Rochester Institute of Technology
RIT Digital Institutional Repository

Theses

5-1-2009

Scalable framework for heterogeneous clustering of commodity FPGAs

Jeremy K. Espenshade

Follow this and additional works at: https://repository.rit.edu/theses

Recommended Citation
Espenshade, Jeremy K., "Scalable framework for heterogeneous clustering of commodity FPGAs" (2009). Thesis. Rochester Institute of Technology. Accessed from

This Thesis is brought to you for free and open access by the RIT Libraries. For more information, please contact repository@rit.edu.

Scalable Framework for Heterogeneous Clustering of Commodity FPGAs

by

Jeremy K. Espenshade

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science
in Computer Engineering

Supervised by

Assistant Professor Dr. Marcin Lukowiak
Department of Computer Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, New York
May 2009

Approved by:

Dr. Marcin Lukowiak, Assistant Professor
Department of Computer Engineering

Dr. Muhammad Shaaban, Associate Professor
Department of Computer Engineering

Dr. Gregor von Laszewski, Associate Professor
Department of Computer Science

Thesis Release Permission Form

Rochester Institute of Technology
Kate Gleason College of Engineering

Title:

Scalable Framework for Heterogeneous Clustering of Commodity FPGAs

I, Jeremy K. Espenshade, hereby grant permission to the Wallace Memorial Library to reproduce my thesis in whole or part.

Jeremy K. Espenshade

Date

Dedication

This thesis is dedicated to all the people in my life who, for better or worse, distracted me from actually working on it over the last year or so. To my family, for always being there. To Becky, for encouraging me throughout and giving me something to look forward to. To my friends in Rochester, for keeping day-to-day life fun and interesting. And specifically to Aric, for making sure my flash game procrastination was of the highest quality.

Acknowledgments

Special thanks to my adviser, Dr. Lukowiak, for his guidance, support, and flexibility. Also to the RIT Computer Engineering Department for its financial and material support. Thanks to RIT and the many professors I have had who encouraged me and prepared me for this work and whatever is next. I would also like to acknowledge the other students in the Hardware Development Lab who, at various times, provided helpful insights or suggestions. Thanks Andy, Aric, Dmitri, Ken and Ken.

Abstract

Scalable Framework for Heterogeneous Clustering of Commodity FPGAs

Jeremy K. Espenshade

Supervising Professor: Dr. Marcin Lukowiak

A combination of parallelism exploitation and application specific hardware is increasingly being used to address the computational requirements of a diverse and extensive set of application areas. These targeted applications have specific computational requirements that often are not able to be implemented optimally on general purpose processors and have the potential to experience substantial speedup on dedicated hardware. While general parallelism has been exploited at various levels for decades, the advent of heterogeneous cluster computing has allowed applications to be accelerated through the use of intelligently mapped computational tasks to well-suited hardware. This trend has continued with the use of dedicated ASIC and FPGA coprocessors to off-load particularly intensive computations. With the inclusion of embedded microprocessors into otherwise reconfigurable FPGA fabric, it has become feasible to construct a heterogeneous cluster composed of application specific hardware resources that can be programatically treated as fully functional and independent cluster nodes via a standard message passing interface.

The contribution of this thesis is the development of such a framework for organizing heterogeneous clusters of reconfigurable FPGA computing elements into clusters that enable development of complex systems delivering on the promise of parallel reconfigurable hardware. The framework includes a fully featured message passing interface implementation for seamless communication and synchronization among nodes running in an embedded Linux operating system environment while managing hardware accelerators through device driver abstractions and standard APIs. A set of application case studies deployed on

a test platform of Xilinx Virtex-4 and Virtex-5 FPGAs demonstrates functionality, elucidates performance characteristics, and promotes future research and development efforts.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

In the domain of high-performance computing, several architectural avenues are being explored in the search for maximum performance, optimal cost/benefit ratios, and flexibility in approaching computationally intensive tasks. Although most commercial super computers continue to be built with many homogeneous uniprocessors or chip-multiprocessors [16] [3], a recognizable trend has been toward inclusion of dedicated hardware to assist in computations that are especially intensive or for which a typical general-purpose processor (GPP) is ill-suited. This acceleration often takes the form of directly connected hardware co-processors using application-specific integrated circuits (ASICs), reconfigurable fabric (FPGAs), or the class of streaming architectures including the STI Cell Broadband Engine and programmable graphics processing units (GPGPUs). Examples of each include the D.E. Shaw Research Anton molecular dynamics simulation supercomputer [37], the Cray XT5h [38] and SRC-7 [8] reconfigurable supercomputers, and the IBM BlueGene/P hybrid supercomputer [7] and Nvidia Tesla based computing solutions [10]. The motivation for all of these architectures is the set of applications that require computational acceleration and contain program segments that are not ideally suited for execution on a general purpose processor (GPP). Some examples of such application areas are cryptanalysis, molecular dynamics simulations, bioinformatics, and high data-throughput image and video processing [17].

While positive results have been garnered in these and many other application areas

using the coprocessor model, a recent area of research interest has been placing increased responsibility with the acceleration hardware rather than having fully capable GPPs dedicated to managing coprocessors. This direction is especially focused on reconfigurable computing elements, as they allow flexibility both in interaction and computation. High performance computing (HPC) research efforts using reconfigurable elements exclusively have been undergone at several universities [30][36] and at the Airforce Research Laboratory Rome [31]. These efforts have been largely successful at demonstrating the potential for reconfigurable computing in a massively parallel environment [17], but, as with all the previously mentioned efforts, keep their systems homogeneous.

An avenue that has yet to be explored to any great extent is in the use of commodity off-the-shelf (COTS) FPGAs as fully functioning and participating nodes in a heterogeneous, Beowulf-style [32] computing cluster for HPC use. With the advent of high performance, high capacity FPGAs with cost similar to that of GPP cluster nodes, such an inclusion is both feasible and promising. Particularly, those FPGAs which include one or more hard-wired PowerPC processors embedded in the reconfigurable fabric provide a familiar and well established processing environment that is relatively high performance when compared to softcore processors like the MicroBlaze [11][1]. With a PowerPC managing on-board peripherals and application stacks within an embedded Linux environment, the need to couple the FPGA with a general purpose host is removed and the co-processor paradigm can be replaced with hardware accelerators acting as autonomous cluster nodes.

To enable FPGA inclusion in such a cluster, one must first consider the communication method used to coordinate processes and pass data between nodes. The commonly accepted standard API for communication on distributed memory systems ranging in size and complexity from small COTS clusters to the BlueGene/P supercomputer is the Message Passing Interface (MPI) [19]. The MPI standard supports a send/receive paradigm for interprocess communication with many additional features for collective communication, process organization, etc [4]. As such, retaining this common interface has the benefits

of allowing reuse of existing code structure and function, building upon a well-established communication infrastructure, and, most importantly, presenting a familiar and understood API for development targeted toward an FPGA cluster.

1.2 Contribution

This thesis contributes a scalable communication framework enabling on and off-chip communication among computing elements on a single FPGA and among multiple FPGAs organized in a computing cluster. A software environment consisting of an embedded Linux operating system and OpenMPI application stack along with custom hardware accelerators abstracted through a device driver allows these hardware accelerators to be treated as fully functioning nodes in an MPI-2 compatible cluster. Furthermore, targeted application case studies demonstrate performance properties of the system and guide future work.

This effort allows further development in the field of high-performance reconfigurable computing by enabling commodity FPGAs to be utilized in a standard cluster environment with the familiar MPI-based parallel programming model. By implementing simple hardware/software communication abstractions, required co-design effort is reduced and distributed heterogeneous system design is eased. With this capability, avenues of inquiry are opened in the areas of dynamic task scheduling, network topology investigation, performance modeling, dynamic reconfiguration evaluation, and flexible application development. The developed hardware/software framework enables application developers to integrate reconfigurable devices in the same way that the original development of MPI enabled the use of general purpose COTS components in cluster computing.

1.3 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 lays the conceptual groundwork with overviews of cluster computing and FPGA technology. Specifically, message passing based parallel programming and Xilinx hybrid FPGA architecture are explained in

detail. Chapter 3 then discusses related work in the areas of Linux and MPI on FPGAs and hardware/software interfacing techniques. Chapter 4 proceeds with an overview of the developed framework consisting of a software environment, hardware/software interface layer, and implied hardware design methodology. The programming model is then discussed to demonstrate how these layers interact. Chapter 5 introduces an example hardware platform of Xilinx Virtex-4 and 5 FPGAs configured with the described framework. Application case studies are presented to demonstrate functionality, elucidate performance characteristics, and guide future development. Chapter 6 concludes this thesis with directions for future research and development.

Chapter 2

Essential Background

To provide conceptual backing and motivation for the following chapters of this thesis, an overview of the underlying technologies is presented here.

2.1 Cluster Computing

2.1.1 Motivation

Historically, computer programs were all written as sequences of instructions to be executed in program order by a single processor. It quickly became apparent that some of these instructions could be executed concurrently as the inputs and outputs were independent [44]. For example, a program that sums two arrays into a third array presents the opportunity to perform the sum at each array index in parallel. This observation gave rise to more complex computer architectures able to execute multiple instructions at a time. In the high performance computing (HPC) market, special purpose computing engines exploiting data parallelism were constructed and marketed as early supercomputers by companies like Cray Research and IBM. These computing engines, capable of performing many similar instructions on different data in parallel, were known as Vector Computers.

Demand for increasing capabilities for complex large-scale data-intensive applications like computational fluid dynamics and physical simulations motivated these “big iron” supercomputers to evolve during the 1970’s and 80’s into parallel processors coordinating multiple vector processing units. In this way, larger problems with higher degrees of parallelism could be partitioned across independent processing units and the results could be

aggregated and synchronized at the system level.

At the same time, consumer microprocessor development largely followed a multiple-issue, pipelined methodology with increasing numbers of functional units able to perform various control and scalar arithmetic operations. Towards the late 1980's commercial software for sharing resources and distributing programs across multiple microprocessors started to arrive. The creation of Parallel Virtual Machine (PVM), an open source tool with libraries for message passing and resource management, allowed heterogeneous processors connected over Ethernet to communicate and coordinate execution. Large groups of microprocessors coordinated in this fashion quickly overtook traditional supercomputers and by 1994, the Beowulf project [5] dedicated to the construction of supercomputers composed of commodity-off-the-shelf (COTS) processors connected over commodity networks was founded. Within two years previously unrealized gigaflop-level performance was demonstrated by NASA and the US Department of Energy, and subsequent success has led such commodity clusters to be classified as "Beowulf class cluster computers" [32].

Driving this technological progression towards clusters providing ever-increasing computational capabilities has been the many applications that either become more useful through increased complexity or become computable in an acceptable length of time. An example of the former is computational fluid dynamics (CFD), which was one of the earliest motivating software application areas. Early hardware limitations required generalized models and approximate calculations, but as the computational capability increased, improved models were able to be simulated. While CFD started with simulations of single air-foils, it has now expanded to large-scale weather models able to calculate and model wind and atmospheric dispersion patterns across millions of computational volumes.

The later category of applications, those for which execution time rather than algorithmic complexity is the prime concern, can be divided into two categories: applications that experience increased usability through longer simulations or faster results and problems

that were simply not computable in a reasonable time without additional resources. Molecular dynamics provides a good example of the former category as increases in computing power allow each time step to be computed more quickly and thereby allow more time steps to be computed in a given simulation time. A good example of the later category is applied cryptanalysis, where cryptographic standards with security based on infeasible computation times required for key identification are made insecure by increased computational power. Infamously, DES was cracked in 1998 through a combined effort of the Electronic Frontier Foundation and Distributed Computing Technologies Inc. that used distributed computing software running in the background of thousands of desktop PCs and a cluster of custom ASICs to crack DES in less than 23 hours.

The reason that clusters of independent computing nodes are able to solve such complex problems is that all of these problems can be broken down into smaller, simpler problems executable on individual nodes concurrently. The degree to which a problem can be broken down in this way is known as the degree of software parallelism. This parallelism can take several forms. At the highest level, task parallelism can be exhibited by software for which multiple threads of execution are possible. These independent "processes" are then executed concurrently if there is enough hardware available to handle all of them. If the degree of software parallelism is different than the degree of hardware parallelism, the lower of the two is the deciding factor for how many processes are executed in parallel.

Within each task, there may then exist data parallelism where the same operations are performed on independent data, instruction-level parallelism where non-dependent instructions can be executed concurrently, or bit-level parallelism where computation on wide bit-width operands occurs within a single operation. Data parallelism promotes vector-type architectures that can perform many similar operations in parallel and while dedicated vector computers have been abandoned, modern processors nearly always contain vector processing units capable of performing a small number of the same operation in parallel. Similarly, instruction-level parallelism is exploited extensively by modern superscalar

and VLIW architectures. Bit-level parallelism is often not even considered, as the parallel computation is performed at the VLSI level with 32 or 64 bit arithmetic and logic units, but processor performance would be abysmal without this level parallelism exploitation.

2.1.2 MPI

Returning to the high-level task parallelism that allows a single application to be organized into multiple processes, a method for interprocess communication and synchronization is required. One such method is to exchange messages containing data and control information between remote processes. By communicating data via point-to-point and collective operations, each process can have the data it needs to continue working and return results in a flexible way that supports a variety of program organizations. Furthermore, small messages implementing interprocess hand shaking can be used to provide synchronization.

This message passing paradigm was adopted by the early PVM software suite and as use of this package became widespread, the need for a standardized parallel message passing API became apparent. To provide this standard, a number of industrial and academic institutions came together to propose the Message Passing Interface (MPI) standard. This standard provided a set of language independent APIs implementing point-to-point and collective communication and global synchronization across a virtual topology of processes. The standard has since been expanded to include APIs for parallel I/O, dynamic process management, and one-sided communication via the MPI-2 standard [4].

To accomplish the range of capabilities that MPI offers, several base concepts were introduced. The virtual topology is organized as a *Communicator*. Communicators are simply groups of processes, each with an assigned process ID number, or *rank*, ranging from '0' to one less than the number of processes, or *size*. A process can determine its rank with *MPI_Comm_rank* and the communicator size with *MPI_Comm_size*. The global communicator is *MPI_COMM_WORLD*, and sub-groups are created by splitting this global communicator if necessary. Point-to-point communication is then accomplished through *sends* via *MPI_Send* and *receives* via *MPI_Recv* between two members of a communicator.

Often, some data needs to be broadcast to all processes in a communicator or synchronization needs to occur globally. While this functionality could be accomplished with individual sends and receives, the process would be cumbersome. As such, collective operations are provided that implement scatter/gather, broadcast/reduce, and barrier synchronization. Examples of these APIs are *MPI_Bcast* and *MPI_Barrier*.

As an open standard, MPI defines the API and behaviors that should be provided by any implementation, but does not implement any of the defined functions itself. Since initial standardization, many implementations of MPI have existed and each provided various degrees of standard compliance and methods of implementing the defined behaviors. Recently several of these implementations merged into OpenMPI, which is one of the two most commonly used implementations along with MPICH.

2.2 FPGA Technology

Field-Programmable Gate Arrays (FPGAs) are a class of special-purpose digital devices that allow hardware connections and logic behavior to be configured by a developer. They are constructed as a network of configurable logic blocks connected on a reconfigurable interconnection mesh. Hardwired components are often included to provide common functionality. For example, Ethernet Media Access Controllers (MAC) or embedded microprocessors are common, as equivalent "soft" cores implemented in reconfigurable fabric often have high area requirements and limited performance. Two companies, Altera and Xilinx, provide over 80% of the FPGA designs with Xilinx controlling over 50% of the market alone. Each company provides a range of solutions with varying numbers of logic blocks and embedded components.

This thesis focuses on Xilinx FPGAs, and specifically those FPGAs which include an embedded PowerPC processor. Xilinx categorizes these "hybrid" FPGAs with hardwired processors and reconfigurable fabric into the Virtex-II Pro, Virtex-4 FX and Virtex-5 FXT series. The most recent two generations are very similar in feature sets, but differ in some

of the internal component architectures. Introduced in 2006, the Virtex-5 contains configurable logic blocks (CLBs) that are constructed from two "slices," each containing two 6-input, single-output look-up-tables (LUTs), two flip-flops, and some miscellaneous arithmetic and control logic. Virtex-4 FPGAs, introduced in 2004, use CLBs that are instead constructed from four "slices" that are similarly organized, but populated with two 4-input LUTs instead of the improved 6-input LUTs. Both generations include a number of DSP48 arithmetic slices that are specially configured to perform complex arithmetic instructions. Each DSP slice contains a 18x18 (Virtex-4) or 24x18 (Virtex-5) bit multiplier and multiple slices can be connected to perform operations on operands of larger bit-widths. Both FPGA generations also contain block RAM modules distributed throughout the chip. Available resources for Virtex-4 FX and Virtex-5 FXT FPGAs are shown in Figures 2.1 and 2.2. While the Virtex-6 generation of FPGAs has recently been announced by Xilinx, the Virtex-5 series represent the current state of the art in FPGA technology.

The embedded processors included in Xilinx hybrid FPGAs are from the PowerPC 4xx series, 405 for Virtex-4 and 440 for Virtex-5. These 32-bit architectures provide simple but competent general purpose computing environments. The PowerPC 405 includes a single-issue 5-stage pipeline with performance enhancing features like static branch prediction, instruction and data caches, and integer multiplication and division arithmetic units [2]. The PowerPC 440 is an improved version of this core, with a dual-issue, 7-stage superscalar execution engine and twice the available cache. Neither core supports floating point operations natively [1].

2.2.1 Base System Builder

Xilinx supports hardware/software development for hybrid FPGAs through their Embedded Development Kit (EDK). The project creation wizard is known as the "Base System Builder" (BSB) and it allows for the creation of a PowerPC or Microblaze centric design with preconfigured peripherals. Custom system building is a tedious process and the BSB reduces development time considerably. The EDK includes a library of Xilinx-provided IP

		Virtex-4 FX FPGA Platform Optimized for Embedded Processing & Serial Connectivity (1.2 Volt)					
	Part Number	XC4VFX12	XC4VFX20	XC4VFX40	XC4VFX60	XC4VFX100	XC4VFX140
	Easypath™ Cost Reduction Solutions [1]	—	—	XCE4VFX40	XCE4VFX60	XCE4VFX100	XCE4VFX140
Logic Resources	Slices [2]	5,472	8,544	18,624	25,280	42,176	63,168
	Logic Cells	12,312	19,224	41,904	56,880	94,896	142,128
	CLB Flip-Flops	10,944	17,088	37,248	50,560	84,352	126,336
Memory Resources	Maximum Distributed RAM (Kbits)	86	134	291	395	659	987
	Block RAM/FIFO w/ECC (18 Kbits each)	36	68	144	232	376	552
	Total Block RAM (Kbits)	648	1,224	2,592	4,176	6,768	9,936
Clock Resources	Digital Clock Managers (DCM)	4	4	8	12	12	20
	Phase-matched Clock Dividers (PMCD)	0	0	4	8	8	8
I/O Resources [3]	Maximum Single-Ended I/Os	320	320	448	576	768	896
	Maximum Differential I/O Pairs	160	160	224	228	384	448
	I/O Standards						
Embedded Hard IP Resources	DSP48 Slices	32	32	48	128	160	192
	PowerPC™ Processor Blocks	1	1	2	2	2	2
	10/100/1000 Ethernet MAC Blocks	2	2	4	4	4	4
	RocketIO™ Serial Transceivers	0	8	12	16	20	24
Speed Grades	Commercial	-10, -11, -12	-10, -11, -12	-10, -11, -12	-10, -11, -12	-10, -11, -12	-10, -11
	Industrial	-10, -11	-10, -11	-10, -11	-10, -11	-10, -11	-10
Configuration	Configuration Memory (Mbits)	4.8	7.2	13.6	21.0	33.0	47.9

Figure 2.1: Virtex-4 FX Resource Table [13]

cores and a selection of these are available for inclusion through the BSB wizard. The IP cores and configuration parameters included in the BSB designs discussed in this thesis are shown in Table 2.1.

Component	Parameters
Processor Type	PowerPC 440 (Virtex-5) PowerPC 405 (Virtex-4)
Processor Clock Frequency	400 MHz (Virtex-5) 300 MHz (Virtex-4)
Processor Cache	64 KB (32KB Data + 32KB Inst) (Virtex-5) 32 KB (16KB Data + 16KB Inst) (Virtex-4)
PLB Clock Frequency	100 MHz
UART Controller	XPS UART16550
Hard Ethernet MAC	Scatter-Gather DMA
DDR2 RAM	PowerPC Memory Controller
Compact Flash	XPS SysAce

Table 2.1: BSB Common Configuration Options

		Virtex-5 FXT FPGA Platform Optimized for Embedded Processing with High-Speed Serial Connectivity (1.0 Volt)				
	Part Number	XC5VFX30T	XC5VFX70T	XC5VFX100T	XC5VFX130T	XC5VFX200T
	EasyPath™ Cost Reduction Solutions (1)	—	XCE5VFXT70T	XCE5VFXT100T	XCE5VFXT130T	XCE5VFXT200T
Logic Resources	Slices (2)	5,120	11,200	16,000	20,480	30,720
	Logic Cells (3)	32,768	71,680	102,400	131,072	196,608
	CLB Flip-Flops	20,480	44,800	64,000	81,920	122,880
Memory Resources	Maximum Distributed RAM (Kbits)	380	820	1,240	1,580	2,280
	Block RAM/FIFO w/ECC (36Kbits each)	68	148	228	298	456
	Total Block RAM (Kbits)	2,448	5,328	8,208	10,728	16,416
Clock Resources	Digital Clock Managers (DCM)	4	12	12	12	12
	Phase Locked Loop (PLL)/PMCD	2	6	6	6	6
I/O Resources (4)	Maximum Single-Ended Pins	360	640	680	840	960
	Maximum Differential I/O Pairs	180	320	340	420	480
	I/O Standards					
Embedded (5) Hard IP Resources	DSP48E Slices	64	128	256	320	384
	PowerPC® 440 Processor Blocks	1	1	2	2	2
	PCI Express Endpoint Blocks	1	3	3	3	4
	10/100/1000 Ethernet MAC Blocks	4	4	4	6	8
	RocketIO™ GTP Low-Power Transceivers	—	—	—	—	—
	RocketIO™ GTX High-Speed Transceivers	8	16	16	20	24
Speed Grades	Commercial	-1, -2, -3	-1, -2, -3	-1, -2, -3	-1, -2, -3	-1, -2
	Industrial	-1, -2	-1, -2	-1, -2	-1, -2	-1
Configuration	Configuration Memory (Mbits)	13.6	27.1	39.4	49.3	70.9

Figure 2.2: Virtex-5 FXT Resource Table [14]

The central bus is an IBM Core Connect bus called the Processor Local Bus (PLB) [20]. Each peripheral besides the DDR2 memory controller connects to this 32-bit bus as a slave device and the PowerPC communicates with them as the bus master. The hard Ethernet MAC provides either 10/100T or 10/100/1000T connection speeds to external networks based on the interface method. The Media Independent Interface (MII) provides 10/100T speeds and enjoys reduced pin requirements while the Gigabit Media Independent Interface provides 10/100/1000T speeds. The SysAce controller provides access to a 512 MB compact flash card that can be used to provide non-volatile storage while the DDR2 memory controller interfaces with either 256 or 512 MB of DDR2 RAM with an effective I/O clock frequency of 200 MHz. An example BSB design is shown in Figure 2.3.

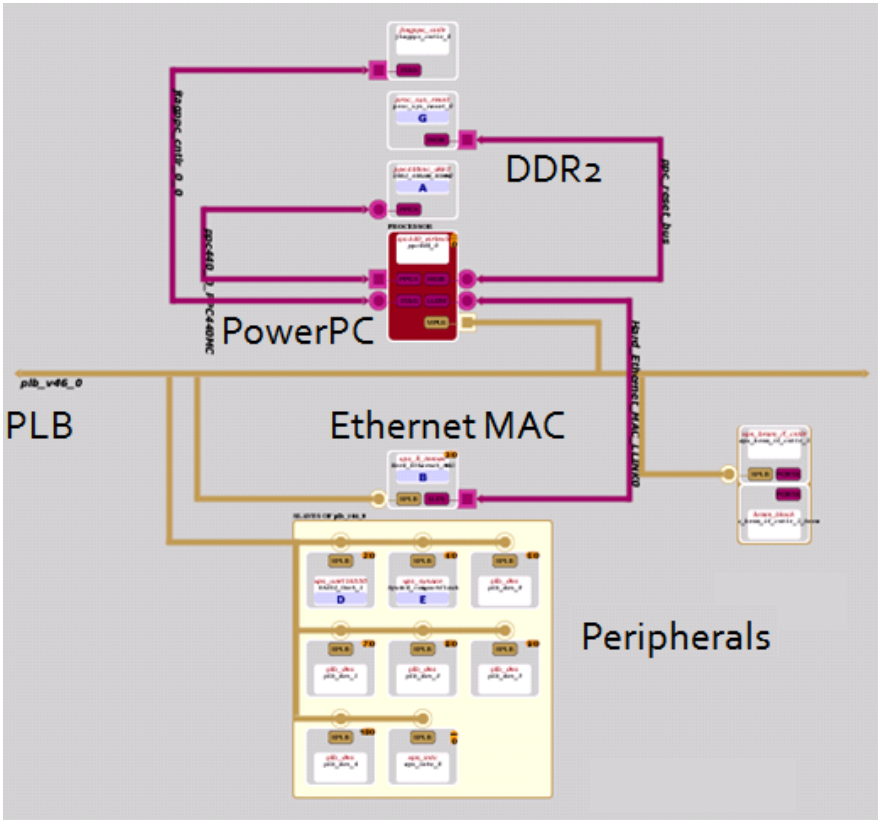

Figure 2.3: Virtex-5 FXT Base System Builder Design [12]

Chapter 3

Related Research

While the developed framework does not build on any single previous research effort, various related works have been published in recent years. To provide an understanding of the current research landscape, a selection of previous publications is summarized here. These related efforts serve mainly to illustrate the gap that this thesis fills in the current field as well as to motivate future research combining previous successes in application development with the added functionality available using the developed framework for FPGA clustering. This chapter is organized into research related to hardware interaction via the Linux operating system and previous implementations of MPI on FPGAs.

3.1 Linux on FPGAs

While not mainstream, the deployment of small Linux distributions on Xilinx FPGAs has been performed in several university efforts [28][21], is commercially supported through MontaVista and WindRiver Linux distributions [39][33], and is minimally supported by Xilinx [24]. Two distinct approaches are targeted toward the two available processor types, the embedded PowerPC processors on Virtex-4/5 FX-series FPGAs and Microblaze soft cores. The former option allows a standard base kernel, compiler, and application stack while the Microblaze requires specialization at all levels to compensate for the lack of a memory management unit (MMU) in version 6.0 and older Microblaze processors. μClinux is the dominant Linux-based distribution for embedded systems ported to Microblaze processors by John Williams at the University of Queensland that emulates memory

management[15].

Unfortunately, significant effort is still required to port some applications to a μClinux environment, especially those applications with many dynamic memory allocations and complex runtime interactions. For example ,porting MPI was completed only after considerable time commitment that resulted in a closed software solution. For this reason, along with improved performance, a PowerPC based environment was chosen as the development target for this work. Other academic efforts selecting this same development path have provided additional literature regarding the use of hardware accelerators implemented in the reconfigurable fabric. These efforts largely focus on the configuration of a single FPGA with a hardware accelerator addressable by a software process to enable a particular behavior.

3.1.1 IRS

One such target behavior is dynamic reconfiguration. With the ability to reconfigure a portion of the FPGA fabric at runtime, single hardware acceleration units can be modified over the course of an application runtime to provide best-suited computation capability. The use of mutable hardware units complicates system design however, and implies a need for a generic, portable, and robust interface layer. One method for establishing such a layer while retaining application flexibility is through a device driver that establishes memory maps and interrupt services for newly reconfigured hardware elements such that these details are abstracted from the interacting application.

Torsten Mehlan and colleagues at the Chemnitz University of Technology implemented a driver and API called the Interface for Reconfigurable Systems (IRS) that demonstrates the above method [27]. While the driver did not directly control the hardware accelerators through read and write operations, robust configuration capability through kernel-level resource access is demonstrated and sufficient abstraction through a user space library and management daemon is provided. As dynamically reconfigured hardware is in some degree more volatile than the flexible but static reconfiguration supported in this thesis, a similar

approach is expected to provide sufficient capabilities.

3.1.2 Hardware/Software Codesign

Increased application portability and deployment capability through high-level language (HLL) based system design is another behavior that is possible to enable through driver usage. Traditionally, the ability to write software that seamlessly integrates hardware and software units through a design flow where portions of an application are compiled down to bit streams have been limited by a number of factors. Often, specific annotations are required and certain language constructs are prohibited. One common prohibition is the use of pointers, which are particularly troublesome when interfacing between a managed virtual memory address space such as in Linux and a hardware unit with direct physical access but no knowledge of virtual memory, including address translation and page structure. Allowing an application to use common pointer values across software and hardware then becomes impossible without circumventing these issues.

To provide such behavior, a combination of hardware DMA and a device driver for control communication was presented as part of the work done by Lange and Koch [23]. As a refined solution, applications were stored entirely within a physically contiguous DMA buffer and the physical to virtual memory offset of this buffer was communicated to the hardware, allowing common pointers to be used across hardware and software with transparent hardware address offsets. Besides demonstrating another beneficial upshot of driver-based hardware addressing, this effort shows that caution is required when using shared memory between the CPU processes and hardware units, specifically with regards to page boundaries.

3.1.3 Platform for Particle Physics

Finally, the use of device drivers under Linux across multiple FPGAs was demonstrated in the construction and analysis of a general purpose computation platform for particle physics [25]. In this work, researchers across several universities collaborated to build a

network of five Xilinx Virtex-4 FX FPGAs. Physics event data was streamed to four of these boards via Gigabit Ethernet connections and inter-FPGA communication was performed across RocketIO serial ports forming a fully connected network. These FPGAs performed algorithmic computation while the fifth FPGA acted as a switch managing communication. Character drivers were used to feed data to the hardware via standard *write* and *read* commands and also to initialize DMA transfers and interrupt handlers. Special character device files were created a priori in the */dev* folder and applications were made to communicate both off-chip and with the driver interface through file operations.

3.2 MPI on FPGAs

In addition to work done supporting device driver abstraction of hardware within a Linux environment, a body of work supporting the use of MPI as a communication API on FPGAs has been advanced in recent years. These works primarily focus on implementing MPI at a hardware level and either forgo operating systems entirely or take the alternative path discussed earlier of using μClinux on Microblaze processors.

3.2.1 TMD-MPI

As the best example of the first method forgoing an operating system entirely, a group of researchers from the University of Toronto have constructed a scalable FPGA-based multiprocessor using Xilinx Virtex-II Pro FPGAs [29]. The system architecture uses custom PCBs with nine FPGAs arranged in a fully-connected network over custom multi-gigabit transceiver links. Each of these "clusters" of nine FPGAs has eight computing FPGAs and one FPGA for handling inter-cluster communication with other identical boards over Ethernet.

The most closely related portion of this effort is the TMD-MPI stripped down MPI communication layer used to support both on and off chip communication and allow an MPI programming model to be used in application development [34]. The MPI layer is

built in a bottom-up manner similar to previous embedded MPI development [26] where the base six MPI instructions (Init, Finalize, Comm_size, Comm_rank, Send, Receive) are implemented and collective operations are built out of combinations of sends and receives. The complete TMD-MPI implementation includes the functions in Figure 3.1, however some have reduced functionality.

MPI_Init	Initialize TMD-MPI Environment
MPI_Finalize	Terminate TMD-MPI Environment
MPI_Comm_rank	Get rank of calling process
MPI_Comm_size	Get number of processes
MPI_Wtime	Report number of seconds elapsed during execution
MPI_Send	Sends message to single destination
MPI_Recv	Receive message from single destination
MPI_Barrier	Global Synchronization
MPI_Bcast	Broadcasts message to all other processes
MPI_Reduce	Reduces values from all processes to single value
MPI_Gather	Gathers values from each process

Table 3.1: TMD-MPI Functionality Table

Because no operating system is used, there are several outstanding issues in TMD-MPI. One of the most severe is that processes must be statically started and ranks cannot be dynamically assigned. Another issue is that only synchronous sends and receives are implemented. Since the most common MPI send/receive commands are locally complete non-blocking, and immediate send/receive commands are also commonly used, this would require (sometimes significant) reimplementation of cluster computing algorithms and incurs additional latency as a result of the required handshaking.

That being said, much of what was accomplished in this effort is directly relatable to the implementation of a more fully functioned and flexible MPI implementation on FPGAs. For on-chip communication, the Fast Simplex Link (FSL) was used to allow up to 16 separate compute units (either hardware or microprocessor) to send information back and forth via MPI send/receive pairs. To allow this to work without an operating system, a hardware MPI engine was designed and implemented to manage the buses and interpret the messages formats [35]. This engine coordinated with the FSL to send packets off-chip

as well. Such a fully connected network was valuable for application flexibility and the use of FIFOs proved to be sufficient means of communication.

3.2.2 FPGA Cluster-on-Chip

Another method of implementation, similar to TMD-MPI in that an MPI communication layer is used among computing elements on an FPGA fabric, is supported by the body of research performed at the University of Queensland [43] [40]. Again, the FSL is used to allow FIFO-queue based communication, however in this case, a single processor is the root node and may either use an embedded Linux operating system like μClinux or low level firmware similar to TMD-MPI. The first application to be targeted to this platform model was image processing, however to implement their test system, a image processing library had to be built on top of the MPI layer. This was done to aid the computing nodes in interpreting the data received and, while understandable, implies a lack of flexibility in the implementation. This effort showed similar scaling across multiple Microblaze cores connected via the FSL, using a reduced MPI implementation supporting the same functionality as TMD-MPI.

The same group investigated operating system abstractions across Microblaze soft cores veering away from MPI and instead implementing a more traditional Unix-style inter-process communication model with a μClinux head node that spawned processes to secondary processing units on the same FPGA [45]. A number of micro benchmarks were used to measure performance of the master, slave, and I/O subsystems. An interesting result they obtained was that the implementation of a large number of soft-core devices provided much higher performance than the inclusion of dedicated hardware, however the hardware was not particularly optimized and could, by their own admission, be improved to a similar level of performance.

Both of these implementation strategies show the feasibility of implementing multiple computing elements on a single FPGA fabric using scalable communication frameworks relying on the internal FIFO communication structures. Although future research has been

proposed in the area of multiple FPGAs, nothing has yet been published, and no indication has been given that suggests that future research will substantially overlap the stated contribution of this thesis.

Chapter 4

Framework Overview

In this chapter, the framework supporting cluster computing on commodity FPGAs is presented. Components of the framework span software, hardware, and interaction across that boundary to produce a functioning whole. Through careful abstractions and a well-defined interface, hardware and software design efforts are decoupled and flexible application development is accommodated. The chapter concludes with practical examples demonstrating the familiar programming model implied by the framework and intuitive methods of hardware/software interaction.

4.1 Software Environment

On one side of the overall hardware/software codesign effort, a base software environment has been developed that supports application development and deployment over multiple FPGAs with multiple independent hardware accelerators. This environment is composed of a Linux operating system deployment and OpenMPI application stack.

4.1.1 Linux Kernel

Distribution of the open source Linux operating system is organized into many different *distros* such as Ubuntu or Fedora that bundle applications, libraries, and utilities, all of which run on top of a system-level software layer called the Linux kernel. This kernel acts as the interface between hardware and the aforementioned higher-level software. As such, it has access to physical memory space, I/O ports, and processor features that it then abstracts to provide a portable API to other applications. Given such low-level access

and variability between platforms and processor architectures, a kernel must be compiled specifically targeting the desired platform.

This targeting is accomplished through a combination of configuration files that select various support options to be included or ignored at build time and kernel branches at supply architecture-specific functionality. Examples of architecture-specific features include memory management unit interaction, mutex lock behavior and IRQ handling. Configuration options, alternatively, deal with higher-level support features such as whether to support a particular file system, include a device driver, or enable features such as virtualization support.

With regard to Xilinx FPGAs, several factors must come together to provide platform support. Most essential is processor support, as memory management, program execution, operand format, etc are all foundational features. The processors included in the hybrid FPGAs are all in the PowerPC 4xx series with the 440 included in Virtex-5 FPGAs and the 405 included in previous generations. Both of these chips are supported in the PowerPC branch of the Linux 2.6.29 kernel and use the common *ARCH=powerpc* build parameter. The PLB interconnect mechanism is an IBM CoreConnect architecture and as such is recognized as a generic "Simple Bus" by the Linux kernel and is trivially handled. Finally, the devices connected to the PLB must be supported by device drivers in order for Linux to be able to interface with them effectively.

Xilinx provides a minimal set of drivers supporting Virtex-4 and 5 series FPGAs. Importantly, drivers are available for the ll_temac Hard-wired Ethernet MAC with various interface methods (MII/GMII) and System Ace compact flash. These drivers and other kernel configuration options are all enabled in a configuration file targeted during kernel compilation. To assist in the configuration file creation process, Xilinx provides default configurations for the Virtex-4 ML405 and Virtex-5 ML507 reference designs. Using these as a template, features can be added and modified through the *menuconfig* make target and manual modification of the configuration file. Required modifications for framework use

include selecting System Ace device driver and Ext2 file system support and changing the default ll_temac interface method from GMII to MII. The later change is required to support the ML410 and ML510 reference designs that do not include GMII hardware support.

4.1.2 Root File System

An appropriate kernel configuration is just one part of a functioning Linux operating system. The other significant aspect is the creation of a root file system (RFS) hosting and supporting any applications and utilities that interface with the kernel. The RFS that is included in large distributions like Ubuntu provides a very rich feature set that comes with appropriately high processing and storage requirements. As the general purpose computing power of the embedded PowerPC processors and DDR RAM interface is well below that of modern desktop components and storage space on compact flash is at a premium, the RFS built for this framework offers more limited features with priority placed on libraries and utilities that directly contribute to the functionality of inter-node communication and hardware interaction.

In addition to providing kernel support for their FPGAs, Xilinx also provides an example RFS that is limited to a single user, *root*, a few basic utilities like *ls* and *cd*, and a simple FTP hosting utility. This is enclosed in a 4 MB ramdisk, which can be downloaded to the FPGA along with the kernel and hosts the RFS in main memory. While RAM disks can be resized and modified within a development environment, they have the notable downside of taking up limited memory resources. Even very modest feature sets can increase the size to a large fraction of the available 256 MB of RAM. As such, the decision was made early on to host the RFS on compact flash.

Starting from the contents of the Xilinx-provided RFS, functionality was added in an iterative way working towards a functional deployment of OpenMPI. OpenMPI is an open-source implementation of the MPI-2 standard that merges and builds on the previously

distinct implementations, FT-MPI, LA-MPI, LAM/MPI, and PACX-MPI. OpenMPI is developed and supported by a consortium of commercial, governmental, and academic institutions including Los Alamos National Laboratories where it is used on the Roadrunner BlueGene/P, currently the fastest supercomputer in the world [41]. OpenMPI benefits from an entirely native implementation in C/C++ that does not rely on higher level languages like Python.

Looking down the application stack, OpenMPI relies on remote terminal connections to each node from each other node. The preferred and secure method for establishing these connections is to use the Secure Shell (SSH) protocol. SSH is organized into a client/server architecture and uses public key encryption to provide credentials and authenticate access. This process involves the creation of a pair of keys, one public and one private, for each host. The public key is then given to a remote server and stored in a known-hosts file there. The private key is used to digitally sign messages, which the remote server can determine originate from the known host via the public key. OpenSSH is an open source implementation of the SSH protocol.

In turn, SSH relies on library functions to make keys and perform encryption/decryption. These libraries are included in the general purpose OpenSSL toolkit. OpenSSL also implements the Secure Sockets Layer (SSL) and Transport Layer Security (TLS). Additional library support underlying OpenSSL is provided by the open source 'zlib' compression library. Finally, additional system utilities were required beyond the base set included in the Xilinx RFS, so the BusyBox embedded utility package [6] was built implementing these additional utilities.

While conceptually simple, building an embedded Linux operating system requires significant development effort. To build the kernel and all applications, a PowerPC 4xx cross compilation environment [18] was first configured in a VMWare virtual machine running CentOS. The existing RFS was placed in a local directory and targeted as the installation

location during subsequent build configurations. Each application and library was then acquired; configured with host, target, and miscellaneous options; compiled; and installed. This process was complicated by limited and/or obtuse configuration options that did not expect cross compilation. Therefore some options like *strip* and *ranlib* that perform file maintenance and library manipulation were not available or required specific and poorly documented configuration.

In addition to the binary executables and newly created libraries installed during this process, shared libraries located in the host's embedded development kit were often required for runtime execution. The embedded RFS cannot support a full compliment of shared libraries, so these shared libraries were added to the RFS after being identified using a combination of the PowerPC *readelf* utility and error code solution searches. Several miscellaneous configuration changes were then made to configure SSH keys, known hosts, environment variables, and to start up processes before full functionality was reached.

Once all configuration had been accomplished, a fully operational OpenMPI application stack was implemented. In this way, applications written using the MPI API calls could be deployed to a cluster of FPGAs using the above described RFS, properly configured kernel, and hardware platform created with the Xilinx BSB.

4.2 Hardware Template

Due to the flexible nature of the developed framework, no restrictions are placed on the hardware design beyond the method of interfacing with the rest of the framework. As such, and with a desire to integrate well with current design tools, the following describes the recommended way to construct a template for new hardware accelerator designs. This template provides the required interface constructs for two-way data and control signal communication and integrates seamlessly with overall systems designs.

4.2.1 Hardware Accelerator Creation and Structure

Once a base system has been created as detailed in Section 2.2.1, a second Xilinx-provided wizard within their EDK is used to create the template for a new hardware accelerator. This "Create/Import Peripheral Wizard" generates a slave interface wrapper for the PLB and optionally provides several useful features for bus interfacing. Of the features provided, three are explicitly used in the larger framework. A read/write FIFO pair acts as the communication buffering and abstraction mechanism, a software reset register allows hardware to be interrupted mid-computation, and an interrupt generator can provide "data ready" interrupts to the PowerPC. This PLB wrapper and bus interfacing structures serve as the base template upon which any and all hardware accelerators are built.

The FIFO pair provides two custom length queues holding 32-bit values. The length can be chosen as any power of two between 4 and 16384 as required for the application. As an easily extensible model demonstrating correct use of the handshaking protocol and vacancy calculation, an example state machine is implemented that connects the Write FIFO to the Read FIFO such that each value written to the Write FIFO can be read from the Read FIFO in sequence. The read and write registers are exposed as physical addresses on the PLB with constant offsets of 0x400 and 0x600 respectively from the peripheral base address. A status and control register used for reseting each of the FIFOs and measuring occupancy is exposed in the same way at 0x300 and 0x500 offsets for the Read and Write FIFOs.

The interrupt generation unit also provides an example state machine that generates interrupts when a 30-bit counter rolls over to 0. This occurs approximately once every 10 seconds at 100 MHz. A control register used to enable interrupts is exposed with an offset of 0x200. Finally, the software reset is controlled by a register exposed at offset 0x100 and connects to the FIFOs and user logic state machines.

Once a hardware unit template is created in this way and the state machines are modified to provide the desired functionality, instances of the peripheral can be added to the base system design. Once there, the PLB slave is connected to the PLB mastered by the

PowerPC. Then, if the application makes use of interrupts, a connection can be made between the interrupt port and the Xilinx XPS Interrupt Controller component that manages all the interrupts in the system. Finally, a base address for each instance can be generated so that the registers mentioned above are assigned unique physical addresses on the PLB.

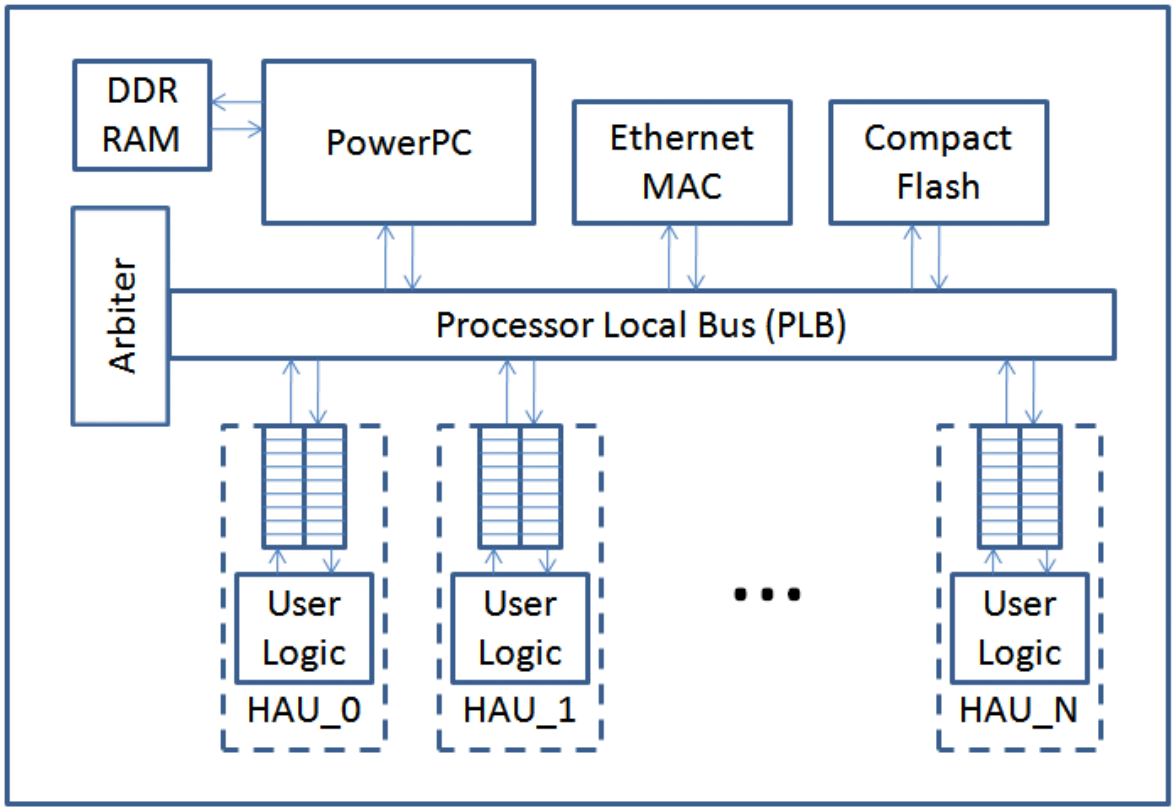

Figure 4.1: Structure of Hardware Acceleration Units on the Processor Local Bus

At this point, any design can be simulated, synthesized, and even tested in hardware using standalone C-language programs running on the PowerPC core. A self-test program is generated automatically, demonstrating access methods that can serve as a valuable template for testing in this fashion. In this way, the hardware design process can proceed in an isolated fashion without regards to the remainder of the framework.

4.2.2 Device Tree Specification

Once a base system is generated and any hardware accelerators are added to the design, an integrated device tree utility is used to generate a device tree specification (DTS) file for the hardware platform. This option is chosen through the "Software Platform Settings" in the Xilinx EDK and acts as an alternative to the stand alone operating system. Once *device-tree* is chosen as the desired OS, the UART device can be selected as the console output and kernel boot arguments can be specified.

The boot arguments allow the special device file for console output (eg. ttyS0) and the root file system(RFS) location to be specified. The RFS can be mounted from a ram disk at */dev/ram*, the System Ace compact flash at */dev/xsa2*, or a network file system (NFS) location */dev/nfs*. A static IP address or dynamic DHCP provided address can also be specified with *ip=192.168.0.1* or *ip=on* respectively. Read/Write access to the RFS can also be specified with *rw*.

Once all desired boot options are selected, libraries and BSPs can be generated, which rebuilds the ppc440_0 directory under the XPS project and generate a *xilinx.dts* file. This file can then be targeted during the Linux kernel build to direct the device drivers and platform settings to be configured appropriately for the hardware. Figure 4.2 shows an example DTS entry for a hardware accelerator. The importance of the individual fields is explained in the next section.

```
plb_des_0: plb-des@c9c00000 {
  compatible = "xlnx,plb-des-1.00.a";
  interrupt-parent = < &xps_intc_0>;
  interrupts = < 2 2 >;
  reg = < 0xc9c00000 0x10000 >;
  xlnx,family = "virtex5";
  xlnx,include-dphase-timer = <0x1>;
} ;
```

Figure 4.2: DES Hardware Accelerator DTS Entry

4.3 Hardware/Software Interfacing

With a functioning cluster of FPGAs capable of distributing jobs using MPI and the ability to create custom hardware acceleration units connected to a larger hardware design, the missing piece is a method for linking these concurrent software processes to hardware accelerators. In the following section, the driver development required to provide this connection is detailed and the hardware abstraction process is explained culminating with a description of the implied programming model.

4.3.1 Driver Structure

The construct that the Linux kernel uses to communicate with hardware is the device driver. Each device or device type has it's own driver that implements behavior for several exposed *routines*. Broadly, there are three driver classifications: character devices, block devices, and network interfaces [9]. Network interfaces deal with packet construction/parsing for communication with a network device. Block devices host file systems and are managed as such by the kernel. Character devices are more generic and handle streams to and from an accessing application. Hardware accelerators fit nicely into the last category as they accept input data and/or instructions and return results.

As such, a general purpose character device driver has been written to manage the interface hardware described in Section 4.2. Just as the interface hardware is intended to allow flexibility in accelerator design and only define methods for integration with software, this device driver provides applications with a well-defined method of interacting with hardware that leaves further software design decisions to the developer. To provide this clean, hardware agnostic interface, the driver level handles hardware-specific considerations like memory addresses, IRQ lines, and device registration.

The driver/device initialization process follows two steps. The first step is driver registration. When the driver is included in the kernel build, the boot process calls a predefined initialization function that in turn calls the *of_register_platform_driver(struct of_platform_driver)*

function. The parameter structure includes two important fields: a list of compatible devices IDs and a probing function. The compatible device list uses the same syntax as the *compatible* field in the DTS file entry shown in Figure 4.2. When the kernel identifies a match between the DTS file and a driver's list of compatible devices, it then probes the device using the probing function provided in the *of_platform_driver* structure.

In the context of the general purpose character driver written to interface with hardware accelerators, the probe function performs all of the allocations and reservations that do not change throughout the lifetime of the operating system. If any of these required actions fails, then the device is not usable and the entire probing process fails. The first allocation that is required is the major device number. Device numbers are the method Linux uses to uniquely identify hardware devices. Organized into a set of two numbers, major and minor, major numbers are traditionally used to designate drivers and minor numbers are used to identify individual devices using that driver. Because of the volatile nature of the devices attached to this driver where each device can perform different functions and the numbers of each device can change without restriction, the use of unique major numbers for each device was desirable. The specific reason for this requirement is made apparent later.

The next registration to be performed is the file operations that define the *read*, *write*, *open*, and *release* behavior. These functions, the contents of which will be described shortly, are stored in a *file_operations* structure. This structure along with the device number is used to register a *cdev* structure with the device. This structure associates system calls on an appropriately numbered device with the kernel-level file operations providing the device-specific behavior.

Next, the physical memory region defined in the *reg* field of the DTS file entry is reserved with *request_mem_region* and then remapped into the operating system managed virtual memory space with *ioremap*. If these memory reservations and remappings complete successfully, addresses for each of the registers in the hardware interface (FIFO data and reset, interrupt control, and software reset) can be calculated with the constant offsets

provided in Section 4.2.

Similar to how virtual memory is used at an operating system layer to abstract and manage physical memory resources, physical interrupt request lines (IRQs) are mapped into virtual IRQs. This results from the historic scarcity of IRQ lines managed by operating systems. DTS file entries also provide physical IRQ information in the *interrupts* field when an interrupt port is connected to the system interrupt controller. If this field exists, the physical IRQ is remapped to a virtual IRQ, and the virtual IRQ is set to *NO_IRQ* otherwise. To allow the driver to sleep while waiting for an interrupt, a wait queue is also initialized. Linux processes handle sleeping by waiting on a wait queue and are awoken by another process waking up that wait queue.

Finally, a semaphore is created as a mutex lock and initialized in unlocked mode. This lock is present to provide exclusive read/write access to an application in a thread-safe way. If multiple processes attempt to access the same hardware resource, only one will be able to acquire the lock and continue to operate on the device. A single structure stores all of the device-specific constructs described here including the addresses, *cdev* structure, IRQ, wait queue, mutex lock, and various flags.

4.3.2 File Operations

Open and Release

As mentioned in the *cdev* registration description, functions implementing the device-specific behavior of file operations are central to the operation of a character driver. The first operation to be called when accessing a character driver is *open*. As such, it manages setup and initialization that satisfies the preconditions of later write and read operations. The two parameters passed into the *open* function are pointers to *inode* and *file* structures. The *inode* structure contains a reference to the *cdev* structure and the *file* structure has a private data field that can be used to store a data structure to be used during read and write operations.

The device-specific structure populated during probing is the containing structure of

cdev, and is located using the *container_of* function. This structure is then stored in the *file* private data field for future use. Immediately before this storage, the process attempts to acquire the lock with the *down_trylock* function. If this is successful, then the mutex lock is "locked" and subsequent processes that attempt to acquire the lock will fail.

Before leaving the *open* function, the hardware accelerator and FIFOs are reset by writing the reset condition, *0x0000000A*, to each register. Interestingly, the *iowrite32* and *ioread32* functions invert the byte order from big-endian to little-endian when used in this platform. This property is not documented and was challenging to identify as the repeating versions, *iowrite32_rep* and *ioread32_rep*, do not exhibit this behavior. Finally, the IRQ was reserved and interrupts enabled if interrupts were used.

The second file operation examined is *release*. The purpose of releasing the device is to free up used system resources and ready the device to be opened by a later process. To accomplish this, the IRQ resource is freed and the mutex lock is unlocked. This causes any subsequent interrupts to be ignored and allows a new process to acquire the lock. Other operations performed during the opening process can then reset, reassign, or harmlessly reinitialize the appropriate values.

Read and Write

As explained in Section 4.2, two hardware FIFOs are generated that act as Read and Write FIFOs. Therefore read operations retrieve data values from the Read FIFO address and write operations push data to the Write FIFO address. Both *read* and *write* functions are passed the same parameters: the same *file* structure, whose private data field is populated during opening, a character array buffer, a byte count, and an offset.

The *write* function receives the designated number of bytes to be written in the character array buffer. This data is in the user memory space, and therefore needs to be copied into allocated kernel space before the kernel writes it to the hardware. This requirement is due to the potential for user-space memory to be paged out or invalid. The *copy_from_user* function handles these problems transparently and allows the *write* function to be reentrant,

a requirement when other kernel-level routines can be executing concurrently. The copied array buffer can then be written to the Write FIFO data register using the *iowrite32_rep* command, after which the kernel memory space is freed.

The *read* function performs similar operations, with the optional additional feature of waiting on interrupts. In the reading case, the buffer is simply a user space pointer where the data should be copied upon retrieval from the device. Again, kernel memory space must be allocated and then the *ioread32_rep* command is used to read the indicated number of bytes. The kernel memory is then copied into user space with *copy_to_user* before being freed. When interrupts are being used, the process waits on the wait queue with the condition that a *read_ready* flag in the device-specific structure is set. If this flag is not set when the thread is awoken, the *read* function is restarted and goes back to sleep. Otherwise, the *read_ready* flag is cleared and the read continues as it does without interrupts. An interrupt handler is a function registered with the IRQ during the opening process that is invoked when an interrupt is received on that IRQ line. In this case, the handler sets the *read_ready* flag and wakes up the wait queue before returning.

4.3.3 Device Addressing

While the described driver abstracts communication and flexibly generates interfaces for any number of devices implementing custom behavior, addressing and interacting with these interfaces remains difficult without added support at the operating system level. The most important construct supporting the driver framework is the Linux concept of special device files. These files typically reside the the */dev* directory and are very helpful for interaction with any hardware devices from user-space applications. These special device files are created using the *mknod* command and can be configured as character or block devices using the major and minor device numbers to link with the appropriate device.

In the implemented framework, these special device files serve the dual purpose of guiding device interaction and exposing the current hardware configuration. While the first purpose is implicit and described in greater detail in the next section, adaptive modification

aligning the available special device files with implemented hardware accelerators requires custom behavior. The result of this alignment is that for each hardware accelerator of type *example*, a special device file named *exampleX* is created in */dev* where *X* is a unique number starting from 0 and going up to one less than the number of devices of that type. As such, a configuration with two *addition* accelerators and three *multiplication* accelerators would appear in the */dev* directory as shown in Figure 4.3.

```
# ls -la /dev
...
crw———-    1 root     root     249,   0 Jan  1  1970 addition0
crw———-    1 root     root     250,   0 Jan  1  1970 addition1
...
crw———-    1 root     root     251,   0 Jan  1  1970 multiplication0
crw———-    1 root     root     252,   0 Jan  1  1970 multiplication1
crw———-    1 root     root     253,   0 Jan  1  1970 multiplication2
...
```

Figure 4.3: Directory Listing of /dev

A small executable was developed and is executed at boot time that creates these special device files so that they align with the current configuration. The configuration is gathered from the */proc/devices* file which lists all devices in the system. The character device driver registration that occurs during probing causes each device to be included with the major device number and a device name. The nonstandard device number usage mentioned previously where major numbers are unique to each hardware accelerator results from the exclusion of minor device numbers in this file. The device name registered during character device registration is the device type as provided in the DTS file entry with "user-" prepended. By searching for user-prepended entries in this file, hardware accelerators can be identified regardless of name. The same configuration of *addition* and *multiplication* hardware accelerators would appear as shown in Figure 4.4. Once the hardware accelerator are identified, new special device files are created that link to the correct major number.

Finally, a utility function for use in application development was implemented that accepts a device name (eg. *addition*) and returns a FILE pointer to a special device file. This

```
# cat /proc/devices
...
249 user-addition
250 user-addition
251 user-multiplication
252 user-multiplication
253 user-multiplication
...
```

Figure 4.4: Contents of /proc/devices File

is accomplished by opening each */dev/additionX* file starting with $X = 0$. When multiple processes are attempting to access the same type of device, the mutex lock in the driver will act as an arbitration mechanism only allowing the first process to attempt to acquire the lock to receive a valid FILE pointer. When the file does not exist or is already locked by another process, X increments and the process repeats until a valid file is found or sixteen attempts are made. Sixteen is an arbitrary, but reasonably high upper bound on the number of hardware accelerators that could be included on a single FPGA. The API call to reserve a special device file is *FILE *AllocateResource(char *name).*

4.4 Programming Model

The described framework provides support for a flexible and scalable parallel configuration of hardware accelerators on multiple FPGAs. While valuable, this contribution would be unlikely to encourage further research and development if it relied on an unfamiliar programming model with exotic API. Indeed, one of the goals of this framework is to retain a familiar program structure and exploit standard APIs. As such, the *AllocateResource* utility function is the only API call that is not part of standard C language libraries or the MPI standard.

To interface with the hardware accelerator and invoke the device driver file operations, standard C language file access functions are used. Once a valid FILE pointer has been returned from *AllocateResource*, the functions in Table 4.1 can be used for all communications with the hardware accelerator. A virtual buffer also must be associated with the

FILE pointer to force data to be immediately forwarded to the file operations using the *setvbuf((FILE *)device, null, _IONBF, sizeof(int))* command.

System Function Call	Invoked Driver Function
fopen(FILE * device)	open
fwrite(void * data, size_t size, int count, FILE * device)	write
fread(void * data, size_t size, int count, FILE * device)	read
fclose(FILE * device)	release

Table 4.1: File Operation Mappings

This mapping of C language file operations to driver level functions represents the interaction between developer-controlled software development and the constant interface layer provided by the device driver. This pairs with the FIFO interaction state machine design separating the interface layer from the independent hardware development. A separation is therefore implied between hardware and software development that effectively decouples the hardware/software codesign effort as shown in Figure 4.5

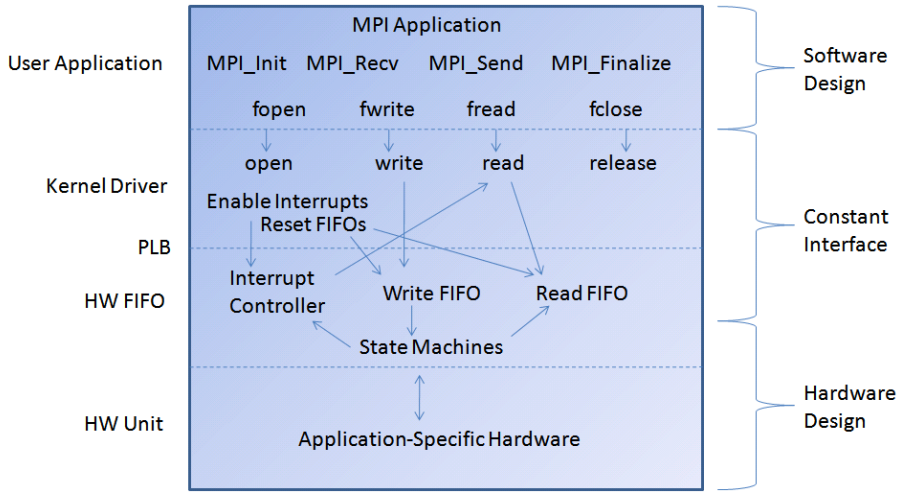

Figure 4.5: Hardware/Software Interaction Stack

Interprocess communication is handled by OpenMPI and therefore makes use of the same syntax and structure as other MPI applications. MPI supports any virtual topology

desired, but most commonly implements master-slave and process tree hierarchies. The following examples illustrate how to realize simple examples of these topologies when hardware accelerators are performing all computation on a single integer. For both examples, common setup steps are required to declare variables and initialize the MPI environment.

```
#include<stdio.h>
#include“mpi.h”
#include“fpga_util.h”
int main(int argc, char* argv[]){
  FILE *device;
  int rank, size, output;
  int *input;
  MPI_Init(argc, argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  ...
```

Figure 4.6: Common Initialization Code

For the master-slave topology, the master often distributes work and manages returned results while not performing any program computations. Although master-slave interaction can be quite complicated when the application requires complex and/or dynamic control, in this example, the master simply sends a command line parameter to all slaves, allows them to perform some computation and then aggregates the results. Figure 4.7 illustrates the virtual topology graphically.

In this topology, process *0* broadcasts the command line data to all other processes, each of which associate themselves with a hardware accelerator of type *example*, pass the received data to the hardware, and receive the result. The master then aggregates these results with a collective reduction and prints the input and output. If the hardware was configured to do nothing other than return whatever data is sent to it, the output would be equal to the command line parameter times one less than the number of processes. While this is a contrived example, especially because all processes would have access to the command line parameters, it demonstrates the ease in which data can be communicated among an arbitrary number of processes and transferred to hardware for custom processing.

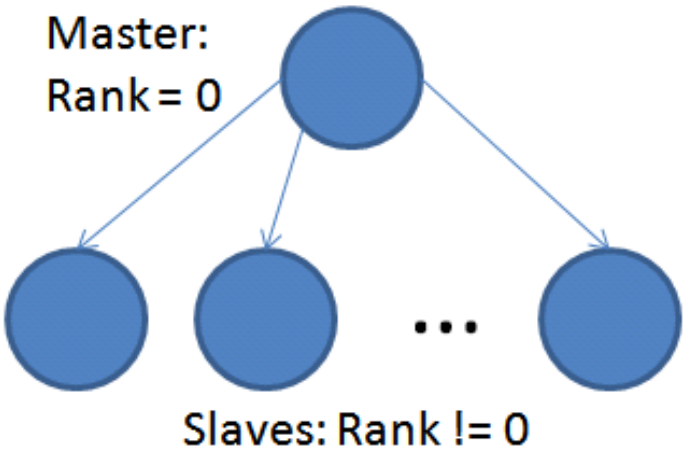

Figure 4.7: Example Master/Slave Virtual Topology

As a second example demonstrating a process tree topology, an inefficient $X = (A*B+C*D)$ operation is implemented using three processes. Distinct hardware units performing multiplication or addition are assumed to be available and a separate process controls each operation as shown in figure 4.9. The four operands are again passed via command line.

In this example, process 0 and 1 have identical behavior with the exception of command line parameter index and therefore are able to execute the same instructions. Process 2 receives the products resulting from the other processes and performing the sole addition, and as such executes different instructions. Common *fread* and *fwrite* commands emphasize how the hardware abstraction framework allows implementation details to be hidden completely from the application developer. The execution time of each type or even of different implementations of the same type of hardware accelerator could vary widely and the application would still generate the correct result without any additional developer effort.

By presenting these simple but illustrative examples, the reader should be able to clarify their understanding of the programming model implied by the hardware/software framework and gain an appreciation for the capability and flexibility in development that is provided. The next chapter will build on this understanding by presenting two non-trivial applications and discussing the required design considerations. The performance results of

```
int main(int argc, char* argv[]){
  // Initialization Code
  ...
  input = (int *)malloc(sizeof(int));
  if(rank == 0){
    *input = atoi(argv[1]);
  }
  MPI_Bcast(input, 1, MPI_INT, 0, MPI_COMM_WORLD);
  if(rank != 0){
    device = AllocateResource("example");
    setvbuf(device,null,_IONBF,sizeof(int));
    fwrite(input, sizeof(int), 1, device);
    fread(&output, sizeof(int), 1, device);
    fclose(device);
  }
  MPI_Reduce(&output, &output, size, MPI_INT, MPI_SUM,
                    0, MPI_COMM_WORLD);
  if(rank == 0){
    printf("Input = %d, Output = %d", *input, output);
  }
  MPI_Finalize();
  return 0;
}
```

Figure 4.8: Master/Slave Topology

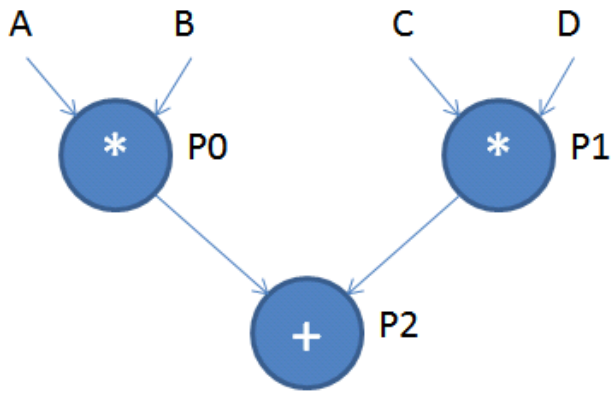

Figure 4.9: Example Process Tree Virtual Topology

hardware accelerated computation in comparison to software-only solutions will demonstrate the strengths and limitations of the implemented framework and provide motivation and direction for future research.

```
int main(int argc, char* argv[]){
  // Initialization Code
  ...
  input = (int *)malloc(2 * sizeof(int));
  if(rank ¡ 2){
    device = AllocateResource(“multiplication”);
    setvbuf(device,null,_IONBF,sizeof(int));
    input[0] = atoi(argv[rank*2 + 1]);
    input[1] = atoi(argv[rank*2 + 2]);
    fwrite(input, sizeof(int), 2, device);
    fread(&output, sizeof(int), 1, device);
    MPI_Send(&output, 1, MPI_INT, 2, 0, MPI_COMM_WORLD);
  }
  if(rank == 2){
    device = AllocateResource(“addition”);
    setvbuf(device,null,_IONBF,sizeof(int));
    MPI_Recv(input[0], 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
                MPI_STATUS_IGNORE);
    MPI_Recv(input[1], 1, MPI_INT, 1, 0, MPI_COMM_WORLD,
                MPI_STATUS_IGNORE);
    fwrite(input, sizeof(int), 2, device);
    fread(&output, sizeof(int), 1, device);
    printf(“Output = %d”, output);
  }
  fclose(device);
  MPI_Finalize();
  return 0;
}
```

Figure 4.10: Process Tree Topology Implementing X = A*B + C*D

Chapter 5

Application Case Studies

5.1 Testbed Configuration

The testbed targeted with the developed framework is composed of Xilinx Virtex-4 FX and Virtex-5 FXT FPGAs configured as described in Section 4.2. A single Virtex-4 XC4VFX60 FPGA is included on a ML410 evaluation board hereafter referred to by the ML410 designation. As shown in Figure 2.1, the ML410 includes 25,280 slices, 4,176 Kbits of distributed block RAM, 128 DSP48 slices, and two embedded PowerPC 405 processors. For the Virtex-5 entries, two XC5VFX70T and two XC5VFX130T FPGAs are included on ML507 and ML510 evaluation boards respectively. The smaller ML507 device features 11,200 slices, 5,328 Kbits of distributed block RAM, 128 DSP48E slices, and one PowerPC 440 processor. Finally, the larger ML510 includes 20,480 slices, 10,728 Kbits of distributed block RAM, 320 DSP48E slices, and two PowerPC 440 processors. Only one PowerPC is used in the developed framework.

Each FPGA is preloaded with a compact flash card containing the root file system supporting the Linux operating system and OpenMPI software environment. The two smaller FPGA types have 256 MB of DDR2 RAM on board while the ML510 has 512 MB. They are connected together over the RIT campus network and are assigned the unique domain names: ML5071.rit.edu, ML5072.rit.edu, ML5101.rit.edu, ML5102.rit.edu, and ML4101.rit.edu.

To serve as a comparison platform for application performance studies included in the following sections, a Rocks Cluster of 2.33 GHz Intel Xeon 5140 dual core general purpose

processors was used. Each of the 16 nodes in this cluster contains two Xeon chips, 2 GB of RAM, and 250 GB of storage. They are running Red Hat 3.4 Linux 2.6.9 with GCC 3.4.6 and MPICH 1.2.7.

5.2 Matrix Multiplication

The first non-trivial application targeted toward the developed framework is integer matrix multiplication. Linear algebra provides the foundation for many scientific computing applications typically deployed on HPC systems. As such, performance scores in terms of floating point operations per second on the linear algebra benchmarking software, LINPACK, are the sole determinates for supercomputer rankings. FPGAs are notorious for under-performing when targeted for floating point heavy computation due to the large area required to implement fast floating point arithmetic. General purpose processors have multiple dedicated floating point arithmetic units with ASIC speeds, and without an ability to synthesize many more arithmetic units in reconfigurable fabric, FPGAs are unable to outperform GPPs. Arithmetic units for other data formats can be implemented in less complex hardware and as such, FPGAs often implement alternatives like Fixed-Point, Galois Field, or Integer arithmetic. For demonstration purposes, the simplest of these, Integer arithmetic, is targeted.

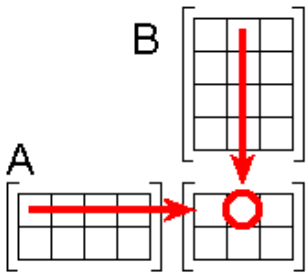

Figure 5.1: C[i][j] is the Dot Product of Row i of A and Column j of B

Matrix multiplication is a fundamental operation that accepts two matrices, *A* and *B*, of size NxM and MxN and produces their product, *C*, an NxN matrix. Each index, *C[i][j]*, is

calculated by performing a dot product between row *i* of *A* and column *j* of *B*.

5.2.1 Hardware

As presented in Section 4.2, the new hardware accelerator template was generated using the Xilinx "Create/Import Peripheral Wizard" and named *plb_matrix*. The strength of FPGAs for arithmetic operations is their ability to perform many operations in parallel. As such, multiple dot products must be performed in parallel to take advantage of the FPGA resources. Looking at the first column of the result matrix, each element, *C[i][0]*, can be calculated by performing a dot product between Row *i* of *A* and the first column of *B*. This process can then be repeated for each column of *C* by subsequent columns of *B*. The implemented design uses this principle and calculates the first *X* rows of *C* where *X* is the number of multiply accumulates that can be performed in parallel.

To have data available for these arithmetic operations, local storage of rows is required. The distributed block RAM resources are well suited for this purpose. Multiplication of large matrices up to 2048 x 2048 elements is targeted and as such, each row stored must contain 2048 32-bit entries. Integer multiplication also requires three of the 24 by 18 bit multipliers contained in the DSP slices. These two criteria guided design and resulted in 32 rows being chosen for *X*. With 32 multipliers and 32 locally stored rows up to 2048 elements, approximately 75% of DSP and BRAM resources are used for the matrix multiplication accelerator on the two smaller FPGA types. This, along with other BRAM resources used by the base system effectively fills the ML410 and ML507 FPGAs. The ML510 with additional resources of both types supports a 64 row version of the matrix multiplier as well. Resource utilization is shown in Table 5.1.

	Registers	LUTs	BRAM	DSP Slices
ML410 (32 MACs)	5 %	8 %	62 %	75 %
ML507 (32 MACs)	6 %	7 %	48 %	75 %
ML510 (32 MACs)	3 %	4 %	24 %	30 %
ML510 (64 MACs)	5 %	7 %	44 %	60 %

Table 5.1: Matrix Multiplication Resource Utilization

This behavioral design implies an interaction specification for the PLB interaction. First, the hardware accelerator must be initialized with matrix dimensions. Then, the 32 rows from *A* are expected and stored as they arrive. Once the rows are stored, columns of *B* are expected. As each column element is received, the 32 multiply accumulates occur in parallel. Once a column has been received, the accumulated values containing elements of the resulting matrix, *C*, are written back. A new column can then be received and the process repeated until all columns have been received after which new rows are expected. This process continues indefinitely to accommodate flexibly sized matrices and variable partitioning schemes. The state machine managing this interaction with the PLB is shown in Figure 5.2.

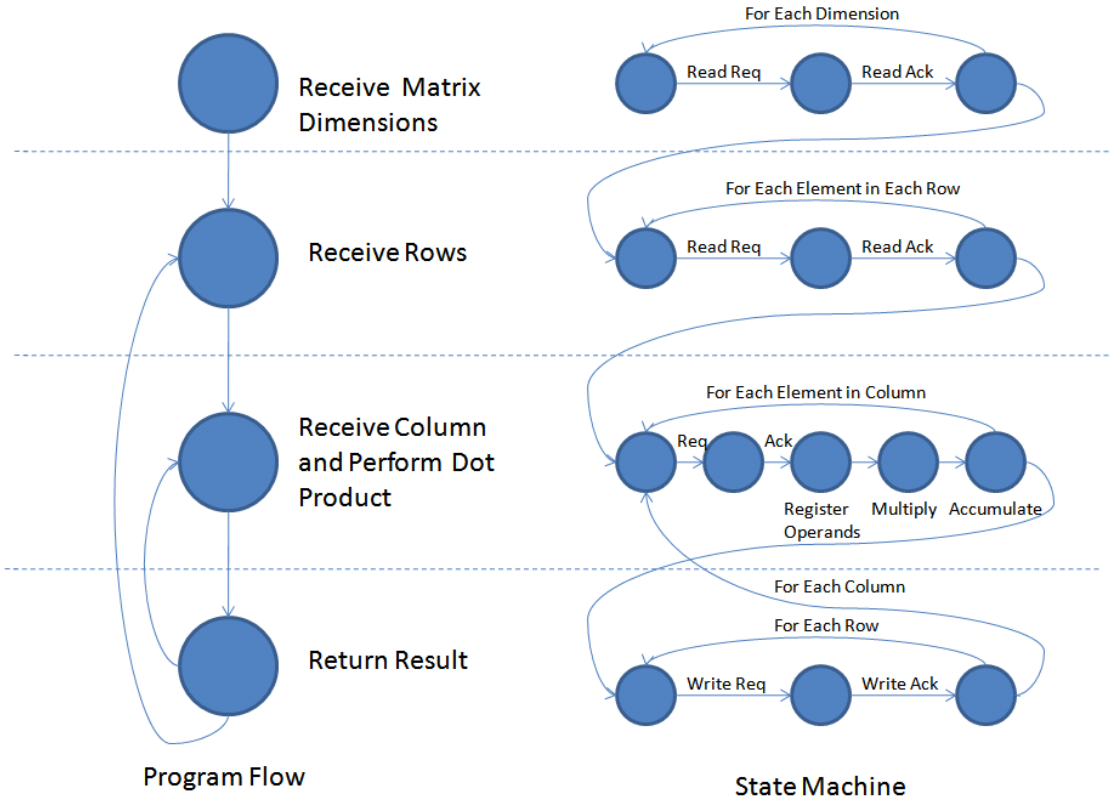

Figure 5.2: Matrix Multiplication Hardware Design

5.2.2 Software

With the hardware interface predefined, software development could proceed independently. Matrix multiplication is a highly data parallel application as each index in the result matrix could be computed independently. With hardware operating on whole rows, the software partitioning could then treat a row as the base unit of work. A basic master/slave paradigm is a natural choice as only one process needs to write out the result matrix. This I/O process can perform it's portion of the matrix multiplication, then collect the results from all other processes. Static partitioning allows simple rank controlled indexing and static communication order to be used to place the slaves' row results into the final matrix.

While final performance results will be presented and explained in the following section, software development proceeded in an iterative manner with an emphasis on performance improvements when possible. The first modification that was made was to read the B matrix in column-order such that columns are physically contiguous in memory. This allowed a single file operation to move a column to the hardware accelerator. It also has the benefit of accessing physically contiguous memory from the PowerPC, which turns out to have a large performance impact. The combination of these changes allowed an approximately 50% performance improvement to be observed during development.

Similarly, the results returned from the hardware are ordered as portions of columns in the result matrix C. For example, after sending the first 32 rows of A and the first column of B, the first 32 elements of the first column of C are returned. Storing these results into their actual locations in C again required non-contiguous access patterns that resulted in severe performance degradation on the order of 200% execution time increases. To avoid this penalty, column segments were stored contiguously in the order received and then memory copied into their appropriate locations in the transpose of C when received by the master process. As such, the result is written to file as the transpose of the result matrix.

An equivalent software-only MPI implementation was written performing sequential multiply accumulates to calculate each index in C. Similar matrix partitioning was used to

ensure a valid comparison. This software implementation was deployed on the cluster of Xeon GPPs described.

5.2.3 Results and Analysis

To begin, single node performance is useful to examine. This begins to elucidate performance of the hardware/software interaction without complication from inter-node communication and synchronization. Multiplications of square matrices of varying size were computed on each FPGA type as well as the a Xeon GPP. The computation times for each platform are listed in Table 5.2 as the average across three runs and shown graphically in Figure 5.3. The number of multiply accumulation units (MACs) inferred in the design are included for reference.

	256 x 256 (s)	512 x 512 (s)	1024 x 1024 (s)	2048 x 2048 (s)
ML410 (32 MACs)	0.162	0.957	6.562	47.139
ML507 (32 MACs)	0.071	0.470	3.427	26.015
ML510 (32 MACs)	0.071	0.470	3.410	26.403
ML510 (64 MACs)	0.043	0.269	1.875	14.269
2.33 GHz Xeon	0.031	0.258	1.956	13.405

Table 5.2: Single Node Matrix Multiplication Execution Times

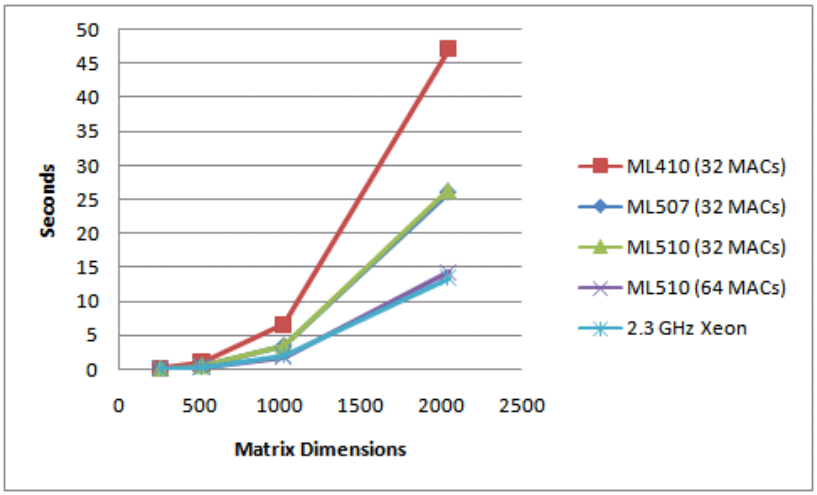

Figure 5.3: Single Node Matrix Multiplication Execution Times

This data shows several things. First, the similarly configured ML507 and ML510

Virtex-5 FPGAs with 32 MACs perform nearly identically. This is to be expected as their architectures are very similar with the exception of increased resources available in the ML510. The ML410 exhibits worse than expected performance, however, highlighting the limitation imposed by the lesser PowerPC 405 processor. The 64 MAC configuration of the ML510 appropriately improves performance by a factor of 1.64x to 1.85x. While a 2x performance increase would be ideal, setup, control, and some data transfers are unchanged with additional arithmetic operations being performed in parallel and thusly are not improved. Finally the performance of all the FPGAs in comparison to the Xeon GPP is disappointing. Figure 5.4 shows how each platform compares to the Xeon.

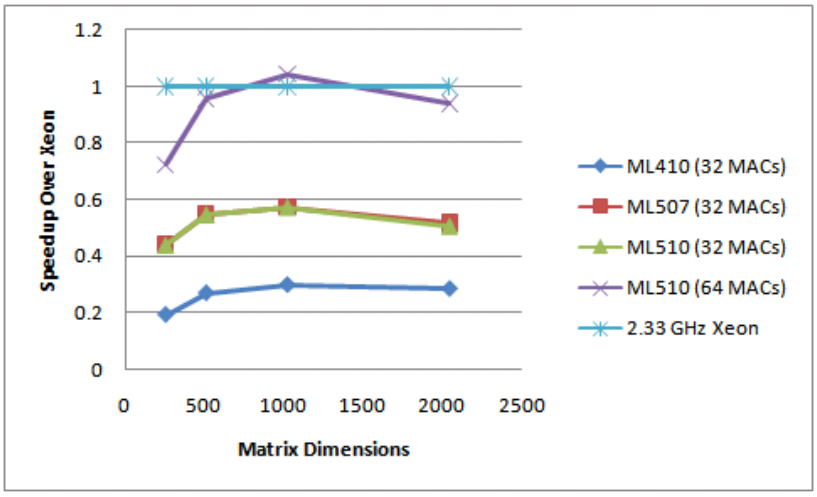

Figure 5.4: Single Node Matrix Multiplication Speedup Over Xeon

With far more concurrent multiplications possible on the FPGAs than on the Xeon, these results require an explanation. The first criteria to examine is how many multiplication operations can actually be performed given the overall state machine, not just the number of MACs. Before calculation starts, the rows of *A* are required. If data is immediately available, three cycles are required for each data element implying $3 * X * N$ cycles where *X* is the number of rows (32 or 64) and *N* is one of the dimensions of the matrices. Following row retrieval, column retrieval and MAC operation requires five cycles for each element of each column, resulting in $5 * N^2$ cycles. Finally, result return requires another

three cycles for each element in the *X* rows, or $3 * X * N$ cycles. During this process, *X* multiplications were performed on *N* elements of *N* columns. This gives the number of MAC operations per cycle as $\frac{X*N^2}{5*N^2+6*X*N}$ or simplified as $\frac{X*N}{5*N+6*X}$.

This information can then be used to examine how efficiently the application makes use of the available MACs each cycle. The entire matrix multiplication algorithm requires N^3 multiplications and with a 100 MHz PLB clock, the 100% efficient use of arithmetic resources would result in an execution time of $\frac{N^3}{100*10^6*\frac{X*N}{5*N+6*X}}$ seconds. Comparing this ideal time with the observed execution time demonstrates the efficiency of the framework in comparison to an ideal hardware usage. This efficiency data is listed in Table 5.3.

	256x256 (%)	512x512 (%)	1024x1024 (%)	2048x2048 (%)
ML410 (32 MACs)	18.573	23.569	26.525	29.007
ML507 (32 MACs)	42.586	47.996	50.788	52.559
ML510 (32 MACs)	42.463	47.871	51.039	51.787
ML510 (64 MACs)	39.500	44.838	48.103	48.795

Table 5.3: Matrix Multiplication Arithmetic Efficiency

As expected, the use efficiency of the arithmetic units on the FPGAs is low, especially for the ML410. Unfortunately, even with 100% efficiency, performance would be very comparable to the Xeon GPP. This is a limitation of the hardware design and FPGAs generally. Redesign could address some limitations but, with resources nearly fully utilized and an upper bound on the realizable MACs/cycle near *X*/3, FPGAs could not solidly outperform a high-end GPP for this application. While matrix multiplication exploits the many arithmetic units available, the other strengths of FPGAs in operating over non-standard bit widths and performing complicated logic operations are not utilized. As such, GPPs are already well suited to this type of computation and the motivation for FPGA use is removed.

Additional useful framework characteristics can be elucidated through continued examination of the performance data, however. Firstly, the reason for the inefficient use of arithmetic resources is helpful to examine. The key assumption made when calculating the ideal MACs/cycle was immediate data availability. If the FIFOs are not populated, the

number of cycles to retrieve new data quickly increases and the number of MACs/cycle decreases, resulting in the inefficiency observed. The bandwidth of data transfers to and from the hardware accelerator is essential to determine why data is not available immediately to the hardware. During a matrix multiplication, the entirety of *A* is transfered to hardware in groups of *X* rows. In the same way the entirety of *C* is transfered back to software. *B* is transfered to hardware once for each group of *X* rows, or N/X times. This results in a total data transfer of $4 * (2 * N^2 + \frac{N^3}{X})$ bytes. Since the software process never waits for data to arrive, dividing the total data transfer by the observed execution time results in the bandwidth between memory within a software process and the hardware interface FIFOs.

	256 x 256 (MBps)	512 x 512 (MBps)	1024 x 1024 (MBps)	2048 x 2048 (MBps)
ML410 (32 MACs)	16.150	19.732	21.731	23.490
ML507 (32 MACs)	37.031	40.183	41.610	42.563
ML510 (32 MACs)	36.924	40.078	41.815	41.938
ML510 (64 MACs)	36.492	38.990	40.272	39.976

Table 5.4: Matrix Multiplication Bandwidth

The data listed in Table 5.4 shows that this bandwidth reaches steady state near 23 MBps for the Virtex-4 and 42 MBps for the Virtex-5s. The discrepancy between 32 and 64 MAC configurations of the ML510 is likely a result fewer transfers of larger blocks of memory when working with 64 rows. This bandwidth represents a performance bottleneck that will cause many applications to perform inefficiently. Removing the PowerPC from the data provider role would alleviate much of the time spent sending each data value, and therefore a direct memory access (DMA) method is suggested as a high priority for future development. See Section 6.1 for additional discussion of this and other directions for further research and development.

Even though the preceding single-node analysis has shown matrix multiplication to perform poorly compared to modern GPPs, the scalability of this application across multiple FPGAs remains a useful characteristic to investigate. Each FPGA contains a single matrix multiplication hardware accelerator due to the area requirements in Table 5.1 and the

bandwidth bottleneck that would cause multiple accelerators to compete over bus usage. Starting with a single ML507, additional FPGAs are added as indicated in Table 5.5 to produce the execution time trend shown in Figure 5.5. The 32 MAC version of the ML510 was used for ease of comparison.

	256 x 256 (s)	512 x 512 (s)	1024 x 1024 (s)	2048 x 2048 (s)
ML507	0.071	0.470	3.427	26.015
2xML507	0.068	0.325	1.889	14.091
2xML507, ML510	0.059	0.297	1.484	10.225
2xML507, 2xML510	0.051	0.277	1.238	7.739
2xML507, 2xML510, ML410	0.098	0.441	2.081	11.930

Table 5.5: Matrix Multiplication Execution Time Scaling

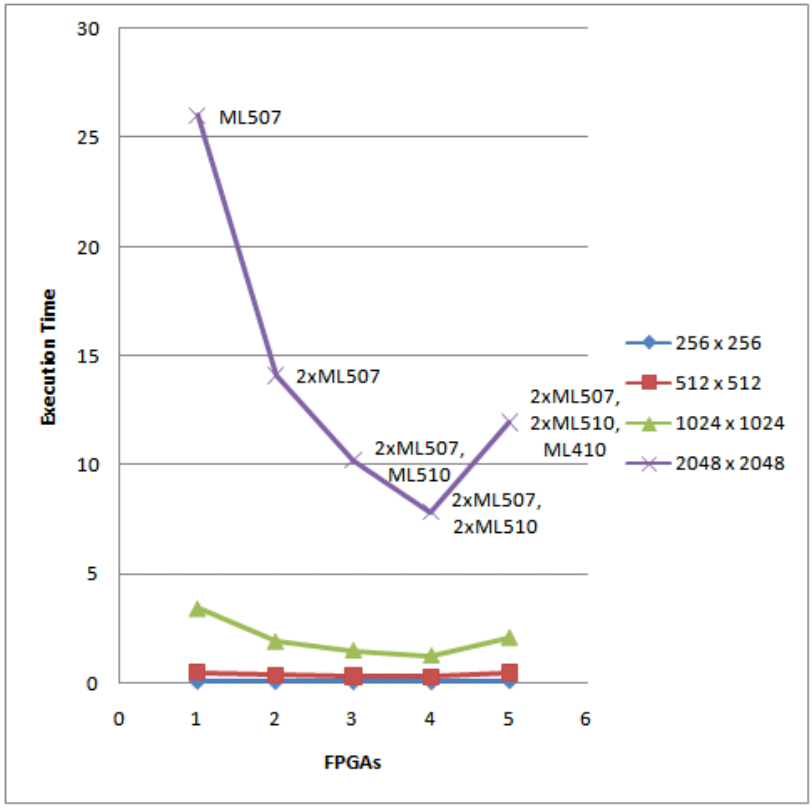

Figure 5.5: Matrix Multiplication Scalability Trends

For the four Virtex-5 devices, performance scales as expected, with a downward trend

toward some asymptote implied by Amdahl's Law where result aggregation becomes the dominant program segment. The larger matrices scale better than the smaller ones, as more work is done by each node and communication time is significantly smaller than computation time. This is clearly illustrated in Figure 5.6 where observed execution time is compared to ideal scaling of single-node execution time. Unfortunately, once the ML410 is added, this trend is reversed and additional time is required to calculate all matrix sizes. The reason for this reversal is that the ML410 takes significantly longer to compute the same portion of the matrix product when compared to one of the Virtex-5's. The static partitioning used allocated the same portion of work to all FPGAs and caused the Virtex-5's to finish early while waiting on the Virtex-4.

	256 x 256	512 x 512	1024 x 1024	2048 x 2048
ML507	1.000	1.000	1.000	1.000
2xML507	0.523	0.723	0.907	0.923
2xML507, ML510	0.040	0.526	0.770	0.848
2xML507, 2xML510	0.345	0.425	0.692	0.830
2xML507, 2xML510, ML410	0.182	0.257	0.390	0.507

Table 5.6: Matrix Multiplication Scaling Efficiency

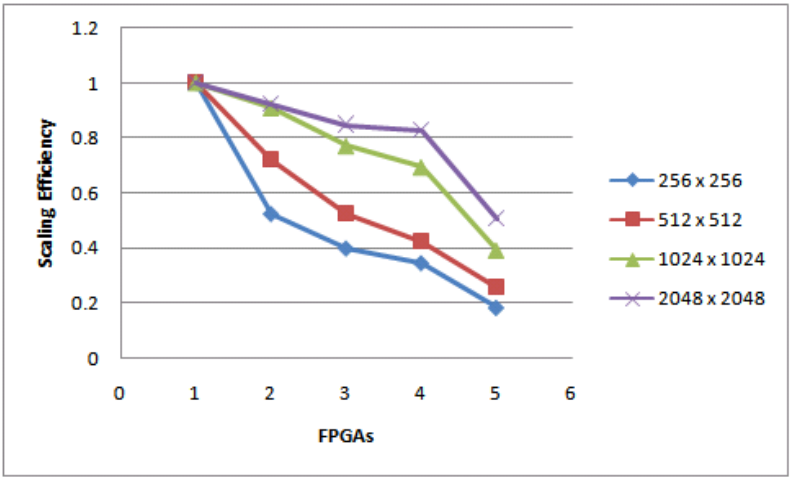

Figure 5.6: Matrix Multiplication Scaling Efficiency

While deployment of matrix multiplication on the cluster of FPGAs using the developed framework was not entirely successful in meeting performance goals, several important characteristics of the framework were identified. Beyond single-node performance of matrix multiplication, PowerPC to hardware accelerator bandwidth was shown to be near 23 MBps for the Virtex-4 and 42 MBps for the Virtex-5, which will guide future application development and suitability analysis while promoting innovative approaches to increasing this transfer speed. The scalability efficiency data listed in Table 5.6 shows that for large matrices, scalability across hardware accelerators with similar computational capabilities retains a high proportion of the ideal scaling factor across multiple FPGAs. It also warns against using static partitioning in a heterogeneous computation environment, a lesson which is immediately applied in the following case study.

5.3 DES Cryptanalysis

While matrix multiplication is not well-suited due to bandwidth limitations, applications exhibiting a lower communication to computation ratio allow the performance of the FPGA computation hardware to outweigh data transfer speeds. One such application area is cryptanalysis, in which encrypted data is analyzed and deciphered. Hardware implementations of cryptographic algorithms often offer far greater performance than software equivalents executed on general purpose hardware due to repeated bit-wise operations across large operands for which dedicated functional units do not exist. So while a modern GPP will be able to compute a 64-bit multiplication very quickly, a complex bit-shifted exclusive-or operation will require multiple simpler instructions to be executed. As such, FPGA's have the advantage that the hardware can be tailored specifically to the computations required and provide more functional units capable of performing those computations concurrently.

One such cryptanalysis target is the Data Encryption Standard (DES). DES is a well-known block cypher created in the 1970's as a Federal Information Processing Standard for the United states. As such, it experienced heavy use within government and commercial

enterprise for the better part of three decades and continues to be used in some settings today. The algorithm uses a 64-bit cryptographic key to encrypt 64-bit blocks of data, or plain text, and produce a 64-bit block of encrypted cypher text. This cypher text can then be communicated to another party and, with knowledge of the same cryptographic key, be decrypted. Any third party would be unable to retrieve the original plain text without knowing or correctly guessing the key with which it was encrypted.

DES cryptanalysis attempts to "crack" this encryption via the later option, correctly guessing the key. Of the 64 bits in the key, only 56 are used for encryption while eight are reserved for parity checking, and as such, there are 2^{56} possible keys. Primitive computing power available in the first twenty years of DES use made this effort impractical, as the months or years required to guess each possible key and provide the original plain text would typically eliminate the value in knowing the contents of a message. As discussed in Section 2.1, improved hardware and distributed computing technology allowed this time to be reduced to hours in the late 1990s, rendering DES practically insecure. The Advanced Encryption Standard (AES) has since superseded DES as the recommended FIPS encryption standard for this reason, however DES continues to be used in some settings.

While complicated approaches have been shown to conceptually reduce the number of keys that must be guessed before arriving at the correct key, the most basic and practical method is a known plain text attack. For this attack, a 64-bit plain text and the corresponding encoded cypher text are assumed to be known in advance. The plain text is then encrypted with each guessed key and the resulting cypher text is compared against the correct cypher text. When the correct text is found, the key used to generate that cypher text can then be used to decrypt the entire message regardless of length by applying the key to each 64-bit block of encrypted cypher text.

A distributed DES cryptanalysis engine has been developed using the FPGA clustering framework and deployed on the testbed configuration described in Section 5.1. This engine coordinates the guessing of each of the possible 2^{56} keys for a provided plain text / cypher

text pair and reports the key used for encryption. The following sections describe the hardware and software required to implement such an engine and present performance results.

5.3.1 Hardware

A DES guessing engine is composed of one or more DES encryption units with some comparison logic to determine if the correct key has been found. Each encryption unit implements the DES algorithm such that the provided plain text and guessed key produce the corresponding cypher text. The inner workings of this algorithm are presented here for completeness and clarity.

Encryption consists of sixteen stages, or *rounds*, in which the key and input data interact to produce an intermediate 64-bit value that feeds into the next round. This overall scheme is shown in Figure 5.7 and is proceeded and followed by bit-wise permutation (initial and final) functions that serve no cryptographic function but are difficult to implement in software. The 64-bit permuted plain text is initially divided into two 32-bit halves that are represented by the left and right vertical lines.

The heart of each round is the *Feistel* function as shown in Figure 5.8. The *E* block expands the right half block to 48 bits after which and exclusive OR (XOR) is performed with a sub key generated by the key scheduler discussed later. The resulting 48-bit value is the divided into 6-bit indexes into eight *Substitution* boxes, or *S-boxes*, which provide the non-linear component of the DES encryption. Such an indexed output is trivially implemented in FPGA look-up-tables or on-chip read-only memory. These *S-boxes* together provide a 32-bit output which is permuted and subsequently XORed with the left half-block producing the next round's right half-block.

The key scheduler follows a similar 16-round approach as shown in Figure 5.9 and produces a different sub key for each round in the Feistel Function. The *PC1* block performs an initial permutation and filters out the parity bits producing two 28-bit key halves. Each half is then shifted left a certain number of bits depending on the round and combined

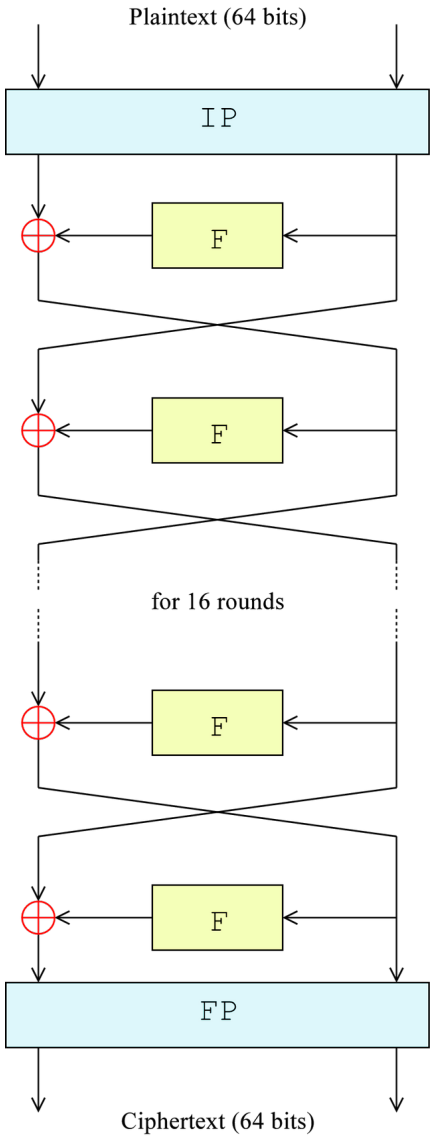

Figure 5.7: Overall Structure of DES algorithm

into a 48-bit sub key using twenty four bits from each half for each round. Details of the permutation operations and *S-box* contents can be found at numerous sources [22].

The implementation of the DES algorithm naturally utilizes a structural design approach connecting data-flow implementations of the various permutation, expansion, and substitution units hereto described. While a purely combinational design is functional, the identical stages lend themselves toward a pipelined implementation that allows a steady-state

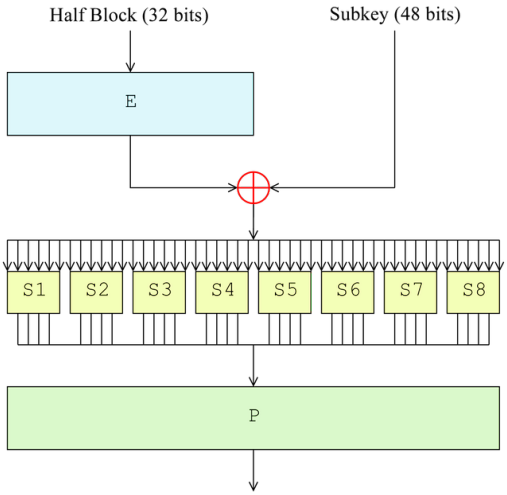

Figure 5.8: Feistel Function (or F-Box) central to DES Encryption

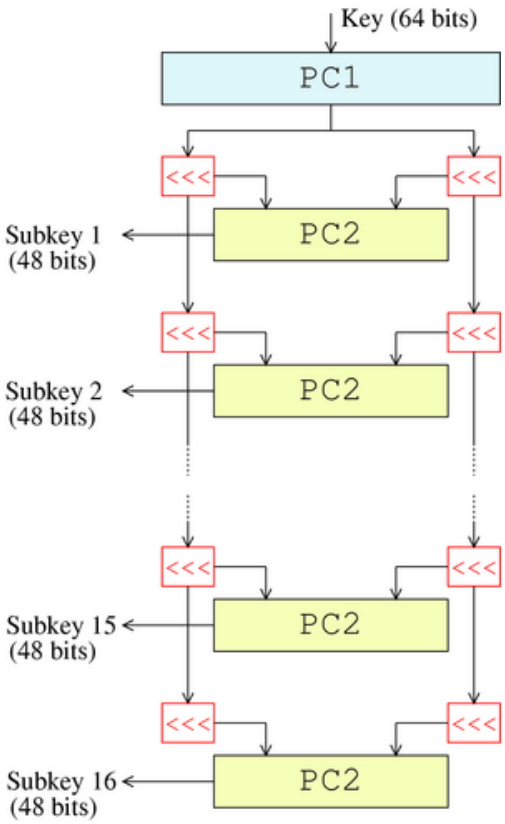

Figure 5.9: DES Key Scheduler Structure

throughput of one key guess each cycle. This level of performance is essential when guessing such a large number of possible keys. Beyond the sixteen pipeline stages implied by

the rounds, three additional stages are used, two for accepting and registering inputs and one for registering outputs.

To construct a key guessing engine, multiple encryption units are attached structurally to logic that feeds them keys and compares the results to the correct cypher text. While a single key guesser could be attached to the PLB, the overhead of this management logic and the bus interface would allow fewer units to fit on a single chip. As such, the final guessing unit organized eight encryption units into each key guesser attached to the PLB as a hardware accelerator. The structure of the key guesser is shown in Figure 5.10.

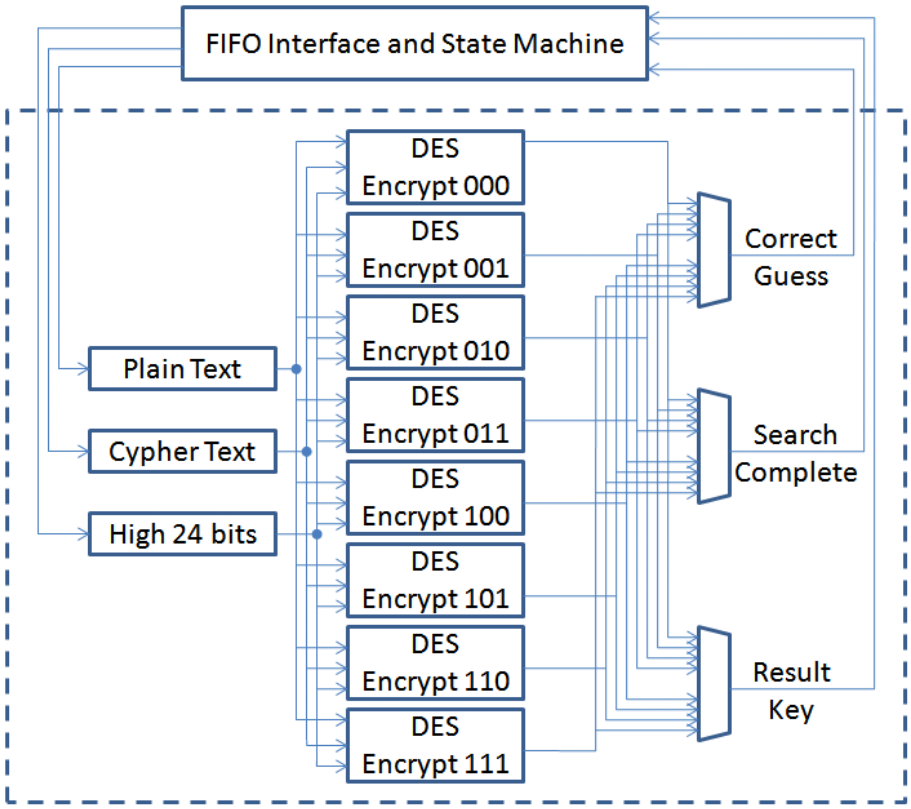

Figure 5.10: Key Guesser Structure and Interface Connections

Overall, the key guesser is responsible for searching all of the keys in a provided subspace of the overall 2^{56}-key search space. This subspace is communicated by specifying the most significant twenty-four bits of the key, which implies a 32-bit key space for the guesser to search. Each encryption unit is then responsible for a subspace of that 32-bit

key space. Similar to the scheme used in the COPACOBANA DES solver [22], each DES encryption unit is assigned a unique 3-bit key prefix which divides the 32-bit search space into eight 29-bit subspaces, one for each encryption unit. Figure 5.11 shows how this key space division occurs.

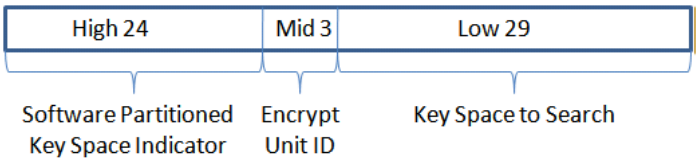

Figure 5.11: Key Space Partitioning

Unlike matrix multiplication, where each arithmetic operation was tightly tied to data availability and therefore integrated into the FIFO interaction state machine, the autonomous key guessing engine is able to receive the initialization information and key space and report back when the key space has been searched. This simplifies state machine design as shown in Figure 5.12. The plain text and cypher text are each communicated as two 32-bit values while the 24-bit key space indicator constitutes the fifth 32-bit configuration value. Subsequent configuration requires only the key space indicator. Results are returned as two 32-bit values containing either the 64-bit key or zero indicating the key is not found. Note that the parity bits ensure that an all-zero 64-bit key is never valid and can therefore safely be used to indicate failure. To allow the software process to sleep while waiting for results, an interrupt generator is also used and an interrupt is generated once the result is placed in the Read FIFO and ready to be read.

The choice of eight encryption engines was not random. Implemented generically, any power of two could have been selected, but eight was ideal when considering the hardware available in the testbed. With space set aside for the PowerPC interface, PLB, and other peripherals, 70% of the FPGA fabric represents a reasonable upper bound on the slice resources that can be consumed by hardware accelerators still allowing the design to be routed and meet timing. As shown in Table 5.7, the two smaller FPGAs, ML410 and ML507, both closely approach this limit with one or two hardware accelerators. If smaller accelerators,

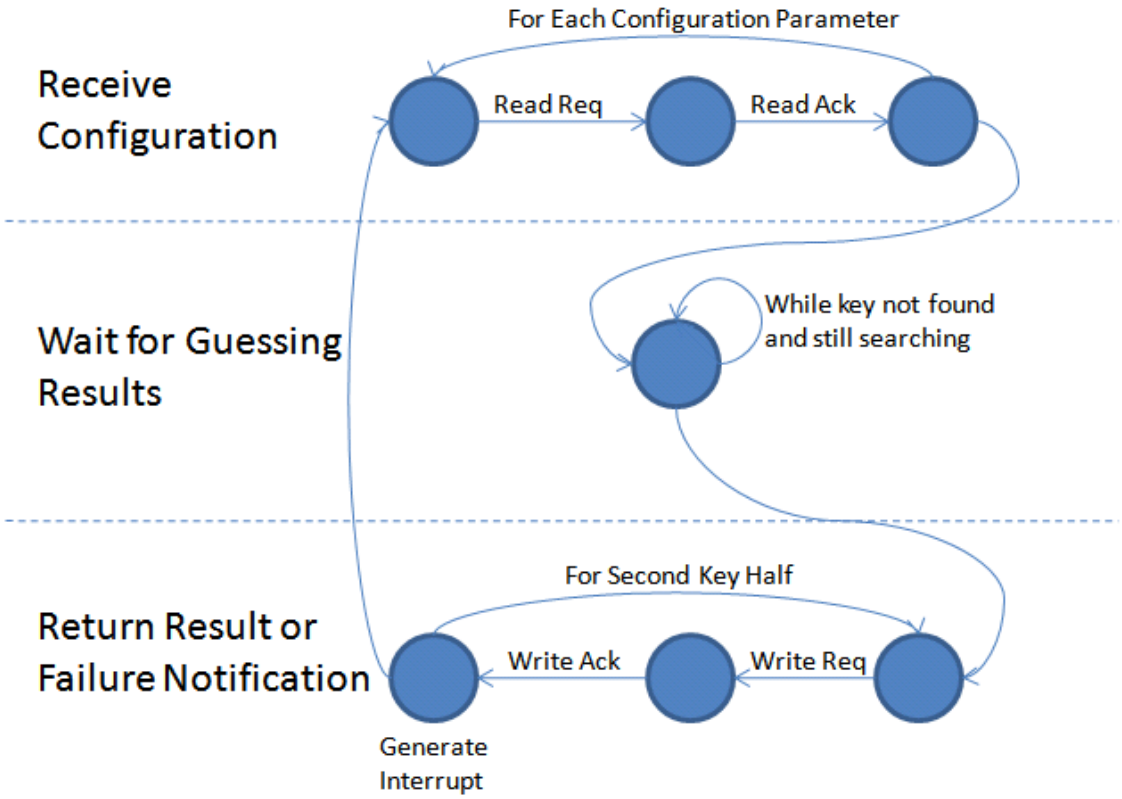

Figure 5.12: DES FIFO Interface State Machine

say with four encryption engines, are used, the overhead associated with the bus interface and other common logic results in fewer total encryption units able to be placed. Larger accelerators with sixteen encryption units would not fit on the ML410 and make no improvement to the capacity of the ML507. The ML510 is noticable under-utilized, which is an unfortunate result of the 32-bit Windows XP development environment which limits application memory usage such that a full ML510 design cannot be synthesized. The design with three hardware accelerators approaches the maximum memory usage and again, changes would only adversely affect the number of encryption engines able to be placed.

5.3.2 Software

As alluded to during the hardware design description, the hardware expects to receive the known plain and cypher text as a one-time initialization, followed by as many key space

	Registers	LUTs	BRAM	DSP Slices
ML410 (1 Unit)	20 %	66 %	0 %	1 %
ML507 (2 Units)	42 %	68 %	0 %	1 %
ML510 (3 Units)	33 %	54 %	0 %	1 %

Table 5.7: DES Cryptanalysis Resource Utilization

indicators as desired. The large key space and potentially variable speed of the hardware accelerators naturally implies a dynamic partitioning of the work, with a central master managing a virtual queue of the 2^{24} key spaces to be checked and distributing spaces to the various hardware accelerators as they return results.

This partitioning begins with each slave process being sent a 24-bit key space indicator. Meanwhile the slaves initialize and configure the hardware accelerator. Upon receiving the key space indicator configuration can complete and the slaves wait on the results. Results are then returned to the master who either responds with a new key space indicator or a stop condition with a '1' in the most significant bit. A new key space indicator instructs the slave to repeat the search process while a stop condition allows the process to exit. Stop conditions are generated if the key is found or the specified search space is complete and no more subspaces are available. For functional testing and performance gathering, search spaces smaller than the entire 2^{56} possible keys are necessary.

A separate, but functionally equivalent software-only implementation was also developed using the GNU C Crypt Library. This library supplies an API to set a key and encrypt a block of data. Both the key and the data are supplied as character arrays with a character representing each bit. This inefficient storage mechanism highlights the advantage that a hardware implementation provides. A 32-bit subspace is much too large to search in software for demonstration purposes, so the subspace is reduced to twenty-four bits by using a 32-bit search space indicator.

5.3.3 Results and Analysis

The hardware DES guessing engines are tied directly to the PLB clock at 100 MHz. With the pipelined design supplying one key guess each cycle after a negligible period of pipeline fill and eight encryption units, each hardware accelerator can then guess 800 million keys per second. The first interesting results to be gathered demonstrate how closely the hardware/software framework adheres to this ideal performance when distributed among all of the FPGAs in the testbed. To gather this performance data, a key space of 2^{36} times the number of hardware accelerators was searched with a plain text / cypher text pair deliberately chosen outside of the space to ensure the entire space was searched.

Number of Hardware Accelerators	Execution Time (seconds)	Key Throughput (keys/second)	Efficiency
1	86.835	791.38 M	0.989
2	87.077	1578.35 M	0.986
3	87.461	2357.16 M	0.982
4	87.366	3146.28 M	0.983
5	87.777	3914.45 M	0.979
6	88.334	4667.70 M	0.972
7	88.451	5438.46 M	0.971
8	88.454	6215.16 M	0.971
9	88.303	7004.00 M	0.973
10	88.490	7765.75 M	0.971
11	88.426	8548.55 M	0.971

Table 5.8: DES Performance Data on FPGA Cluster

Table 5.8 shows the changes in execution time, number of keys checked per second, and implied efficiency compared to the ideal 800 million keys per second. Nearly 99% efficiency is observed for a single node, which demonstrates that the software framework provides neglible overhead when applied to a well-suited problem like cryptanalysis. Figure 5.13 graphically illustrates the nearly ideal scaling across the eleven total hardware accelerators distributed across the five FPGAs.

The lack of substantial efficiency degradation when adding additional FPGAs implies

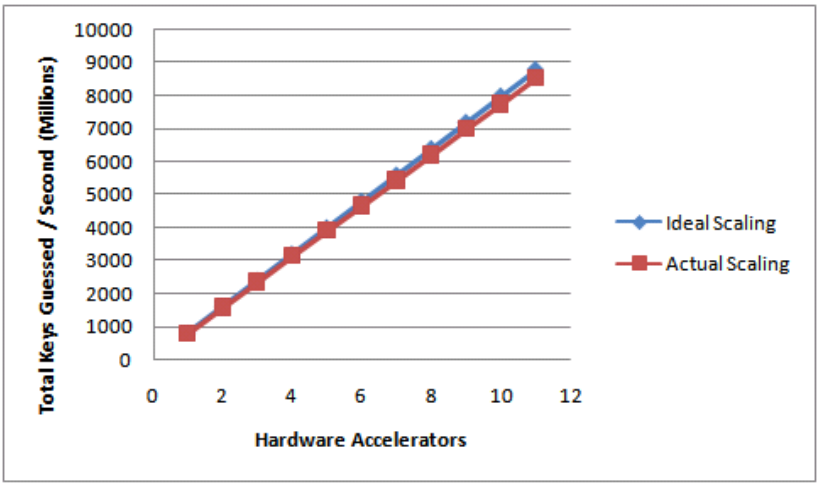

Figure 5.13: Scaling performance compared to ideal hardware speed with increasing key space

that further additions would continue to provide good scalability and computational capability expansion. Figure 5.14 provides a clear indication that while efficiency decreases with additional nodes, performance remains exceptionally close to the ideal speed. Single node efficiency provides a similar lesson, albeit with a relatively steeper decline in efficiency when increasing the number of hardware accelerators hosted on a single FPGA.

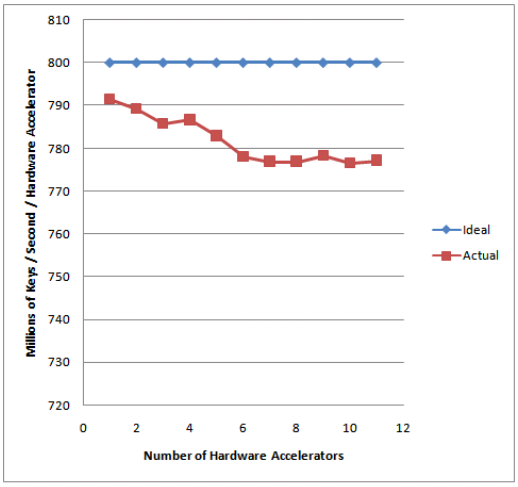

Figure 5.14: Per-Hardware Accelerator Efficiency Scaling

While the above results demonstrate the superb efficiency and scalability of DES cryptanalysis on an FPGA cluster, it is useful to compare absolute performance to other platforms. The first platform of interest is the cluster of GPPs already introduced during the matrix multiplication evaluation. Contrary to the matrix multiplication results, this cluster of 2.3 GHz Xeons is far outperformed by the COTS FPGA cluster. Table 5.9 provides the same performance criteria with efficiency measured in reference to single-node key guess throughput. While slightly more erratic, the scalability shown in Figure 5.15 is comparable to the FPGA equivalent. Overall FPGA system speedup ranges from 1075x to 1115x over the equivalent number of Xeon cores, which implies that approximately 12164 Xeon cores, or 6082 dual-core Xeons would be required to match the performance of the five FPGAs. At current retail prices, the Xeons would cost approximately 253 times as much as the FPGAs.

Number of Xeon Cores	Execution Time (seconds)	Key Throughput (keys/second)	Efficiency
1	365.086	0.735 M	1.000
2	368.629	1.456 M	0.990
3	368.629	2.185 M	0.990
4	369.758	2.904 M	0.987
5	372.570	3.602 M	0.980
6	374.898	4.296 M	0.973
7	374.074	5.023 M	0.976
8	381.945	5.622 M	0.956
9	384.707	6.280 M	0.949
10	385.078	6.971 M	0.948
11	382.406	7.722 M	0.955

Table 5.9: DES Performance Data on 2.3 GHz Xeon Cluster

Finally, it is interesting to compare the capabilities of the COTS FPGA cluster to other commercial FPGA supercomputers. Both SRC Computer Corp. and Cray Inc. provide hybrid reconfigurable supercomputers using the coprocessor model with FPGAs attached to AMD Opteron GPPs. SRC uses Altera Stratix II FPGAs in it's SRC 6 and SRC 7 product offerings while Cray uses Xilinx Virtex-4 LX FPGAs in it's XT5h product. An

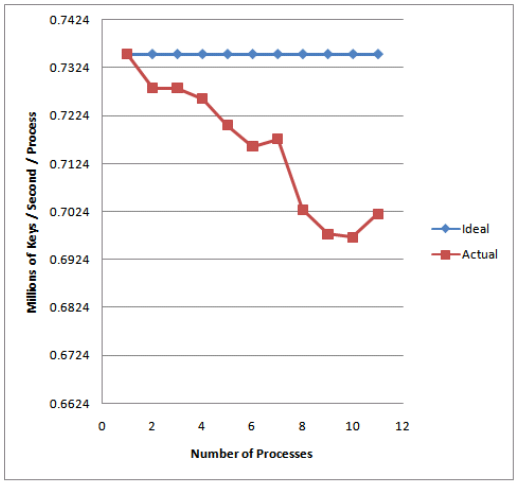

Figure 5.15: Per-Xeon Efficiency Scaling

evaluation of cryptanalysis for high performance reconfigurable computing was included in [17] and DES was one of the algorithms investigated. Only ideal performance of hardware is provided for these commercial supercomputers.

Platform	Key Throughput (keys/second)	Speedup Over Software Solution
Commodity FPGAs	8,548 Million	11630 x
SRC-6	4,000 Million	5442 x
Cray XD1	7,200 Million	9796 x
2.3 GHz Xeon	0.735 Million	1 x

Table 5.10: FPGA Supercomputer DES Performance Comparison

5.4 Inter-node Communication Bandwidth

Finally, to assist future development, a study of node interconnection was performed documenting the communication bandwidth achievable between any two processes. Processes can manage hardware accelerators or simply execute on the PowerPC, but interconnection speeds are reliant on the PowerPC and Ethernet MAC regardless. Communicating processes can execute on the same FPGA or on remote FPGAs and interconnection bandwidth varies between these scenarios.

To measure interconnection speeds using MPI, the open source NetPIPE [42] application was used and performance was gathered on all combinations of two processes running across the three types of FPGAs. NetPIPE measures the time to communicate messages of varying sizes ranging from a single byte to several gigabytes. The bandwidth between remote processes increases up to a steady-state as message sizes increase. Processes running remotely on each type of FPGA were partnered with similar and other FPGA types, however the remote ML410 to ML410 scenario was not performed as only one ML410 was available. The interconnection pairing for remote processes are shown in Table 5.11 along with the associated steady-state bandwidth. Each pairing demonstrates very similar interconnection speeds between 81 and 88 MBps. This is somewhat surprising given that the ML507 uses gigabit Ethernet, and may be caused by a limitation either in PowerPC to Ethernet MAC communication or in the network infrastructure.

	ML410	ML507	ML510
ML507	81 MBps	88 MBps	88 MBps
ML510	88 MBps	88 MBps	88 MBps

Table 5.11: Steady-state bandwidth between remote processes running on pairs of FPGAs designated by the row and column headers

Single FPGA bandwidth between processes is also important as multiple hardware accelerators can be synthesized as in the case of DES cryptanalysis. The OpenMPI libraries manage interprocess communication such that same-node communication does not have to go over the network, and as such bandwidths are higher. This has important ramifications when designing a complex heterogeneous system with varying communication requirements between nodes. Nodes that communicate more often will benefit from being located on the same FPGA by enjoying the higher interprocess communication bandwidth. The bandwidth results are shown in Table 5.12.

Interestingly, bandwidth on a single FPGA increased with larger message sizes up to a four kilobyte boundary, then fell and increased to steady-state as message sizes continued to increase. For the Virtex-5 devices, this early bandwidth peak is far higher than the

	ML410	ML507	ML510
4 KB Message	80 MBps	490 MBps	485 MBps
Steady-state	156 MBps	290 MBps	283 MBps

Table 5.12: Bandwidth between local processes running on a single FPGA

steady-state results with larger messages. The cause of this is suspected to be the memory organization. With messages containing fewer than four kilobytes, a single page of virtual memory is sufficient to store the message and can be communicated by simply copying that page into the address space of the receiving process. With larger message sizes, multiple, possibly noncontiguous pages are required. This implies that memory management rather than the MPI communication libraries are providing the bottleneck for communication bandwidth. Also, the performance limitations of the PowerPC 405 processor included in the ML410 are highlighted here in comparison to the Virtex-5 PowerPC 440 processor.

Chapter 6

Conclusions

This thesis describes a flexible framework for clustering commodity FPGAs into computing clusters. The developed framework allows application-specific hardware accelerators implemented on one or more FPGAs to communicate as fully functioning nodes within an MPI virtual topology. This work supports the current body of research into high performance reconfigurable computing by enabling commodity FPGA resources to be exploited in a clustered environment.

A Linux operating system and OpenMPI application stack were deployed on embedded PowerPC processors include in Virtex-4 FX and Virtex-5 FXT devices. OpenMPI provided a standard parallel programming environment with support for all MPI-2 APIs. A device driver for addressing custom hardware accelerators was developed that supports standard C language file operations for hardware access. A base system design and hardware peripheral template generation procedure were specified for improved hardware design flow.

In addition to explaining the hardware, software, and hardware/software interface development that supports this framework, the implied programming model was illustrated in detail through application case studies. In depth evaluation of matrix multiplication and DES cryptanalysis applications validated the framework function and scaling behavior while elucidating important performance characteristics. Among those characteristics is a PowerPC to hardware accelerator bandwidth of approximately 23 MBps on Virtex-4 and 42 MBps on Virtex-5 devices.

The demonstrated cluster consisting of two ML507 Virtex-5 FXT evaluation boards, two ML510 Virtex-5 FXT evaluation boards, and one ML410 Virtex-4 FX evaluation board

provided a truly heterogeneous development platform. When applied to the well suited application of DES cryptanalysis, a key search rate of 8548 Million keys per second was observed. This is over 97% of the hardware key search rate demonstrating remarkable scaling properties across the eleven independent DES guessing units spread across five FPGAs. When compared to a general purpose 2.33 GHz Intel Xeon processor, speedups in excess of 11,600x were observed.

6.1 Future Work

Perhaps most importantly, the developed framework supports a wide range of further research topics. This section details some of the most apparent and pressing directions for future research and development.

As mentioned in the matrix multiplication application study, low processor to hardware accelerator bandwidth currently limits the efficient use of arithmetic resources implemented in the FPGA fabric. This motivates the inclusion of a direct memory access (DMA) transfer mechanism. Xilinx provides a DMA Controller IP core connected over the same PLB, which may be ideal for this application. The recommended method for DMA data transfers is to send the source and destination addresses to the hardware accelerator FIFO. The hardware accelerator could then communicate directly with the DMA controller to request the desired data. Physical to virtual memory conversion and arbitration among hardware accelerators present the largest design challenges.

Along similar lines, communication between hardware accelerators located on the same FPGA currently requires all data to be sent to the PowerPC where it is transfered into kernel memory space, then into the user process memory space, to another process with MPI, back to kernel memory space, and finally back to hardware. This transfer process could be eased and the bandwidth increased through direct accelerator-to-accelerator communication.

Changing focus to the software environment, many improvements related to robust cluster management are possible. Currently, misuse of hardware accelerators can result in a

kernel panic, transparently causing an FPGA to go offline. While fault tolerance in this scenario is important, the loss of an FPGA during execution also causes the entire MPI application to deadlock if communication with the failed host is required. Addressing both concerns and providing useful feedback to the developer is key to providing a robust cluster environment both for production and development use. Furthermore, application deployment would be eased through the use of third party resource management utilities such as Condor or LSF.

The design flow is also overly cumbersome at this stage of development. While hardware design within Xilinx development tools and software design using standard practices are both smooth, the new hardware design deployment process is fragmented. Kernel configuration and compilation are currently performed manually and both the device tree file and resulting ELF programming file must be manually moved. Perhaps the most cumbersome step is the deployment of a new kernel, which requires either direct connection through Xilinx tools or physical access to the compact flash card. A large opportunity is therefore present to provide an integrated and intuitive design flow that has minimal developer involvement in these trivial design steps.

Finally, an investigation into the suitability of dynamic reconfiguration of hardware accelerators is warranted. With a well-defined interface constant, the behavior of the hardware accelerator could potentially be altered to positive benefit either before or during application deployment. If enough flexibility is available in the extent of reconfiguration possible, the framework would allow complete application target changes without rebooting the operating system or ever recompiling the kernel.

Bibliography

[1] The PowerPC 440 Core. Technical report, IBM Corp, Research Triangle Park, NC, 21 Sept 1999. http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs///852569B20050FF77852569970063431C/$file/440_wp.pdf.

[2] PowerPC Processor Reference Guide. Technical report, Xilinx Inc., San Jose, CA, Jan 2007. Datasheet UG011. http://www.xilinx.com/support/documentation/user_guides/ug011.pdf.

[3] Exploring Science Frontiers at Petascale. Technical report, Oak Ridge National Laboratory, 2008. http://www.nccs.gov/wp-content/media/nccs_reports/Petascale_Brochure.pdf.

[4] MPI: A Message Passing Interface Standard, June 2008. http://www.mpi-forum.org/docs/mpi21-report.pdf.

[5] The Beowulf Project, 2008. http://www.beowulf.org/.

[6] Erik Andersen. BusyBox: The Swiss Army Knife of Embedded Linux. http://www.busybox.net/.

[7] Kevin J. Barker, Kei Davis, Adolfy Hoisie, Darren J. Kerbyson, Mike Lang, Scott Pakin, and Jos Carlos Sancho. Entering the petaflop era: the architecture and performance of Roadrunner. In *SC*, page 1. IEEE/ACM, 2008.

[8] SRC Computers. SRC-7 Overview, 2009. http://srccomputers.com/products/src7.asp.

[9] Jonathan Corbet, Allesandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers*. O'Reilly, third edition edition, 2005. http://lwn.net/Kernel/LDD3/.

[10] Nvidia Corp. Tesla S1070, 2009. http://www.nvidia.com/object/product_tesla_s1070_us.html.

[11] Xilinx Corp. MicroBlaze Processor Performance, 2008. http://www.xilinx.com/products/design_resources/proc_central/microblaze_per.htm.

[12] Xilinx Corp. System Assembly View - Block Diagram, 2009. Xilinx EDK.

[13] Xilinx Corp. Virtex-4 Multi-Platform FPGAs, 2009. http://www.xilinx.com/products/virtex4/.

[14] Xilinx Corp. Virtex-5 Multi-Platform FPGAs, 2009. http://www.xilinx.com/products/virtex5/.

[15] D. Jeff Dionne and Michael Durrant. uClinux Embedded Linux/Microcontroller Project. Arcturus Networks Inc. http://www.uclinux.org.

[16] T Domany and et al. An Overview of the BlueGene/L Supercomputer. In *SC '02: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, Baltimore, MD, USA, 2002.

[17] Tarek El-Ghazawi, Esam El-Araby, Miaoqing Huang, Kris Gaj, Volodymyr Kindratenko, and Duncan Buell. The Promise of High-Performance Reconfigurable Computing. *IEEE Computer Magazine*, 41(2):69–76, 2008.

[18] DENX Software Engineering. Embedded Linux Development Kit, 2009. http://www.denx.de/wiki/DULG/ELDK.

[19] Douglas Gregor and Andrew Lumsdaine. Design and implementation of a high-performance MPI for C# and the common language infrastructure. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 133–142, New York, NY, USA, 2008. ACM.

[20] IBM. 128-bit Processor Local Bus Architecture Specifications, May 2007. http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/3BBB27E5BCC165BA87256A2B0064FFB4/$file/PlbBus_as_01_pub.pdf.

[21] John Kelm. *Running Linux on a Xilinx XUP Board*. University of Illinois - Urbana Champaign, 23 June 2006. http://courses.ece.illinois.edu/ece412/MP_Files/mp2/20060623-XUP-Linux-Tutorial-REVISION-FINAL.pdf.

[22] Sandeep Kumar, Christof Paar, Jan Pelzl, Gerd Pfeiffer, Andy Rupp, and Manfred Schimmler. How to Break DES for 8,980 Euro. In *SHARCS'06 - Special-purpose Hardware Attacking Cryptographic Systems*, Cologne, Germany, April 2006.

[23] H. Lange and A. Koch. An Execution Model for Hardware/Software Compilation and its System-Level Realization. *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pages 285–292, Aug. 2007.

[24] John Linn. Xilinx Open Source Linux Wiki, 2009. Xilinx Corp. http://xilinx.wikidot.com/.

[25] Ming Liu, W. Kuehn, Zhonghai Lu, A. Jantsch, Shuo Yang, T. Perez, and Zhenan Liu. Hardware/Software Co-design of a General-Purpose Computation Platform in Particle Physics. *Field-Programmable Technology, 2007. ICFPT 2007. International Conference on*, pages 177–183, Dec. 2007.

[26] T. P. McMahon and A. Skjellum. eMPI/eMPICH: Embedding MPI. In *MPIDC '96: Proceedings of the Second MPI Developers Conference*, page 180, Washington, DC, USA, 1996. IEEE Computer Society.

[27] Torsten Mehlan, Jochen Strunk, Torsten Hoefler, Frank Mietke, and Wolfgang Rehm. IRS - A Portable Interface for Reconfigurable Systems. In *PARELEC '06: Proceedings of the international symposium on Parallel Computing in Electrical Engineering*, pages 187–191, Washington, DC, USA, 2006. IEEE Computer Society.

[28] Brent Nelson. The BYU Linux on FPGA Project, 2005. http://ccl.ee.byu.edu/projects/LinuxFPGA/.

[29] A. Patel, C.A. Madill, M. Saldana, C. Comis, R. Pomes, and P. Chow. A Scalable FPGA-based Multiprocessor. *Field-Programmable Custom Computing Machines, 2006. FCCM '06. 14th Annual IEEE Symposium on*, pages 111–120, April 2006.

[30] C. Patterson, B. Martin, S. Ellingson, J. Simonetti, and S. Cutchin. FPGA Cluster Computing in the ETA Radio Telescope. *Field-Programmable Technology, 2007. ICFPT 2007. International Conference on*, pages 25–32, Dec. 2007.

[31] Gregory D. Peterson. Programming High Performance Reconfigurable Computers. Final Technical Report AFRL-IF-RS-TR-2003-2, Airforce Research Lab Rome Research Site, Rome, NY, January 2003.

[32] Jacek Radajewski and Douglas Eadline. Beowulf HOWTO. Vol 1.1.1, 22 Nov 1998. http://www.ibiblio.org/pub/linux/docs/HOWTO/archive/Beowulf-HOWTO.html.

[33] Wind River. Board Support Packages, 2009. http://www.windriver.com/products/bsp_web/ bsp_vendor.html?vendor=Xilinx.

[34] M. Saldana and P. Chow. TMD-MPI: An MPI Implementation for Multiple Processors Across Multiple FPGAs. *Field Programmable Logic and Applications, 2006. FPL '06. International Conference on*, pages 1–6, Aug. 2006.

[35] Manuel Saldana. A Parallel Programming Model for a Multi-FPGA Multiprocessor Machine. Master's thesis, University of Toronto, 2006.

[36] R. Sass, W.V. Kritikos, A.G. Schmidt, S. Beeravolu, and P. Beeraka. Reconfigurable Computing Cluster (RCC) Project: Investigating the Feasibility of FPGA-Based Petascale Computing. *Field-Programmable Custom Computing Machines, 2007. FCCM 2007. 15th Annual IEEE Symposium on*, pages 127–140, April 2007.

[37] David E. Shaw and et al. Anton, a special-purpose machine for molecular dynamics simulation. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 1–12, New York, NY, USA, 2007. ACM Press.

[38] Addison Snell, Debra Goldfarb, and Christopher G. Willard. Designed to Scale: The Cray XT5 Family of Supercomputers. Technical report, Tabor Research, Nov 2007. http://www.cray.com/Assets/PDF/products/xt/Tabor_XT5_Whitepaper.pdf.

[39] MontaVista Software. Linux Board Support Packages, 2009. http://www.mvista.com/boards.php.

[40] I. Syed, J.A. Williams, and N.W. Bergmann. A Hybrid Reconfigurable Cluster-on-Chip Architecture with Message Passing Interface for Image Processing Applications. *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pages 609–612, Aug. 2007.

[41] TOP500.org. Top 500 Supercomputing Sites. http://www.top500.org/.

[42] Dave Turner, Adam Oline, Xuehua Chen, and Troy Benjegerdes. Integrating New Capabilities into NetPIPE. In *Euro PVM/MPI*, Venice, Italy, 30 Sep 2003.

[43] J. A. Williams, I. Syed, J. Wu, and N. W. Bergmann. A Reconfigurable Cluster-on-Chip Architecture with MPI Communication Layer. In *FCCM '06: Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 350–352, Washington, DC, USA, 2006. IEEE Computer Society.

[44] Gregory V. Wilson. The History of the Development of Parallel Computing, Oct 1994. http://ei.cs.vt.edu/ history/Parallel.html.

[45] Xin Xie, J. Williams, and N. Bergmann. Asymmetric Multi-Processor Architecture for Reconfigurable System-on-Chip and Operating System Abstractions. *Field-Programmable Technology, 2007. ICFPT 2007. International Conference on*, pages 41–48, Dec. 2007.