputs the calling program needs to provide are the number of inputs and the number of outputs such that illegal reads of the host's memory do not occur.

Wrapping a module was therefore quite straight forward. The next sub-section deals with wrapping ROCCC systems which directly try to access memory.

3.3.2 Wrapping ROCCC Systems

ROCCC systems are generated from C or C++ code which has a memory pointer as one of it's parameters indicating that the function code accesses and modifies values in memory. Such a case can arise from functions which read a single element array ,operate on it and write back the results immediately in a streaming mode of operation. Accordingly ROCCC treats array addresses as stream addresses that are used to access data in a loop. The operations inside the function results in output streams which are supposed to end up back in the memory. Normal variables which are called as scalar values are also allowed as the parameters of a stream oriented function and these are converted to registers as in the case of modules.

Figure 3.12 shows the C code intended to be a ROCCC System. The code simply checks if the passed points are within the bounds of a particular region and writes the results back. As indicated by the example, the C/C++ code for systems is assumed to have all the logic of the system inside the loop which accesses external data. This is important for proper optimization of the generated code and is the basis of ROCCC systems. The system logic can also be confined to a ROCCC module and then the module can be called as a simple function call from inside a ROCCC system. This lends modularity to a larger and more complex design. Additionally these conditions apply for code intended to be treated as a ROCCC system:

- Non-array variables declared inside a function that are only read get treated as input scalar ports;
- Non-array variables inside a function that are only written to are treated as output scalar ports;

- Variables that are both read and written to are treated as internal variables;
- If one of the above variables are read before writing to it, the hardware will generate an input port for setting its initial value;
- If the last thing done with one of the internal variables is a write to it, an output port is generated holding it's final value;
- Return statements are ignored inside the loop construct.

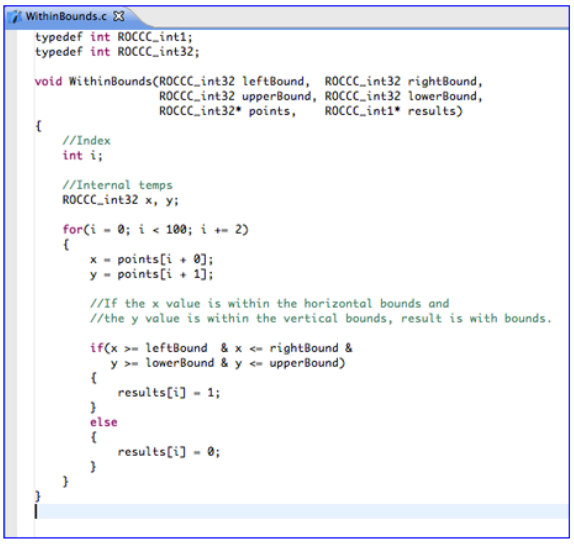
The iterations of the loop decides the size of the input and output streams and the appropriate amount of data must be supplied before the output can be produced.

Thus for a simple system such as the one shown in Figure 3.12, the hardware generated is shown at the right. The *points* pointer is made into an input stream and the *results* pointer into an output stream. The rest of the scalar values are turned into registers with an input port for each. Every input stream gets a standard set of interfacing signals to load data into it. These are the following:

- Write Clock In: clock used for writing to this stream;
- Channel In bus: bus containing the data to be loaded in the current clock cycle;
- Channel Address Out bus: bus containing the address from where the system is requesting data;
- Full Port Out: goes high if the stream is full;
- Write Enable In: has to be pulled high externally when data is ready in the Channel In bus;
- Write Stall In: has to be pulled high externally if the internal FSM should be stalled temporarily.

An output stream also gets it's own set of interfacing signals similar to the input streams

- Read Clock In: clock used for reading from this stream;
- Channel Out bus: bus containing the data that can be read in the current clock cycle;
- Channel Address Out bus: bus containing the address from where the data from the above data bus should go;
- Empty Port Out: goes high if the stream is empty and no more data can be read from the stream;
- Read Enable In: has to be pulled high externally when data should be placed on the data bus for reading.



WithinBounds.c

```c
typedef int ROCCC_int1;
typedef int ROCCC_int32;

void WithinBounds(ROCCC_int32 leftBound,  ROCCC_int32 rightBound,
                  ROCCC_int32 upperBound, ROCCC_int32 lowerBound,
                  ROCCC_int32* points,    ROCCC_int1* results)
{
    //Index
    int i;

    //Internal temps
    ROCCC_int32 x, y;

    for(i = 0; i < 100; i += 2)
    {
        x = points[i + 0];
        y = points[i + 1];

        //If the x value is within the horizontal bounds and
        //the y value is within the vertical bounds, result is with bounds.

        if(x >= leftBound  & x <= rightBound &
           y >= lowerBound & y <= upperBound)
        {
            results[i] = 1;
        }
        else
        {
            results[i] = 0;
        }
    }
}
```

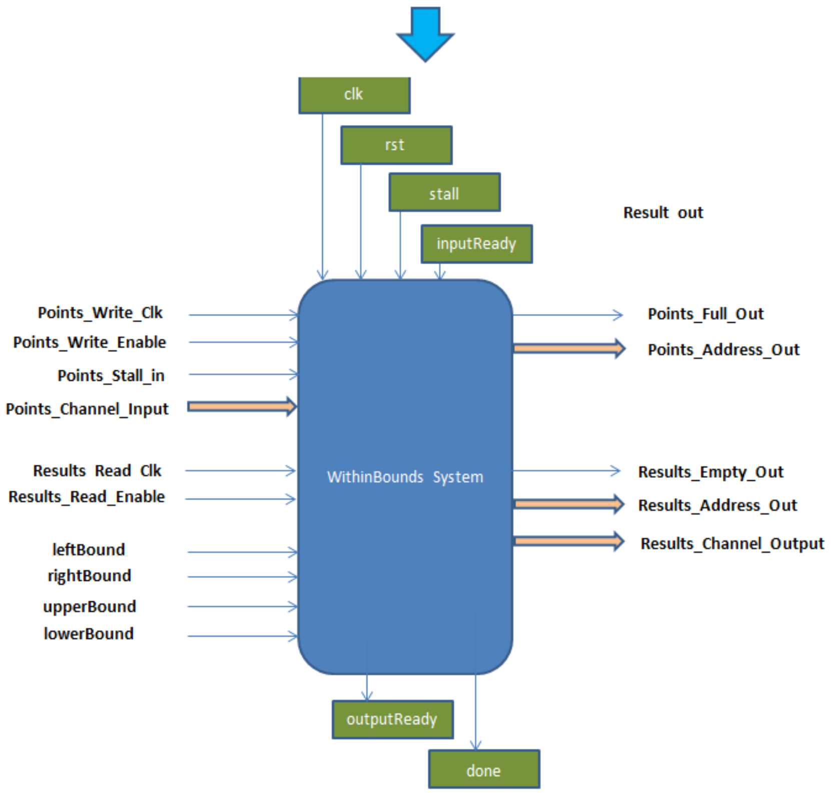


Figure 3.12: The VHDL top level entity generated for a simple ROCCC System called WithinBounds.

These interfacing ports are generated for each pointer variable in the function's parameter list. Thus a wrapper needs to interface with these signals correctly so that a stream gets loaded and unloaded at the appropriate time. Since the HTX platform supports simultaneous reads and writes I generate two FSMs one for loading data into the input streams and one for unloading results from the output streams. These run concurrently in hardware and synchronize with the ROCCC system using the provided interface. The output ports are similarly unloaded by a single FSM. This resulted in a delay in the operational speed of the ROCCC system which waits during each iteration for data to finish loading in all the input streams, and after the system logic completes, for data to be unloaded from all the output streams. However I carried out an optimization to eliminate this issue as discussed in the next section.

Now I move on to the various optimizations done after the modifications and features added to the HTX Platform, to get the best possible results.

3.4 Optimizations

A number of optimizations were carried out during the course of my thesis to extract the maximum performance from the HTX Platform. These optimizations related to the C code and to VHDL generation and are discussed in the sub-sections below.

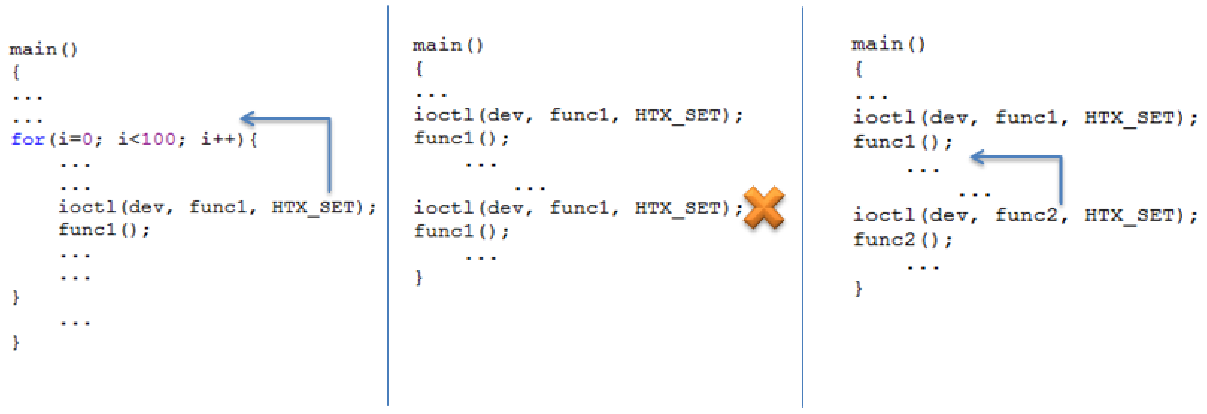
3.4.1 Optimizations on the C code

An important optimization was related to prefetching when using two different hardware accelerated functions in the same program. These optimizations related to hiding the reconfiguration latency of the FPGA before it can be used to execute the hardware accelerated function. Let us assume that there are two bitstreams for two different functions in the executable - say func1() and func2(). In the C code there may be a straight line of execution without any branches but containing two calls to say func1(). In that case there is no need to configure the FPGA twice. This optimization was extended to include the case where func1() is the only function within the bounds of a loop. This implies that multiple iterations of the loop call the function each time and the FPGA need not be reconfigured except just before entering the loop.

Yet another case is when there is a call to another hardware accelerated function func2() in between two calls to func1(). Then it becomes necessary to reconfigure the FPGA each time and a delay is incurred before the function can be executed. However an optimization can still be affected by starting the reconfiguration for func2() immediately after func1() returns. This increases the possibility that by the time control lands on func2(), the FPGA will be ready or at least some of the reconfiguration latency will have been overlapped with useful work. Conditional branches can lead to more complicated situations and as of now I insert a call to configure the FPGA in case such statements are present, to be on the safe side. These optimizations are shown in Figure 3.13.

3.4.2 Optimizations on the hardware wrapper for ROCCC output

There were also optimizations done while generating the wrapper for ROCCC modules and systems. One of them was parallely loading data for systems. All input streams can



```
main()
{
...
...
for(i=0; i<100; i++){
    ...
    ...
    ioctl(dev, func1, HTX_SET);
    func1();
    ...
    ...
}
    ...
}
```

```
main()
{
...
ioctl(dev, func1, HTX_SET);
func1();
    ...
        ...
ioctl(dev, func1, HTX_SET);
func1();
    ...
}
```

```
main()
{
...
ioctl(dev, func1, HTX_SET);
func1();
    ...
        ...
ioctl(dev, func2, HTX_SET);
func2();
    ...
}
```

Figure 3.13: Optimizations done by the modified GCC 4.5 compiler.

be loaded concurrently if concurrent requests are sent out to the host's main memory using the HTX bus. However our design does not currently support IDs attached to requests so that they can be identified and sent to their respective streams in the CCU when the data has been fetched. There is a field present in the HTX Bus packet format which does allow such a field to be set during a data request and it provides scope for future work. The way I currently handle parallel loading is by requesting less data for the first stream say about one-fourth of the maximum amount required. Such a request completes faster than a request for the complete amount of array data and consequently the data for the next stream can be requested immediately. Such a design requires keeping track of the address from where the last data fetch was done as well the size of each request. But the additional hardware is minimal and performance was slightly improved.

In addition to the above some algorithms such as the Secure Hashing Algorithm 1(SHA1) algorithm, used for generating fingerprints of messages in security applications, required a significant rewrite to successfully generate hardware using ROCCC. This is primarily due to the fact that ROCCC is optimized to generate hardware for loop nests and the entire logic of the algorithm should be present inside the loop. The transformation required to adapt the C code to a form accepted by ROCCC is shown in Figure 3.14 along with the ports generated for the top-level entity.

Yet another case is when the algorithm cannot be modified to fit the entire logic into a single large for-loop or nest of loops. Such a case exists for the x264 video encoder, whose most computer intensive function has a series of for-loops. These loops are converted to separate ROCCC systems as shown in Figure 3.15. Since the algorithm entails that the output of one loop goes to the next one thus these systems are managed by separate FSMs and their outputs and inputs are connected using FIFOs. I hereafter refer to such systems as *Chained Systems*. Chained Systems increase the scope of usage of the hardware generation process of the HTX Platform as the logic of many algorithms cannot be expressed within a single loop.

Moreover the concept of Chained Systems was extended to support parallelism. If multiple systems are required to express an algorithm in hardware then *wrappergen* generates an FSM for each of these systems. These FSMs supply the data in the correct timing sequence to the wrapped ROCCC system. In the case where a loop produces output that another loop can immediately consume, the FSMs for both systems can be started at the same time. Data can be passed from the producer to the consumer system using FIFOs and synchronization is acheived by separate signals or through the FIFO becoming full or empty which halts the producer FSM or consumer FSM respectively. This method has the added advantage that data required by a consumer FSM need to be read again from the host's main memory after the producer has written it there, thus avoiding read/write delays. It does however require FIFO resources on the FPGA but the FIFO sizes can be kept quite small depending on the amount of data passed between the FSMs.

3.4.3 Optimization of the CCU generation process

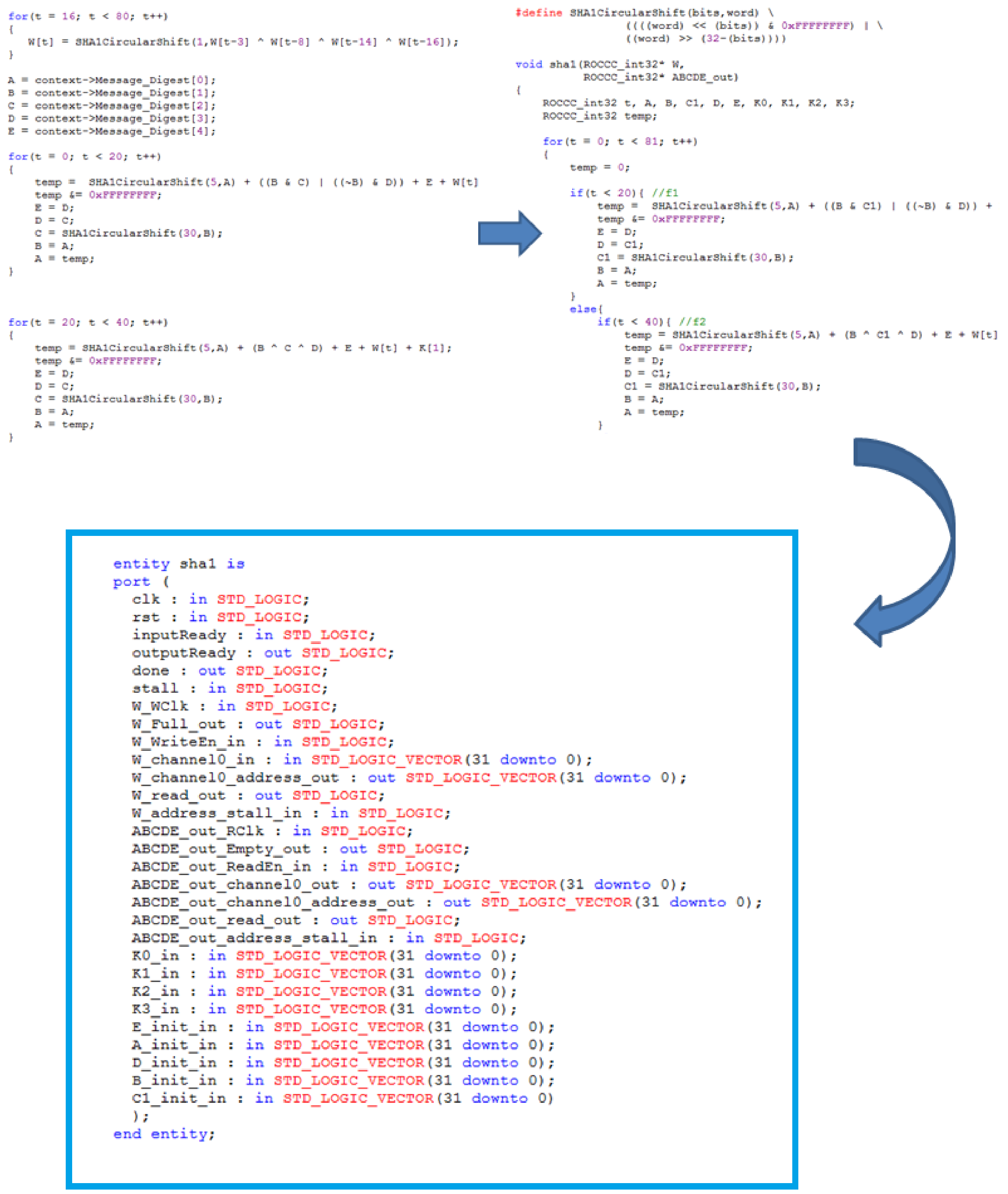
It is worth noting another feature which I added to the system. The CCU creation process is fully automated with the user only having to provide the C code and then press a button. Complete synthesis, mapping and place route is then done automatically with generation of supporting test benches and test programs for verifying the created CCUs.

This section discussed the optimizations done to extract maximum benefits from the work done by me on the HTX platform. In the next chapter I present the setup for evaluating this performance. I finally come to the conclusion of this chapter.

3.5 Conclusion

In this chapter I talked about the modifications done to enable partial reconfiguration on the HTX Platform. I began with a discussion about the interfaces provided by Xilinx to configure their FPGAs and went into particular detail about the Internal Configuration Access Port. Subsequently I discussed the modifications that were applied to the Xilinx bitstream format before it could be used to configure the FPGA. I also discussed the development of the ICAP Controller and how the size of Read FIFOs affected the design. Then I focused on floor planning of components for our platform and the final plan I settled on for getting optimum timing results for our design. This was followed with a discussion about compiler support for partial reconfiguration and driver modifications done for non-blocking execution of the calling process. This enabled us to add support for prefetching in the design. I finally talked about hardware generation using the ROCCC compiler and wrapping ROCCC systems and modules through the *wrappergen* program. In the last section I concluded with a discussion of the optimizations done to extract the best possible performance from the system.

I now proceed to the analysis of my work in the next chapter. After explaining the experimental setup for this analysis I go into the results obtained and it's interpretation.

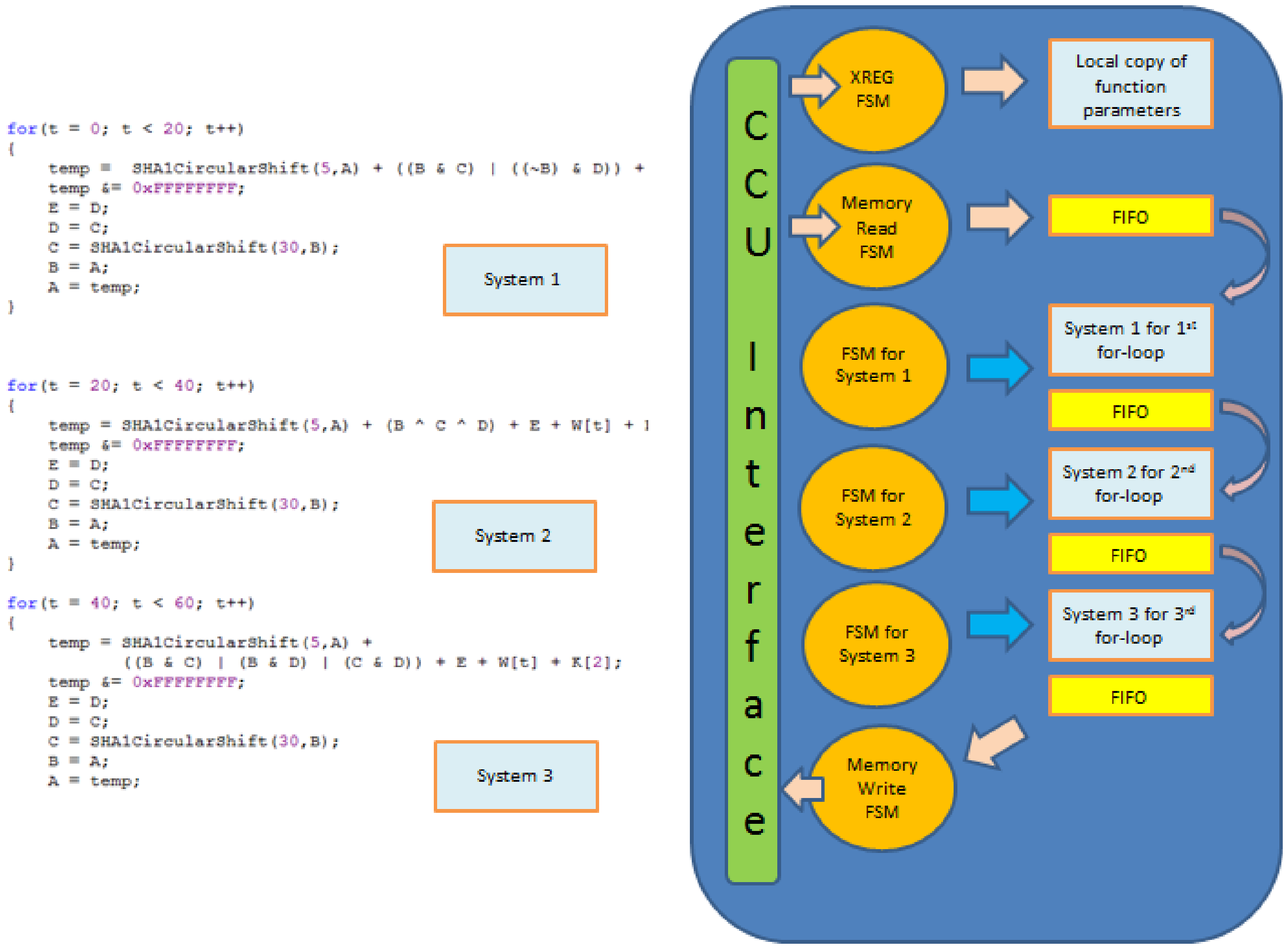


```
for(t = 16; t < 80; t++)
{
   W[t] = SHA1CircularShift(1,W[t-3] ^ W[t-8] ^ W[t-14] ^ W[t-16]);
}

A = context->Message_Digest[0];
B = context->Message_Digest[1];
C = context->Message_Digest[2];
D = context->Message_Digest[3];
E = context->Message_Digest[4];

for(t = 0; t < 20; t++)
{
    temp =  SHA1CircularShift(5,A) + ((B & C) | ((~B) & D)) + E + W[t]
    temp &= 0xFFFFFFFF;
    E = D;
    D = C;
    C = SHA1CircularShift(30,B);
    B = A;
    A = temp;
}


for(t = 20; t < 40; t++)
{
    temp = SHA1CircularShift(5,A) + (B ^ C ^ D) + E + W[t] + K[1];
    temp &= 0xFFFFFFFF;
    E = D;
    D = C;
    C = SHA1CircularShift(30,B);
    B = A;
    A = temp;
}
```

```
#define SHA1CircularShift(bits,word) \
                (((((word) << (bits)) & 0xFFFFFFFF) | \
                ((word) >> (32-(bits))))

void sha1(ROCCC_int32* W,
          ROCCC_int32* ABCDE_out)
{
    ROCCC_int32 t, A, B, C1, D, E, K0, K1, K2, K3;
    ROCCC_int32 temp;

    for(t = 0; t < 81; t++)
    {
        temp = 0;

        if(t < 20){ //f1
            temp =  SHA1CircularShift(5,A) + ((B & C1) | ((~B) & D)) +
            temp &= 0xFFFFFFFF;
            E = D;
            D = C1;
            C1 = SHA1CircularShift(30,B);
            B = A;
            A = temp;
        }
        else{
            if(t < 40){ //f2
                temp = SHA1CircularShift(5,A) + (B ^ C1 ^ D) + E + W[t]
                temp &= 0xFFFFFFFF;
                E = D;
                D = C1;
                C1 = SHA1CircularShift(30,B);
                B = A;
                A = temp;
            }
```

```
entity sha1 is
port (
  clk : in STD_LOGIC;
  rst : in STD_LOGIC;
  inputReady : in STD_LOGIC;
  outputReady : out STD_LOGIC;
  done : out STD_LOGIC;
  stall : in STD_LOGIC;
  W_WClk : in STD_LOGIC;
  W_Full_out : out STD_LOGIC;
  W_WriteEn_in : in STD_LOGIC;
  W_channel0_in : in STD_LOGIC_VECTOR(31 downto 0);
  W_channel0_address_out : out STD_LOGIC_VECTOR(31 downto 0);
  W_read_out : out STD_LOGIC;
  W_address_stall_in : in STD_LOGIC;
  ABCDE_out_RClk : in STD_LOGIC;
  ABCDE_out_Empty_out : out STD_LOGIC;
  ABCDE_out_ReadEn_in : in STD_LOGIC;
  ABCDE_out_channel0_out : out STD_LOGIC_VECTOR(31 downto 0);
  ABCDE_out_channel0_address_out : out STD_LOGIC_VECTOR(31 downto 0);
  ABCDE_out_read_out : out STD_LOGIC;
  ABCDE_out_address_stall_in : in STD_LOGIC;
  K0_in : in STD_LOGIC_VECTOR(31 downto 0);
  K1_in : in STD_LOGIC_VECTOR(31 downto 0);
  K2_in : in STD_LOGIC_VECTOR(31 downto 0);
  K3_in : in STD_LOGIC_VECTOR(31 downto 0);
  E_init_in : in STD_LOGIC_VECTOR(31 downto 0);
  A_init_in : in STD_LOGIC_VECTOR(31 downto 0);
  D_init_in : in STD_LOGIC_VECTOR(31 downto 0);
  B_init_in : in STD_LOGIC_VECTOR(31 downto 0);
  C1_init_in : in STD_LOGIC_VECTOR(31 downto 0)
  );
end entity;
```

Figure 3.14: Transformations done to the SHA1 transform block function to allow automatic hardware generation by ROCCC.



```
for(t = 0; t < 20; t++)
{
    temp =  SHA1CircularShift(5,A) + ((B & C) | ((~B) & D)) +
    temp &= 0xFFFFFFFF;
    E = D;
    D = C;
    C = SHA1CircularShift(30,B);
    B = A;
    A = temp;
}


for(t = 20; t < 40; t++)
{
    temp = SHA1CircularShift(5,A) + (B ^ C ^ D) + E + W[t] + I
    temp &= 0xFFFFFFFF;
    E = D;
    D = C;
    C = SHA1CircularShift(30,B);
    B = A;
    A = temp;
}

for(t = 40; t < 60; t++)
{
    temp = SHA1CircularShift(5,A) +
           ((B & C) | (B & D) | (C & D)) + E + W[t] + K[2];
    temp &= 0xFFFFFFFF;
    E = D;
    D = C;
    C = SHA1CircularShift(30,B);
    B = A;
    A = temp;
```

Figure 3.15: Conversion of the x264 kernel into hardware using Chained Systems.

4 Evaluation

In this chapter I describe the application used to evaluate my work on the HTX platform. The application considered is for secure audio processing. Secure audio is used for secure teleconferencing and to secure telecommunications, e.g. in defense, business, financial, public administration, homeland security etc. The application first processes the input of 16 audio channels using the beamforming technique and subsequently the output of the audio processing is encrypted using AES encryption. In the experiments, both beamforming audio processing and AES encryption functions are dynamically installed and accelerated in reconfigurable hardware.

After explaining the application in sufficient detail in Section 4.1, I present the results of accelerating this application on the HTX Platform. Also, I interpret these results and point out where dynamic partial reconfiguration is interesting. Then in Section 4.3 I describe how the wrappers generated by *wrappergen* were verified. Finally I conclude this chapter in Section 4.4.

Now I proceed to an explanation of the kernel used for beamforming Audio processing and the AES kernel which are used in the application.

4.1 The AES and Audio cores

In this section I describe the CCUs used to evaluate my work for the thesis. The Audio CCU is used for beamforming audio processing, and uses the design proposed in [24]. Beamforming is a signal processing technique that improves the signal strength received from a specific location. It has already been used for many years in areas of SOund Navigation And Ranging (SONAR), RAdio Detection And Ranging (RADAR) etc. The design used for this specific case is of a non-adaptive beamformer, which can be used in small hand-held devices as well as in large 3-D audio systems. Further details can be found in [25].

The AES security kernel, used to encrypt the audio processing output, is an Advanced Encryption Standard (AES) cryptography application, the design of which was proposed in [17]. The AES is a streaming application that encrypts data using the Rijndael algorithm. The hardware accelerated function uses a 256-bits key, and supports both ECB and CBC cyphering modes. The application encrypts data blocks of 128 bits at a time reading them sequentially. The processing of each data block takes 14 clock cycles, plus 2 additional cycles for read and write.

Now I move on to the performance of these kernels when combined into a real world application and the interpretation of these results.

4.2 Results using DPR

In this Section, I describe the experimental results from utilizing the HTX Platform for secure audio processing. I also present the operating frequency and area overhead of the FPGA hardware modules. Subsequently, I evaluate the latency of dynamically reconfiguring hardware functions of various bitstream sizes. Finally, I use the secure audio processing application, described earlier, to evaluate the performance of the HTX system versus software, using static and dynamic hardware acceleration.

	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	9,274	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: Usage of various types of components in the FPGA.

Table 4.1 presents the area requirements of the FPGA modules. The HTX interface occupies about 5,000 slices and 36 BRAMs. That is due to multiple queues needed for communication with the HyperTransport bus and the tables required to keep track of ongoing transactions. The wrapper needs more than 2,500 slices and 69 BRAMs; it is worth noting that address translation and the DMA manager occupies the largest part of the wrapper, mostly due to it's large queues and memory blocks. The maximum size of the partially reconfigurable 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, while the AES accelerator covers 5,000 slices. Although the rest of the design can operate at 200MHz, the frequency of the entire FPGA device is limited by the AES accelerator to 100 MHz.

Next the latency of the dynamic partial reconfiguration for the HTX platform is measured. Figure 4.1 shows the variation of reconfiguration latency with the bitstream size. As expected a larger bitstream causes a higher latency in reconfiguring the FPGA. Also a drop in the reconfiguration latency at about 600kb before again beginning to rise at about 1.1mb can be attributed to variations in the speeds of the HTX bus. At about 600k HTX packets have a relatively higher ratio of data versus packet header information. This would have caused a slight variation in the speed at which the packets were delivered to the FPGA. Once the reconfiguration process has been initiated by the ICAP controller, the bitstream needs to be fed to the ICAP without any interruptions. In order to ensure this, a large part of the bitstream (128KBytes) needs to be read and stored in the Read DMA Queue before starting the reconfiguration. This increases the required size of the DMA queue which leads to an increase in the reconfiguration latency compared to the theoretical minimum. Bitstreams of a few tens of Kbytes require less

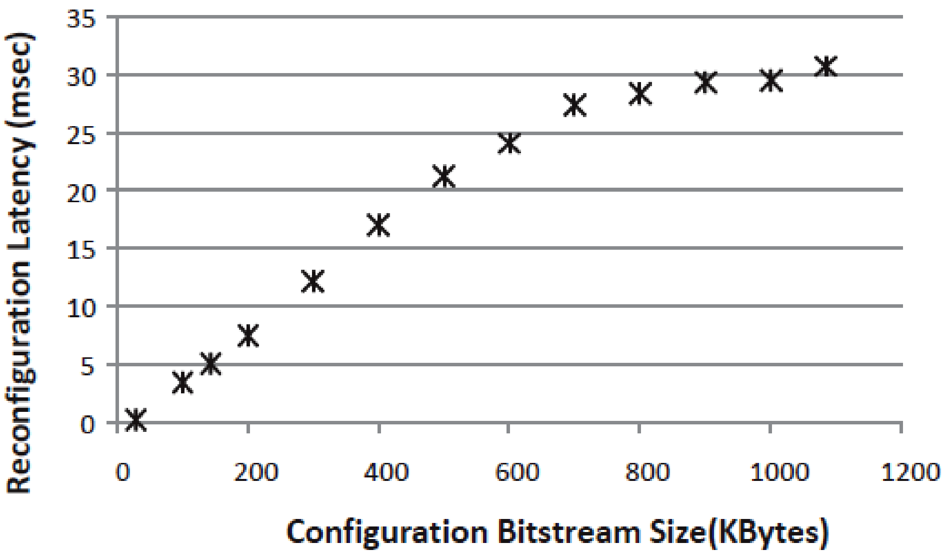


Figure 4.1: Bitstream configuration latencies.

than a millisecond to be configured, a 256 KByte bitstream roughly needs about 10 milliseconds, while bitstreams larger than a MByte have a reconfiguration latency of more than 30 milliseconds.

Large latencies offered by the FPGA pose a major impediment to getting impressive accelerations. To hide such latencies prefetching techniques were used as discussed below.

The performance of the HTX system was evaluated using the secure audio application described earlier. The application consists of two kernels - the Audio processing and AES kernel, which do not fit together in the largest possible partially reconfigurable region of our design. This is largely due to the size of the Audio Kernel which requires significant amounts of BRAM resources amounting to nearly half of the total present on the FPGA board. Thus the two kernels are time multiplexed into the FPGA through partial reconfiguration with the Audio CCU going first. I measure and compare the following alternative implementations of the application under study:

1) a purely software implementation, having both the Audio and the AES kernels running on the AMD host processor; 2) both the Audio and the AES kernels accelerated in the reconfigurable hardware, the first one statically preconfigured and the second one dynamically reconfigured on the fly; 3) the software Audio implementation and the hardware AES version, dynamically configured with the prefetching option used in the compiler; 4) the software Audio implementation and the hardware AES version, dynamically configured without prefetching the bitstream; 5) the Audio kernel accelerated in hardware (statically preconfigured), and the AES kernel implemented in software.

Figure 4.2 shows the execution time of the five implementation alternatives for different input sizes of audio input, while Figure 4.3 shows the speedup of the hardware approaches compared to the software implementation of the application. The latter graph shows minor oscillations due to variations in the reconfiguration time as reconfig-

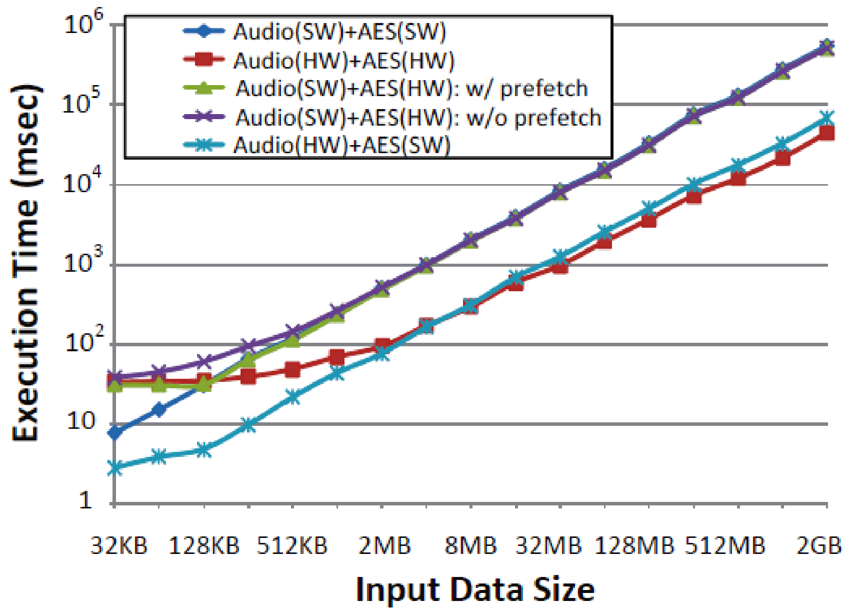


Figure 4.2: Execution Time versus bitstream size.

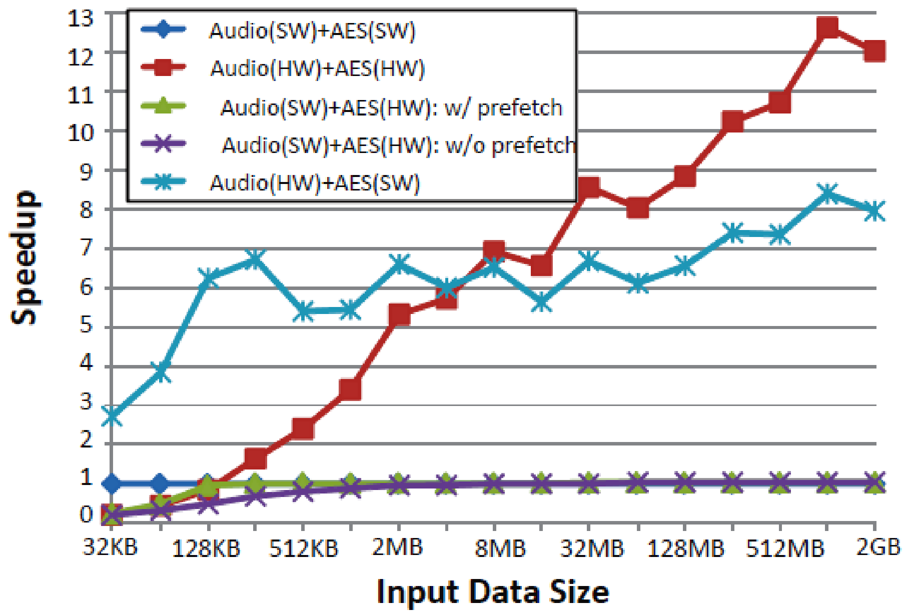


Figure 4.3: Speedup versus bitstream size.

uration is initiated through the process running in the host CPU. The operating system in the host CPU may schedule this process at random times leading to speedup which varies slightly from the expected linear increase. For small input sizes, the purely software implementation is better than the purely hardware one, due to the reconfiguration overhead of the AES kernel; in these cases the overall execution time is a few tens of milliseconds, which is comparable to the 30 msec reconfiguration latency of the AES. Even for such small input sizes, accelerating only the Audio kernel(implementation-5) is 3 to 7 times better than software. For input sizes larger than 128 KBytes the purely hardware solution is better than the purely software implementation, however it is still worse than executing only the audio kernel in hardware; this is again due to the extra latency introduced by the dynamic AES reconfiguration in the former case. For inputs larger than 8 MBytes the execution times are in the range of seconds, thus the reconfiguration latency becomes negligible compared to the total latency, as well as compared to the benefits of hardware acceleration. The purely hardware solution is then the fastest choice; it achieves an increasing speedup which, for 2 GBytes inputs, is over 12 times better than software. Accelerating only the Audio processing part in hardware achieves a speedup of up to 8 compared to software. Accelerating only the second (AES) kernel in hardware, while running Audio processing in software, has only a marginal improvement over purely software solutions which is not evident in the graphs.

This is primarily due to the fact that Audio processing is the most computationally intensive kernel, covering about 95% of the total execution, consequently, accelerating the AES in hardware gives only a 2-4% performance improvement. I evaluated two different implementations for accelerating only the second AES kernel in hardware. The first one is with prefetching of the AES configuration and the second without prefetching. For small input sizes the benefit of prefetching(implementation-3) is evident being 1.2-1.9 faster compared to implementation-4.

To summarize, it was found that dynamic reconfiguration is interesting in cases where the hardware-accelerated function has execution times larger than the reconfiguration latency and a significant speedup over software implementations. In the case of the HTX this was true for input sets larger than a MByte. The above is illustrated better in Figure 4.4, which in addition shows that the Audio is the most computationally intensive part. Yet another point to be observed is that the AES latency becomes significant (30-40% of the total execution time), when Audio is the only kernel implemented in hardware (implementation-5). Finally, the effect of prefetching is noticeable in Figure 4.4; the reconfiguration overhead is gradually hidden in implementation-3 as opposed to implementation-4, which does not prefetch the reconfiguration bitstream.

I now move on to the tests carried out to verify the wrapper generated for the ROCCC output, by the *wrappergen* program.

4.3 Testing ROCCC Code through an implementation of the AES algorithm

This section describes the experimental verification of the automatically generated CCUs. Three particularly demanding applications were chosen to test the wrapper generated by *wrappergen*. Their compute intensive functions were determined through profiling and is shown in Table 4.2. They were chosen on the basis of the varying approach required to convert them to hardware which in turn required extensions to the *wrappergen* program.

Algorithm	**Compute-Intensive Function**	**Changes required in the C code**
SHA1	Transform Message Block Function	The multiple for-loops were converted into a functionally equivalent single for-loop with appropriate conditional statements
X264 Encoder	Pixel Averaging Function	Each of the for-loops were converted into an unique ROCCC system and chained together in the CCU.
Fast Fourier Transform	FFT Butterfly operation	The criss-cross butterfly connections were coded as one stage in a module and this was called multiple times within a loop

Table 4.2: The algorithms chosen to test the output of *wrappergen*.

The CCU is generated completely without any user intervention. However there was in general a reduction in performance a compared to purely hand designed VHDL implementations. This can be attributed to the highly generic nature of the FSM based code produced by ROCCC which lacks the many optimizations possible in a pure handcrafted solution. Automatic CCU generation does save time and effort on the part of the user however and therefore the boost in productivity compensates for the loss in performance.

This concludes the evaluation of the system. I conclude this chapter in the next section and then move on to the final chapter which sums up my work.

4.4 Concluding the Evaluation

In this chapter I presented the experimental setup for evaluating the performance of the HTX Platform after it was augmented with Partial Reconfiguration. A secure audio processing application was presented along with it's use in the real world. I then explained how such an application can be accelerated on the HTX Platform by time multiplexing the Audio and AES cores in the FPGA. The working of the Audio and AES cores were described with particular attention paid to the AES core which was a part of my work. The performance of the cores were then examined through the variation of their combined execution speed and speedup as a function of the input data size. I also presented the variation of reconfiguration latency as a function of the configuration bitstream size and highlighted the need to hide this latency through prefetching methods. I evaluated our approach using a secure audio application composed of beamforming audio processing and an AES encryption kernel. I measured the reconfiguration latency in the system and evaluated the performance gain of various implementation alternatives compared to

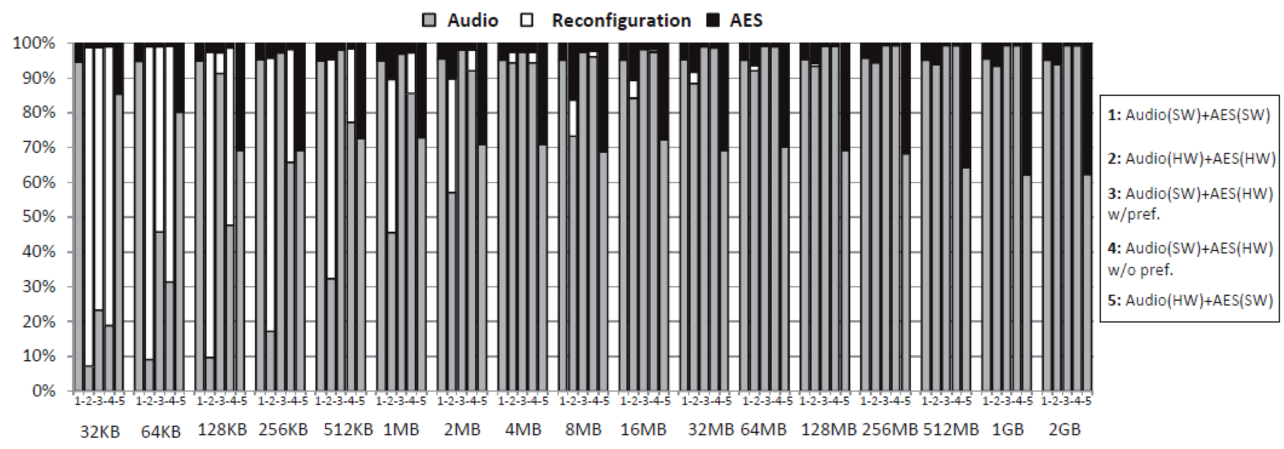


Figure 4.4: Breakup of Execution Time versus total time.

purely software solutions.

The enhancements done to the HTX platform resulted in the secure audio application running up to 12.6 times faster than software when both kernels are accelerated in the FPGA device using dynamic partial reconfiguration. The overhead of reconfiguration and the benefit of prefetching a configuration bitstream in the device was analyzed in detail with prefetching reducing the execution time by up to 1.9 times. Furthermore dynamic reconfiguration was proven to be beneficial in large input sets where the execution time of the accelerated function was significantly higher than the reconfiguration latency. Finally the efficiency of CCUs created through ROCCC and *wrappergen* was examined by automating the hardware generation of a software intensive function in the AES algorithm. In the next chapter I briefly recall the work presented in my thesis and it's real world applications. I wrap up this thesis with suggestions for future work and possible improvements to the existing system.

Conclusion 5

Hardware acceleration of general purpose applications through optimum use of reconfigurable devices has been the focus of this thesis. I described the HTX Platform which demonstrates a way to integrate reconfigurable computing with general purpose computing machines. The distinguishing factor in our platform is the multi-gigabit high speed link which provides low latency access to the host's memory from the FPGA device. Existing support for running programs on the HTX Platform at the time I began my work was described. I then elaborated on my enhancements to it as well as my attempts to ease the usage of such platforms through the integration of a C to VHDL compiler. Optimizations done to achieve the best possible outcomes of my work were discussed along with the setup used to evaluate the end results.

In this chapter I summarize the work presented in each chapter in Section 5.1 and provide directions for future work on the HTX Platform in Section 5.2.

5.1 Summary and contributions of this thesis

I began with an introduction to reconfigurable computing and the goals of this thesis in Chapter 1.

In Chapter 2 I discussed the HTX Platform and how it offers a new approach towards a general-purpose system with reconfigurable acceleration. I discussed how the FPGA was integrated and the mechanisms available to control it. This chapter also provided an introduction to the basic premise of dynamic partial reconfiguration and the adaptations required in a typically flat design to add partially reconfigurable regions. I discussed the hierarchical design methodology and some of the terminology associated with this flow. I then moved on to describe some existing commercial and research machines existing today that have similarities to the HTX Platform. This included the Convey Computer, the XtremeData DISC, SRC Computers and machines from DRC. Finally I touched upon the state of current research in C-to-VHDL compilers and a few existing solutions. These were the DelftWorkBench, Handel C, Catapult C, SRC's Map compiler and a few others. I concluded with a discussion of the ROCCC compiler and the reasons for integrating it into our platform.

Moving on, in Chapter 3 I talked about the modifications done to enable partial reconfiguration on the HTX Platform. I began with a discussion about the interfaces provided by Xilinx to configure their FPGAs and went into particular detail about the Internal Configuration Access Port. Subsequently I discussed modifications that applied to the Xilinx bitstream format before it could be used to configure the FPGA through the ICAP. I also discussed the development of the ICAP Controller and how the size of Read FIFOs affected the design. Then I focused on floor planning of components for our platform and the final plan I settled on for getting the optimum timing results for our design. This was followed with a discussion about compiler support for partial reconfiguration and driver modifications done for non blocking execution of the calling process. This enabled us to add support for prefetching in the design. I finally talked about hardware generation using the ROCCC compiler and wrapping ROCCC systems and modules through the *wrappergen* program. The chapter concluded with a discussion of the optimizations done to extract the best possible performance from our system.

Ultimately in Chapter 4 I described the secure audio processing application used to evaluate my work. I discussed the Audio and AES cores and the resources occupied by them. Then I moved on to a discussion about the execution time obtained for different input data sizes and the variation of speedup with the same sizes. I also compared five different prefetching techniques and the performance each of them delivered. Towards the end I delved on the function used to test the CCU wrapper generated by *wrappergen* around the ROCCC output modules. I concluded with a comparison of execution time for each kernel used to benchmark our design , as a percentage of the total time of execution of the secure audio processing application.

Furthermore this thesis added several features to the HTX Platform related to dynamic partial reconfiguration. It's most important contributions were the prefetching techniques enabled through modifications of the host side driver. Compiler enhancements allowed hardware descriptions of functions to be included in the executable binary code and then get dynamically installed on-demand during program execution. The successful integration of an open source and actively developing C to VHDL compiler lowers the barriers for usage of the platform for software programmers. It also increases the scope of application of the platform with hardware acceleration support for any general purpose application now easily possible without manually building the hardware. Its worth noting that at this point most of the suggestions in [26] have been realized for the HTX Platform.

I now finish this presentation with suggested directions for future research.

5.2 Proposed directions for future research

Reconfigurable devices offer considerable benefits with regard to performance versus price and performance versus power. The idea of a general purpose computing machine integrated with a large reconfigurable device leads to a very attractive picture of machines with "under the hood" acceleration. Adoption of such technologies will however be decided largely by the success in addressing latency and ease of use issues which accompany FPGAs. This thesis attempted to address some of these issues. The possible improvements to the current HTX platform involve adding support for parallel data loading to start CCU execution as quickly as possible, use of branch prediction techniques to reconfigure the FPGA before it's scheduled use, runtime profiling and decisions for hardware usage and further improvements to the C-to-VHDL compiler platform.

5.2.1 Parallel Data loads

As discussed in the section on Optimizations in the previous chapter, ROCCC systems can benefit from loading data streams in parallel. This can be supported in the HTX

Platform through the SeqID field in the HyperTransport request and response command packet. If a ROCCC system has multiple streams which require to be loaded before it can begin then the streams can be numbered. The data for each input stream can be requested in parallel with the SeqID set to the stream number. When the response is available, the SeqID can be read from the response packet and the data directed to the appropriate stream. This technique can be combined with interleaving of smaller chunks of data such that the CCU is never starved for data and consequently paused for multiple clock cycles.

5.2.2 Branch prediction techniques for runtime configuration decisions

Static branch prediction techniques that examine the frequency of branches taken and accordingly schedule instructions can be used to configure the FPGA at the correct time. This can be done at compile time. Using static brnach prediction the compiler could choose to reconfigure the FPGA beforehand if the likelihood of it's being used inside a branch is high. On the other hand it may choose not to configure it at all if the only call to the FPGA is from a branch with a low chance of being taken. It may instead configure the FPGA with a function that may be called further down the sequence of instructions thus providing more time for the reconfiguration to complete. These techniques can overlap considerable reconfiguration time with execution on the host CPU if implemented correctly. In a sense hiding reconfiguration latency then becomes similar to attempts to hide main memory access latency and a cache of bitstreams may well serve a similar purpose here.

5.2.3 Profiling for deciding hardware usage

As can be seen in the results presented in the previous chapter, both the Audio and the AES kernels have quite linear increase in their execution time versus the input size. 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 profiling information. In doing so, it can be predicted dynamically at runtime whether or not dynamic partial reconfiguration and hardware acceleration of a function would deliver speedup. The HTX platform supports such runtime decisions since both software and hardware implementation co-exist in the same binary code and the most suitable one can be chosen dynamically on-demand. Note that this is different from scheduling hardware reconfiguration as presented in the previous sub-section as this deals with whether or not to use the hardware at all.

5.2.4 Improvements to the C-to-VHDL compiler platform

The C-to-VHDL compiler can be improved with specific passes introduced at the Lo-CIRRF level for generating CCUs directly as ROCCC output. This eliminates the need for separately wrapping the generated modules later on through another program such as *wrappergen*. Also some optimizations specific to the HTX Platform can be enabled which will lead to timing improvements and higher speedups.

Bibliography

[1] Y. D. Yankova, G. Kuzmanov, K. Bertels, G. N. Gaydadjiev, Y. Lu, and S. Vassiliadis, "Dwarv: Delftworkbench automated reconfigurable vhdl generator," in *In Proceedings of the 17th International Conference on Field Programmable Logic and Applications (FPL07)*, pp. 697–701, August 2007.

[2] "SRC Computers," http://www.srccomp.com/.

[3] M. Gokhale, J. Stone, J. Arnold, and M. Kalinowski, "Stream-oriented fpga computing in the streams-c high level language," in *Field-Programmable Custom Computing Machines, 2000 IEEE Symposium on*, pp. 49 –56, 2000.

[4] "Virtex-4 FPGA Configuration User Guide," June 2009. Xilinx Inc.

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

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

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

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

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

[10] "Intel Quickpath.," http://www.intel.com/technology/quickpath/.

[11] "Hyper Transport Bus.," www.hypertransport.org.

[12] "DRC Computers," http://www.drccomputer.com/.

[13] J. Villarreal, A. Park, W. Najjar, and R. Halstead, "Designing modular hardware accelerators in c with roccc 2.0," in *Proceedings of the 2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, FCCM '10, (Washington, DC, USA), pp. 127–134, IEEE Computer Society, 2010.

[14] "Mentor Catapult C.," http://www.mentor.com/esl/catapult/overview.

[15] "Handel-C Overview," http://www.celoxica.com.

[16] "HTX Board," http://ra.ziti.uni-heidelberg.de/index.php?page=projects&id=htx.

[17] A. Brandon, I. Sourdis, and G. N. Gaydadjiev, "General purpose computing with reconfigurable acceleration," in *20th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 588–591, August 2010.

[18] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Panainte, "The Molen Polymorphic Processor," *IEEE Transactions on Computers*, vol. 53, pp. 1363–1375, 2004.

[19] "Hierarchical Design Methodology Guide," March 2011. Xilinx Inc.

[20] "The SUIF Compiler System.," http://suif.stanford.edu/.

[21] "The LLVM Compiler Infrastructure.," http://llvm.org/.

[22] "Jacquard Computing.," http://www.jacquardcomputing.com/.

[23] Z. Guo, B. Buyukkurt, and W. Najjar, "Input data reuse in compiling window operations onto reconfigurable hardware," in *Proc. ACM Symp. On Languages, Compilers and Tools for Embedded Systems (LCTES*, pp. 249–256, ACM Press, 2004.

[24] D. Theodoropoulos, *Custom Architecture for Immersive-Audio Applications.* PhD thesis, May 2011.

[25] V. Venkatasubramanian, "Hardware support for dynamic partial self-reconfiguration of the htx reconfigurable platform," Master's thesis, Delft University of Technology, September 2011.

[26] A. Brandon, "General purpose computing with reconfigurable acceleration," Master's thesis, Delft University of Technology, November 2010.

Curriculum Vitae

Abhijit Nandy was born in Jamshedpur, India on 24th March in 1984. He finished his schooling from Loyola School, Jamshedpur in 2002 with a GPA of 8.0/10.0. He then went on to pursue an interest in Computer Science at Utkal University, India graduating with a Bachelor's degree in 2006. His Bachelor's thesis was related to pattern recognition using the Particle Swarm Optimization technique.

After graduation he joined the Tata Group working for the Tata Consultancy Services from 2006 to 2009 mainly working with Teradata data warehouses and application development in Java. In 2009 he decided to pursue his interest in hardware acceleration using FPGAs at TU Delft and joined the Computer Engineering program in September 2009. He finished this M.Sc. thesis under Ioannis Sourdis and Georgi Gaydadjiev.

His research interests include hardware acceleration in FPGAs, networks on chip, embedded systems and physics simulation on the GPU. He is an active code contributor to the BRL-CAD open source solid modeling tool and the Orbiter space flight simulation software.