PRODUCTIVELY SCALING HARDWARE DESIGNS OVER INCREASING RESOURCES USING A SYSTEMATIC DESIGN ANALYSIS APPROACH

by

Andrew Gregory Schmidt

A dissertation submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in
Electrical Engineering

Charlotte

2011

Approved by:

Dr. Ronald R. Sass

Dr. James M. Conrad

Dr. Bharat Joshi

Dr. Ryan Adams

Dr. Shannon Schlueter

ABSTRACT

ANDREW GREGORY SCHMIDT. Productively scaling hardware designs over increasing resources using a systematic design analysis approach.
(Under the direction of DR. RONALD R. SASS)

As processor development shifts from strict single core frequency scaling to heterogeneous resource scaling two important considerations require evaluation. First, how to design systems with an increasing amount of heterogeneous resources, and second, how to maintain a designer's productivity as the number of possible configurations grows. Therefore, it is necessary to determine what useful information can be gathered from existing designs to help predict or identify a design's potential scalability, as well as, identifying which routine tasks can be automated to improve a designer's productivity. Moreover, once this information is collected, how can this information be conveyed to the designer such that it can be used to increase overall productivity when implementing the design over increasing amounts of resources?

This research looks at various approaches to analyze designs and attempts to distribute an application efficiently across a heterogeneous cluster of computing resources through the use of a Systematic Design Analysis flow and an assortment of productivity tools. These tools provide the designer with projections on the amount of resources needed to scale an existing design to a specified performance, as well as, projecting the performance based on a specified amount of resources. This is accomplished through the combination of static HDL profiling, component synthesis resource utilization, and runtime performance monitoring. For evaluation, four case studies are presented to demonstrate the proposed flow's scalability on a small scale cluster of FPGAs. The results are highly favorable, providing orders of magnitude speedup with minimal intervention from the designer.

ACKNOWLEDGMENTS

I would like to thank everyone who has knowingly (or unknowingly) given me advice, support, encouragement, hope, laughter, and caffeine. Throughout it all I have been blessed with inspiring professors, brilliant peers, amazing friends, and a wonderfully loving family. I hope that someday I might be able to express my deepest gratitude for all that you have shared with me.

To the army of graduate students in the RCS lab (Will, Scott, Robin, Shanyuan, Siddhartha, Ashwin, Yamuna, Bin, Rahul, Shweta, and countless others) it has been a real pleasure to work with you, learn from you, and spend countless late nights together, waiting for systems to synthesize.

To my committee, I am grateful for the many hours you have taken to help advise me. From the words of encouragement while passing in the hallway to the detailed discussions regarding my research and my future plans, you have helped me stay focused, positive, and moving forward.

Certainly, none of this would have been possible without my advisor. Ron, even with a Cambrian Explosion of synonyms for the qualities you possess, I could not thank you enough or express my appreciation for all that you have done for me. I am honored that you are my advisor and I am thrilled to call you my friend.

Of course having such a supportive family helps too. Greg, Bonnie, Matt, Rachel, Chuck, Linda, and Tyler, family does not get much better than this! Without you all I would not have gotten started.

Hilary, you made this feel easy. Without you this would not have mattered.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

AIREN	Architecture Independent Reconfigurable Network
ALL	AIREN Data Link Layer
BBD	Black Box Description
BLAS	Basic Linear Algebra Subprograms
BLAST	Basic Local Alignment Sequence Tool
BRAM	Block Random Access Memory
BSB	Base System Builder
CIF	Context Interface
CIP	Create and Import Peripheral
DCI	Digitally Controlled Impedance
DCM	Digital Clock Mangers
DRAM	Dynamic Random Access Memory
DRC	Design Rule Check
DSP	Digital Signal Processing
EOF	End of Frame
FF	Flip-Flops
FPGA	Field Programmable Gate Array
FSC	FPGA Session Control
FSM	Finite State Machine
GUI	Graphical User Interface
HDL	Hardware Description Language
HPC	High Performance Computing
HWFS	Hardware Filesystem
ILA	Integrated Logic Analyzer
IPIC	Intellectual Property Interconnect
IPIF	Intellectual Property Interface

ISE	Integrated Software Environment
LUT	Lookup Table
MAcc	Multiply and Accumulate
MHS	Microprocessor Hardware Specification
MMM	Matrix-Matrix Multiplication
MMU	Memory Management Unit
MPD	Microprocessor Description
NCD	Native Circuit Description
NFS	Network Filesystem
NGD	Native Generic Database
NPI	Native Port Interface
PAO	Peripheral Analysis Order
PAR	Place and Route
PLB	Processor Local Bus
PSoC	Programmable System-on-Chip
RAID	Redundant Array of Independent Disks
RCC	Reconfigurable Computing Cluster
RSH	Remote Shell
SDAflow	Systematic Design Analysis Flow
SGEMM	Single Precision General Matrix-Matrix Multiplication
SRAM	Static Random Access Memory
SOF	Start of Frame
SRP	Synthesis Report
XCO	Xilinx CoreGen
XMP	Xilinx Microprocessor Project
XST	Xilinx Synthesis Tool

CHAPTER 1: INTRODUCTION

For decades programmers have relied upon frequency scaling to run the same computer applications faster with each generation of new processor. Unfortunately, this trend of ever faster clock frequencies has, for the most part, peaked and have begun to level off at around 2-3 GHz [1]. This is in part due to the increased power consumption and heat dissipation. As a result, industry has shifted away from strict single processor core frequency scaling in favor of resource scaling — increasing the number of compute (processor) cores on the chip with each generation [2].

While initially these chips consisted of homogeneous (dual core processors, quad core, etc.) processing elements, newer chips are being constructed with an assortment of processor cores, memory controllers, graphics controllers, discrete processing elements, and others [3, 4, 5, 6]. These heterogeneous chips with specialized compute cores could potentially offer significant performance improvements by offloading computation that is not well suited for a single sequential processor.

However, it remains unclear how to best assemble these compute cores on a single chip. Take a simple example of how to connect compute cores on a single chip. For a relatively small number of compute cores it is possible, in some cases, to connect them all together in a fully connected direct network (complete graph), as seen in Figure 1.1(a). As the number of compute cores increase, the interconnection resources increase (at the rate of $c = n(n-1)/2$, where n represents the number of nodes and c represents the number of connections). As the number of nodes increase, the amount of communication resources using the fully connected network becomes too costly and the propagation delay becomes too great. As a result, an alternative interconnect is required, such as a crossbar switch shown in Figure 1.1(b). The crossbar switch still

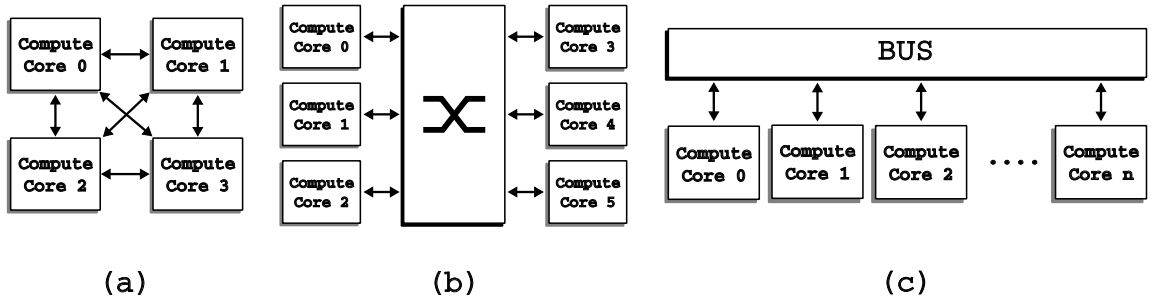


Figure 1.1: block diagram of different ways to connect compute cores: (a) direct connect, (b) crossbar switch, and (c) a shared bus

allows every core to communicate with every other core, but not all at the same time in (the number of interconnects depends on the implementation). A third configuration of these compute cores could be a shared bus, Figure 1.1(c), which limits the amount of parallel communication between cores, but minimizes the amount of interconnection resources.

Moreover, as the number of different types of compute cores increase, so does the complexity associated with best placing them on a chip. This placement can greatly effect the performance of a system, especially in the area of High Performance Computing (HPC). Here, every last bit of processing potential is eked out by programmers in order to maximize performance and surpass computing barriers such as the recently broken PetaFLOP barrier.

As another example, consider a design that includes a single compute core as an application accelerator. The designer can optimize the core to best utilize all of the available resources to yield the greatest performance. However, this is not always as simple as it sounds. When just considering access to off-chip memory, the designer must consider the bandwidth and latency requirements in order to determine whether the core should rely on the processor to initiate the data transfer (shown in Figure 1.2(a)), or if the core should initiate the data transfer. If the core is to initiate the transfer, should it be connected to a shared bus as a bus master, (Figure 1.2(b)), or should it be directly connected to the memory controller (Figure 1.2(c))?

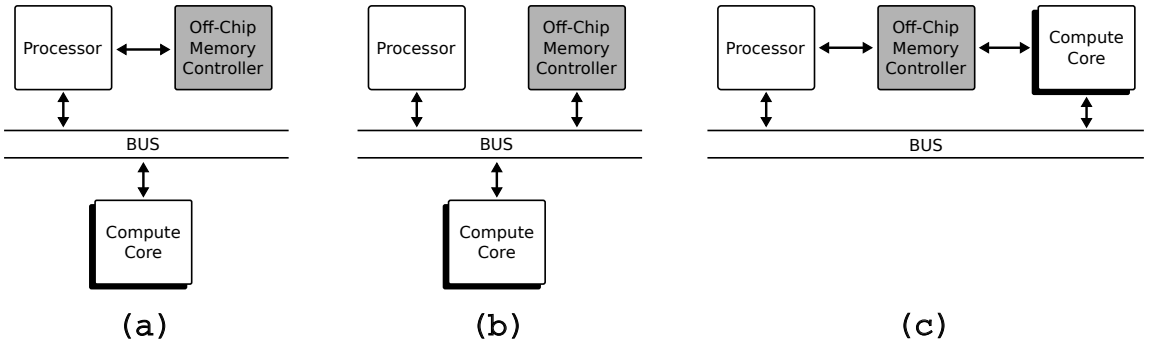


Figure 1.2: block diagram of (a) one compute core connected to a shared bus and acting as a slave, receiving data from the processor which is directly connected to the memory controller, (b) one compute core connected to a shared bus and acting as a bus master, and (c) one compute core directly connected to the memory controller

These issues are easy to visualize, but efficient and practical design decisions have to be addressed. One important issue is how to partition on-chip resources. In a general purpose processor there is a trade-off between the size of the caches and number of processor cores. While it may be a difficult decision, with the advancements in fabrication, multi-processor core designs are not drastically sacrificing cache sizes. On the other hand, in more heterogeneous designs the trade-off becomes an even more concerning problem as it is difficult to compare the significance of one resource over another. Making a trade-off between adding an additional memory controller compared to particular a discrete processing element is application specific, and the designer of the chip has to make a decision that may result impact sales.

Another issue is with respect to a design where more resources have become available (whether due to using more chips, migrating to a larger chip, or because the core only initially consumed a portion of the available resources). Now the designer has to go through an even more complex problem to try and scale the number of cores. Often the simple solution is to replicate n number of these cores, using all of the resources available. Yet, without careful consideration a thoughtful design may actually degrade system performance. This is the old adage, "more isn't always better."

There may be little effect on the performance by changing how a single core is

connected, to say off-chip memory; however, when adding more cores, the contention for the processor, shared bus, or the memory controller itself can degrade the overall performance. The processor may be able to sustain a single or even a few compute cores, but at some point the processor may be overwhelmed trying to service too many cores. This is also true with bus contention [7]. As the number of cores that are connected to the same shared bus increase, each must wait for its turn to access the bus. This increases the latency between issuing requests and receiving data. Finally, even if there were enough resources to connect these cores directly to the memory controller, it becomes the bottleneck trying to service an increasing number of parallel requests to memory.

Figure 1.3 shows two examples of the performance gained by scaling a system from a single cores to 16 cores. The first configuration was quickly assembled and resulted in a poorly implemented design with performance that does provide a speedup, but only does so at a very modest 2.25× speedup. Now compared to the second configuration, a carefully thought out design, the speedup is significantly improved and actually tracks slightly better than linear speedup. While not every design may scale so fortuitously, the implementation can play a vital role in how the system scales. Furthermore, both designs peak prior to fully utilizing the available resources at 16 cores. This again illustrates the point that even if it is possible to put more cores in the design, it may not provide an increase in performance.

1.1 Field Programmable Gate Arrays

The work presented here focuses on the use of multiple Field Programmable Gate Arrays (FPGAs), where resources are more diverse and the scaling potential is even more evident. FPGAs now have the ability to assemble a programmable system-on-chip (PSoC) that includes processors, memory controllers, system buses and on-chip peripherals. And with the programmable logic resources increasing with every generation [8], still tracking Moore's law (the number of transistors will double ap-

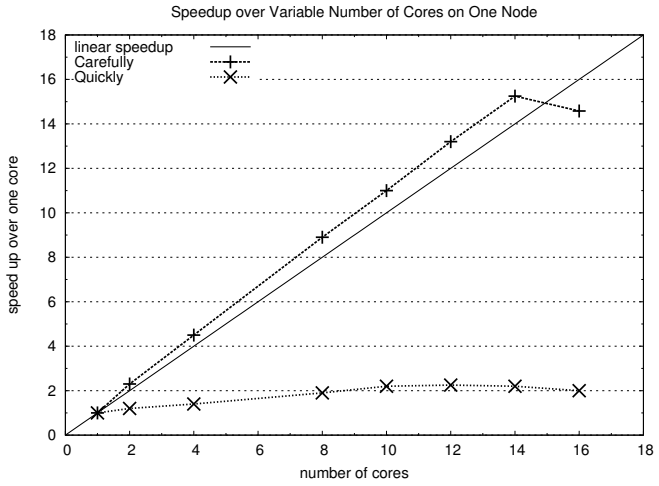


Figure 1.3: sample speedup performance comparison of two designs

proximately every two years [9]), it is possible to experiment with the scalability and connectivity of designs (please see Chapter 2 for a more detailed background on FPGAs).

FPGAs enable this research to investigate real applications running in hardware with actual data sets as opposed to a simulated analysis. What this means is that the entire system is part of the evaluation. Often times simulation can limit the scope due to increased evaluation times that are necessary for a sequential processor to model the behavior of a parallel system. Simulation also must predict performance of non-deterministic components such as access to off-chip memory. Experimenting with FPGAs provides tangible results of fully running systems with complex hardware and software configurations. Moreover, running these experiments is crucial to understanding the systems behavior because static analysis of the performance might not be accurate when an application's data sets can play a significant role in the overall performance. That is to say, run under one set of inputs the application may demonstrate tremendous speedups, yet under another it may perform dismally.

The main focus of this research looks at various approaches to analyze and attempts to distribute an application efficiently across a heterogeneous cluster of com-

puting resources. The process will also provide the designer with projections on the amount of resources needed to scale an existing design to a specified performance, as well as, projecting the performance based on a specified amount of resources. More specifically, this work will investigate the feasibility to productively scale existing hardware designs to a large number of FPGA resources in an all-FPGA cluster.

1.2 Productivity Tools

To aid in the development of a system, a series of development tools are presented that operate in collaboration with the existing synthesize and analysis tools that designers are accustom to using. Chapter 2 and Chapter 3 provide a more detailed description of these existing tools, whereas Chapter 4 discusses the specific tools developed as part of this work. This section briefly highlights the type of tools that have been developed as part of this work to help improve a designer's overall productivity.

Traditionally a designer relies upon the exiting tools to assemble designs, but often must make significant modifications to the design manually. During the development stage a designer may need to known the resource utilization of a component in the design. To identify the utilization the designer must manually create a project, synthesize the design, and read the synthesis report. While tools and graphical user interfaces (GUIs), such as Xilinx ISE [10], exist to help, the designer is still tasked with not only the complete assembly of the project, but also the analysis of the synthesis results to intelligently create efficiently performing systems. What is needed are tools that can quickly assemble these project autonomously, perform routine tasks such as synthesis, parse and analyze results,and present the designer with recommendations on how to scale the design and/or connect the design for increased performance.

Scaling a design can also mean migrating a design from one device to another. A designer does not have the luxury of simply resynthesizing/recompiling the design to take advantage of the increase in resources that have been so favorable for software

developers during the hey days of processor frequency scaling. This is due to the fact that during synthesis the HDL source is used to generate the logic configuration for the specific device. Without changing the design, even if the new device has twice the amount of resources, the HDL specification will still produce the same design. Instead the system must be redesigned. Even in the trivial case of scaling the number of components to utilize the now larger device requires significant design effort. To assist the designer, migration tools can be used to port a design to the new device along with replication tools to utilize the increase in available resources.

Often times a system does not perform better by simply increasing the number of components operating in parallel, perhaps due to other bottlenecks present in the system such as buses or accessing memory. As a result, the designer may spend a significant amount of effort augmenting a design to combat the bottleneck. Instead, tools to parse the existing design to identify the interfaces between the components in the design could be used to then assemble alternative configurations utilizing different types of interconnections.

Regardless of the scalability or the performance due to how a system is configured, a designer often evaluates the performance of individual components in a system. For example, a designer may wish to determine the utilization of a particular resource. Doing so requires the designer to manually create and insert specific monitors and add them to their design. Moreover, the designer must also provide a mechanism to retrieve the data from these monitors, often at the expense of other resources (e.g. ChipScope [11, 11]) or runtime performance when the processor must stop its calculations to retrieve and store the data for the designer. Therefore it would be advantageous for a tool or set of tools to exist to not only identify what should be monitored in a system, but to insert the necessary infrastructure to enable the monitoring to be minimally invasive.

These types of tools can help improve a designer's productivity by reducing the

amount of unnecessary or often redundant work that comes from hardware design. The goal of these tools is not to fully replace the designer, but instead to allow the designer to focus on the development of the critical components of the system. Furthermore, these tools can improve the lifetime of the developed system by enabling the system to migrate to newer devices without significant effort by the original designer. The rest of this work is aimed at not only the development of such tools, but the evaluation of their effectiveness to improve the productivity during a system's design.

1.3 Thesis Statement

As we shift from frequency scaling to resource scaling we need to consider how to not only design for such systems, but to maintain a designer's productivity by minimizing the design search space as the number of possible configurations grows. We need to determine what routine tasks can be automated to further improve a designer's productivity. We need to understand what useful information can be gathered from existing designs to help predict or identify scalability and performance potential of future designs. We must also address how can this information be conveyed to the designer such that it can be used to increase overall productivity when implementing future designs on an increasing amount of resources.

Furthermore, with so many configuration choices, there needs to be a means to compare each configuration. But how or what should be compared? Should the comparison focus on computation rate, throughput, resource utilization or some other metric? And even if such a metric for comparison exists, how can the performance be describe such that a designer can quickly understand how to efficiently (and productively) utilize the system?

Thus, the ultimate question we are trying to address is: *can the knowledge of an experienced hardware designer be codified into a design flow and a set of tools?* If so, this will make system designers more productive.

In this work we propose to answer all of these questions through a Systematic Design Analysis flow (SDAflow) that includes the use of static hardware profiling, timing and resource profiling in concert with runtime performance monitoring to create a model of performance and a set of tools for spatial scaling that will aim to increase designer productivity.

The remainder of this dissertation is organized as follows. Chapter 2 presents the necessary background information the reader should be familiar with in order to understand the dissertation work. This is followed by Chapter 3 which covers related work found in both academia and industry. Next, Chapter 4 covers the design and implementation of the Systematic Design Analysis flow (SDAflow) along with the productivity tools created for this work. In Chapter 5 the experimental setup and evaluation methodology for the design flow and tools is presented. Chapter 6 follows with four case studies used to evaluate the Systematic Design Analysis flow and the effectiveness of the productivity tools. Chapter 7 concludes with a brief summary of the research.

CHAPTER 2: BACKGROUND

This chapter begins with an overview of Field-Programmable Gate Arrays (FPGAs) and the components that can be found on modern FPGAs in Section 2.1. For those more familiar with FPGAs reading this section may not be necessary, but it does provide a good overview and review for the less initiated. Section 2.2 briefly covers hardware descriptions languages (HDLs). It is not necessary to be proficient with a specific HDL, but it might prove useful to be familiar with the terms and the capabilities of HDLs as some of the design and analysis includes discussions and code examples.

In Section 2.3 the Xilinx tool chain is presented. This tool chain is used extensively throughout this work and it may be useful to refer back to this section in later chapters. The basic tool flow is covered with a description of each basic step from synthesizing a hardware description language design to the generation a FPGA configuration file. In Section 2.4 the Reconfigurable Computing Cluster (RCC) project currently under evaluation at the University of North Carolina at Charlotte is presented. A significant portion of the research discussed here will be accomplished with hardware and software that are part of the RCC project. Finally, in Section 2.5, the custom high-speed network that is used as part of the RCC project is discussed in thorough detail. The detail is presented in this chapter as background to help explain the capabilities of this custom network and how its integration into the FPGA cluster enables research to study scalability of designs beyond a single node in the cluster.

2.1 Field-Programmable Gate Array

A modern FPGA consists of a 2-D array of programmable logic blocks, fixed-function blocks, and routing resources implemented in the CMOS technology. Along

the perimeter of the FPGA there are special logic blocks that are connected to external package I/O pins. Logic blocks consist of multiple logic cells, while logic cells contain function generators and storage elements. These general terms will be discussed in more detail throughout this section.

2.1.1 Function Generators

FPGA devices use *function generators* to implement Boolean logic functionality rather than physical gates. For example, to implement the Boolean function:

$$f(x, y, z) = xy + z'$$

using a 3-input function generator, first create the eight row Boolean truth table for this function. For each input the truth table represents what the Boolean function's output will be. If each of the function's output bits were stored into individual static memory (such as SRAM) cells and connected as inputs to an 8x1 multiplexer (MUX), the three inputs (x,y,z) would be the select lines for the MUX. The result is commonly what is known as a *look-up table* (LUT).

It is also important to understand that unlike a digital circuit implemented within logic gates, the propagation delay from a single LUT is fixed. This means, regardless of the complexity of the Boolean circuit, if it fits within a single LUT, the propagation delay remains the same. This is also true for circuits spanning multiple LUTs, but instead, the delay depends on the number of LUTs and additional circuitry necessary to implement the larger function.

To generalize, the basic n-input function generator consists of a 2^n-to-1 multiplexer (MUX) and 2^n SRAM (static random access memory) cells. By convention, a 3-LUT is a 3-input function generator. The 3-input structure is mentioned for demonstration purposes, although 4-LUT and 6-LUTs are more common in today's components, such as the Xilinx Virtex 4 and 5 FPGAs.

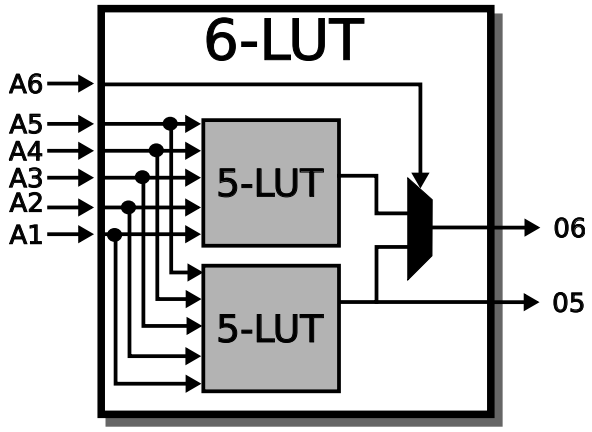


Figure 2.1: Virtex 5's six input LUT is built from two five input LUTs with the sixth input controlling a MUX between the two LUTs

To implement a function with more inputs than would fit in a single LUT, multiple LUTs are used. The function can be decomposed into sub-functions of a subset of the inputs, each sub-function is assigned to one LUT, and all of the LUTs are combined (via routing resources) to form the whole function. There are also some dedicated routing resources to connect neighboring LUTs with minimal delay to support low propagation delays.

In practice, using HDL to describe the digital circuit and then using synthesis tools to map the textual description to an equivalent look-up function is more common than the designer defining the LUTs logic itself. What is important as far as the designer is concerned is how to represent a circuit to efficiently utilize the available resources. The 6-LUT on the Virtex 5 can either be used as a single 6-LUT or as two 5-LUTs as long as both 5-LUTs share the same inputs. A designer can take advantage of this when building digital circuits by not including the unnecessary inputs which the synthesis tools may infer to a larger LUT. Figure 2.1 represents the block diagram of a 6-LUT. In the event that all 6 inputs are used for the LUT, the bottom output `O5` is not used.

An important observation is that SRAM cells are volatile, if power is removed the value is lost. As a result, we need to learn how to set the SRAM cell's value.

This process, called *configuring* (or programming), could be handled by creating an address decoder and sequentially writing the desired values into each cell. However, the number of SRAM cells in a modern FPGA is enormous and random access is rarely required. Instead, the configuration data is streamed in bit-by-bit. The SRAM cells are chained together such that, in program mode, the data out line of one SRAM cell is connected to the data in line of another SRAM cell. If there are n cells, then the configuration is shifted into place after n cycles. Some FPGA devices also support wider, byte-by-byte, transfers as well to support parallel transfers for faster programming.

2.1.2 Storage Elements

While the function generators provide the fundamental building block for combinational circuits, there are additional components within the FPGA to provide a wealth of functionality. As is common with other programmable logic devices, the D-type flip-flops are incorporated in the FPGA. The flip-flops can be used in a variety of ways, the simplest being data storage. Typically, the output of the function generator is connected to the flip-flop's input. Also, the flip-flop can be configured as a latch, operating on the clock's positive or negative level. When designing with FPGAs, it is suggested to configure the storage elements to be D flip-flops instead of latches. A latch being level-sensitive to the clock (or enable) increases the difficulty to route clock signals within a specific timing requirement. For designs with tight timing constraints, such as operating custom circuits at a high operating frequency that span large portions of the FPGA, D flip-flops are more likely to meet the timing constraints.

2.1.3 Logic Cells

By combining a function generator and a storage element the result is commonly what is referred to as a *logic cell*. Logic cells are really the low-level building block upon which FPGA designs are built. Both combinational and sequential logic can

be built from within a logic cell or a collection of logic cells. Many FPGA vendors will compare the capacity of an FPGA based on the number of logic cells (along with other resources as well). In fact, when comparing designs, it is no longer relevant to describe an FPGA circuit in terms of "number of equivalent gates (or transistors)". This is because a single LUT can represent very modest equations which would only require a few transistors to implement, or very complex circuit such as a RAM which would require many hundreds of transistors. While the process of mapping larger circuits for logic cells has not been described yet, it is possible to identify based on the number of logic cells how big or small a design is and whether or not it will fit within a given FPGA chip.

The Xilinx Virtex 5 combines four of these logic cells to create a *slice*, as is shown in a simple block diagram in Figure 2.2. With four 6-LUTs and D flip-flops contained within close proximity, it is possible to use these components to design more complex circuits. In addition to Boolean logic, a slice can be used for arithmetic and RAMs/ROMs. Some slices are connected in such a way that they can be used for data storage as distributed RAMs or shift registers. This is accomplished by combining multiple LUTs in the slice. The distributed RAM can be configured as single, dual and in some cases quad ports providing independent read and write access to the RAM. The depth of the RAMs vary based on the number of ports, but can range from 32 to 256 1-bit elements. The distributed RAMs data width can be increased beyond 1-bit; however, there will be a trade off between the width and depth and resource usage. For example, a 64x8 (64 8-bit elements) RAM is implemented in nine LUTs (1-LUT per 64x1 RAM and an additional LUT for logic). However, a 64x32 extends beyond an efficient use of the configurable resources and is moved into what will be discussed shortly, Block RAM (BRAM).

In addition to logic and memory, slices can be used as shift registers. A shift register is capable of delaying an input x number of clock cycles. Using a single LUT,

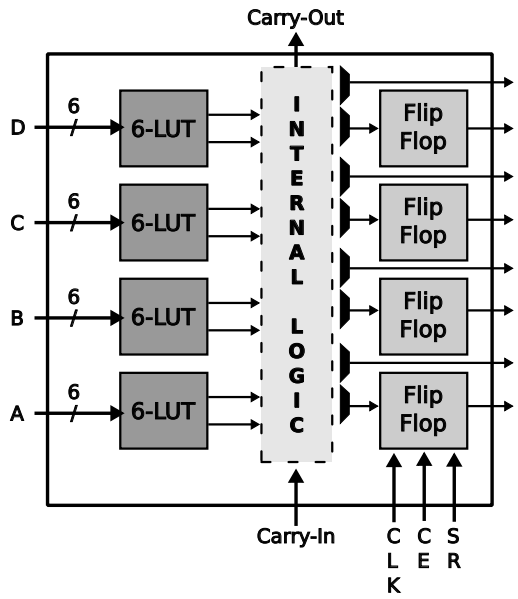


Figure 2.2: Virtex 5's logic slice block diagram with four 6-LUTs and four flip-flop/latch storage elements

data can be delayed up to 32 clock cycles. Cascading all four LUTs in one slice the delay can increase to 128 clock cycles. This is useful for small buffers that would traditionally be implemented within a more valuable resource such as a Block RAM.

With all of these possible uses, a D flip-flop can be added to provide a synchronous read operation. With the additional D flip-flop, a read will be subject to an additional latency of one clock cycle. This may or may not impact a design, but for designs with high timing constraints, adding the synchronous operation can relax the constraint.

2.1.4 Logic Blocks

While logic cells could be considered the basic building blocks for FPGA designs, in actuality it is more common to group several logic cells (slices) into a block and add special-purpose circuitry, such as an adder/subtractor carry chain, into what is known as a *logic block*. This allows a group of logic cells that are geographically close to have quick communication paths, reducing propagation delays and improving design implementations. For example, the Xilinx Virtex 5 families put four logic cells in a *slice*. Two slices, and carry-logic form a *Configurable Logic Block* or CLB.

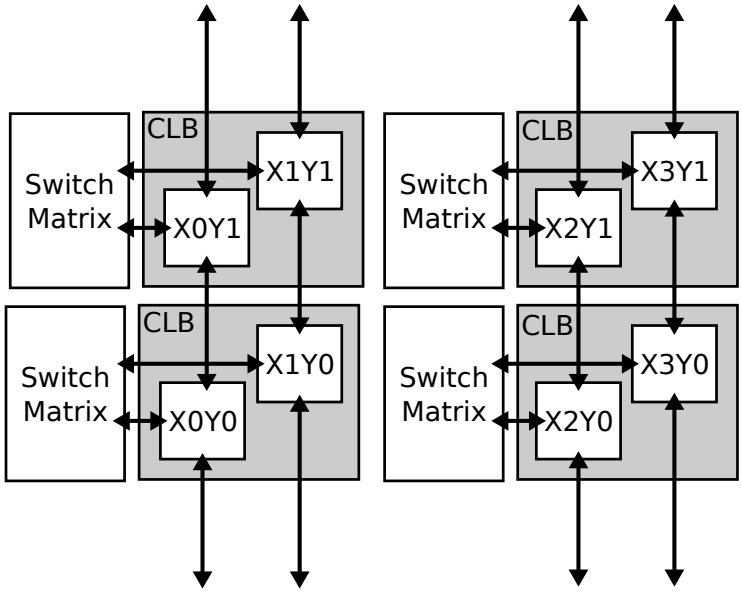


Figure 2.3: Virtex 5 CLB block diagram of the interconnection between each slice in the CLBs and their connection to the switch matrix

Figure 2.3 depicts the basic structure and interconnection of the CLBs, slices and switch matrices for the Virtex 5.

Abstractly speaking, the logic blocks are what someone would see if they were to "look into" an FPGA. The exact number of logic cells and other circuitry found within a logic block is vendor specific; however, we are now able to realize even larger digital circuits within the FPGA fabric.

Logic blocks are connected by a routing network to provide support for more complex circuits in the FPGA fabric. The routing network consists of switch boxes. A *switch box*, is used to route between the inputs/outputs of a logic block to the general on-chip routing network. The switch box is also responsible for passing signals from wire segment to wire segment. The wire segments can be short (span a couple of logic blocks) or long (run the length of the chip). Since circuits often span multiple logic blocks, the carry chain allows direct connectivity between neighboring logic blocks, bypassing the routing networking for potentially faster implemented circuits.

Competing vendors and devices often have different routing networks, different

special-purpose circuitry, and different size function generators. So it is difficult to come up with an exact relationship for comparison. The comparison is further complicated since it also depends on the actual circuit that is being implemented.

2.1.5 Special-Purpose Function Blocks

So far the focus has been on the internals of the FPGA's configurable (or programmable) logic. A large portion of the FPGA consists of logic blocks and the routing logic to connect the programmable logic. However, as semiconductor technology advanced and more transistors became available, FPGA vendors recognized that they could embed more than just configurable logic to each device.

Platform FPGAs combine the programmable logic with additional resources that are embedded into the fabric of the FPGA. A good question to ask at this point is "why embedded specific resources into the FPGA fabric?" To answer this question consider FPGA designs compared to ASIC designs. An equivalent ASIC design is commonly considered to use fewer resources and consume less power than an FPGA implementation. However, ASIC designs are not only often found to be prohibitively expensive, the resources are fixed at fabrication. Therefore FPGA vendors have found a compromise with including some ASIC components among the configurable logic.

The block diagram of a Platform FPGA, seen in Figure 2.4, shows the arrangement of these special-purpose function blocks placed throughout the FPGA. Which function blocks are included and their specific placement is determined by the physical device, this illustration is meant to help better understand the general construct of modern FPGAs. The logic blocks still occupy a majority of the FPGA fabric in order to support a variety of complex digital designs; however, the move to support special-purpose blocks provides the designer with an ASIC implementation of the block as well as removes the need to create a custom design for the block. For example, a processor block could occupy a significant portion of the FPGA if implemented in the logic resources.

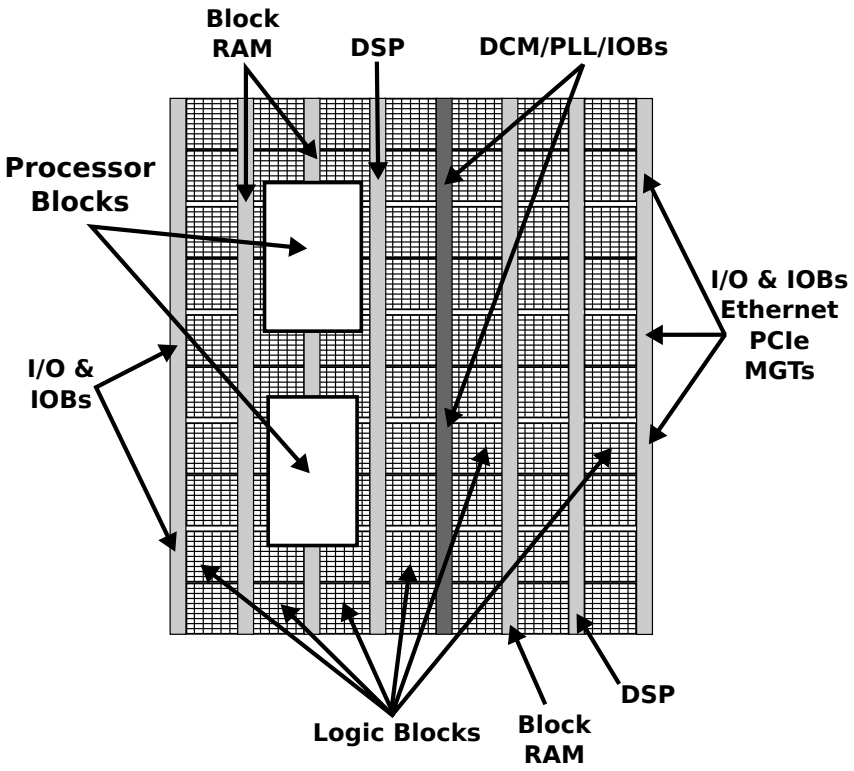


Figure 2.4: high-level view of a Platform FPGA

2.1.5.1 Processors

Arguably one of the more significant additions to the FPGA fabric is a processor embedded within the FPGA fabric. For many designs requiring a processor, often choosing an FPGA device with an embedded processor (such as the FX series part for the Xilinx Virtex 5) can greatly simplify the design process while reducing resource usage and power consumption. The IBM PowerPC 405 and 440 processors are examples of two processors included in the Xilinx Virtex 4 and 5 FX FPGAs respectively. The PowerPC 440's block diagram is shown in Figure 2.5.

These are conventional RISC processors which implement the PowerPC instruction set. Both the PowerPC 405 and 440 come with some embedded system extensions, but do not implement floating-point function units in hardware. Each come with level 1 instruction and data cache and a memory management unit (MMU) with translation look-aside buffer to support virtual memory. A variety of interfaces exist

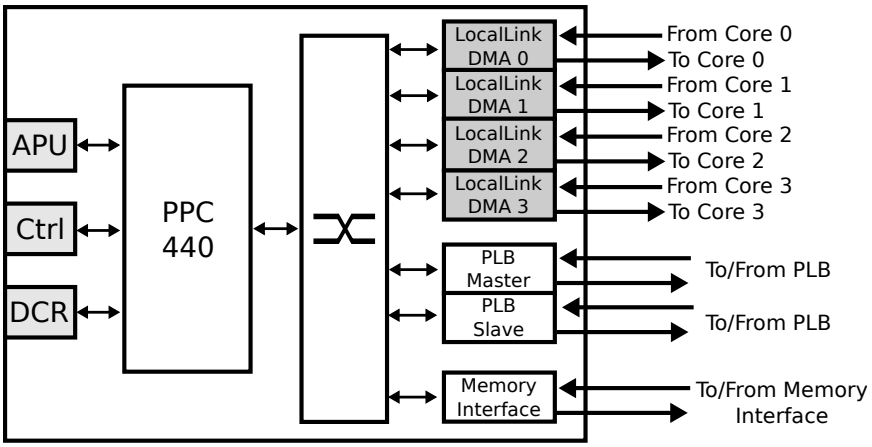


Figure 2.5: PowerPC 440 processor block diagram showing the available interfaces between processor and the FPGA fabric

to connect the processors to the FPGA programmable logic to allow interaction with custom hardware cores.

Not all FPGAs come with a processor embedded into the FPGA fabric. For these devices the processor must be implemented within the FPGA fabric as a soft processor core. These range from small, simple purpose processors such as the Xilinx PicoBlaze which can be useful for implementing more complex state machines in Assembly rather than a hardware description language. A MicroBlaze is the bigger brother of the PicoBlaze, occupying approximately 1,500 slices (an order of magnitude more than the PicoBlaze) but can support a full running operating system such as Linux. Still more complex processors can be included, the Sun UltraSparc for example has been implemented on an FPGA; however, it consumes a significant amount of resources.

2.1.5.2 Block RAM

Many designs require the use of some amount of on-chip memory. Using logic cells it is possible to build variable sized memory elements; however, as the amount of memory needed increases, these resources are quickly consumed. The solution, to provide a fixed amount of on-chip memory embedded into the FPGA fabric called *Block RAM* (BRAM). The amount of memory depends on the device; for example,

the Xilinx Virtex 5 XC5VFX130T (on the ML-510 development board) contains 298 36 Kb BRAMs, for a total storage capacity of 10,728 Kb. Local on-chip storage such as RAMs and ROMs or buffers can be constructed from BRAMs. BRAMs can be combined together to form larger (both in terms of data width and depth) BRAMs. BRAMs are also dual-ported, allowing for independent reads and writes from each port, including independent clocks. This is especially useful as a simple clock crossing device, allowing one component to produce (write) data at a different frequency as another component consuming (reading) the data.

One common uses of BRAMs in FPGA designs is for FIFOs. FIFOs, or simply data queues, are primitives the designer can take advantage of, rather than building their own out of BRAM logic, reducing design and debugging time. Recently, FPGAs have started to include FIFOs as separate components within the FPGA fabric. The Virtex 5 and 6 are two such devices, although the physical limitations on the functionality may rule out their use in a design.

2.1.5.3 Digital Signal Processing Blocks

To allow more complex designs which may consist of either digital signal processing or just some assortment of multiplication, addition and subtraction, *Digital Signal Processing Blocks* (DSP) have been added to many FPGA devices. As with the Block RAM, it is possible to implement these components within the configurable logic, yet it is more efficient in terms of area, performance, and power consumption to embed multiple of these components within the FPGA fabric. At a high level, the DSP blocks of a multiplier, accumulator, adder, and bitwise logical operations (such as AND, OR, NOT, NAND, etc.) It is possible to combine DSP blocks to perform larger operations such as single and double precision floating point addition, subtraction, multiplication, division and square-root. The number of DSP blocks is device dependent; however, they are typically located near the BRAMs which is useful when implementing processing requiring input and/or output buffers.

In the Virtex 5, the DSP slices are known as DSP48E (48-bit DSP element) slices. The DSP slices include a 25x18 two's complement multiplier, 48-bit accumulator (for multiply accumulate operations), an adder/subtractor for pipelined operations and bitwise logical operations. Embedding this functionality into the slice provides a significant savings in FPGA resources since implementing the equivalent resources in LUTs is quite expensive.

For applications needing filters, such as comb and finite impulse response, to transforms, such as fast and discrete Fourier, to CORDIC (coordinate rotational digital computer) algorithm, the DSP slices are used when available. The Virtex 5 FX130T on the ML-510 contains 320 DSP slices. Compared to the 20,480 regular slices, this seems like disproportionate amount, yet not all designs require the use of DSP slices. For designs requiring a higher percentage of DSP slices, the Xilinx `SX` series FPGAs include more DSP resources. There are tools, one of which we will introduce shortly, which help the designer quickly implement customized DSP components. They are also useful for resource and performance approximation.

2.1.5.4 Select I/O

Interfacing off the FPGA is another important issue when designing for embedded systems. In most cases there will be a need to interface with some physical device(s). Depending on the number of I/O pins required, some devices are better suited than others, but they are all built around *Input/Output Blocks* (IOB).

Now, let's augment the logic block array with IOBs that are on the perimeter of the chip. These IOBs connect the logic block array and routing resources to the external pins on the device. Each IOB can be used to implement various single-end signaling standards, such as LVCMOS(2.5 V) and LVTTL (3.3 V) and PCI (3.3 V). IOBs can also support double data rate signaling used by commodity static and dynamic random access memory. The IOBs can be paired with adjacent IOBs for differential signaling, such as LVDS.

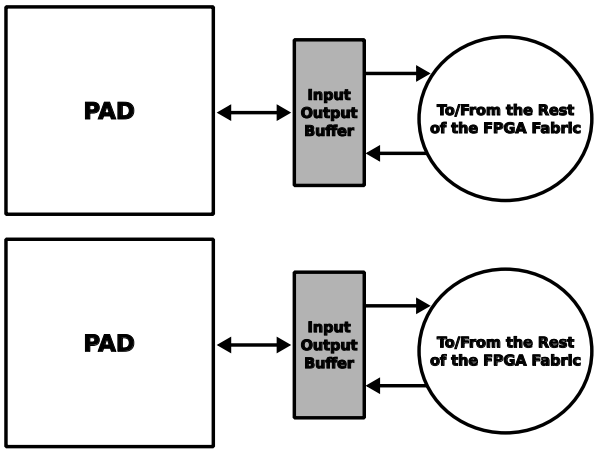


Figure 2.6: Xilinx Select IOB tile, for connections between FPGA fabric and the pad on the FPGA device

From Figure 2.6 each I/O tile spans two pads (which connect to physical pins). Each Pad connects to a single IOB which connects to the input and output logic. Xilinx uses the term *Select I/O* to refer to configurable inputs and outputs which support a variety of standard interfaces (LVCMOS, SSTL, LVDS, etc.) Select I/O also take advantage of Digitally Controlled Impedance (DCI) to eliminate adding resistors close to the device pins which are needed to avoid signal degradation. DCI can adjust the input or output impedance to match the driving or receiving trace impedance. Some advantages include the reduction in the number parts which simplifies the PCB routing effort. It also provides a way to correct for variations in manufacturing, temperatures, and voltages.

2.1.5.5 Multi-Gigabit Transceivers

Over the last twenty years, digital I/O standards have varied between serial and parallel interfaces. Serial interfaces time-multiplex the bits of a data word over a fewer conductors while parallel interfaces signal all the bits simultaneously. While the parallel approach has the apparent advantage of being faster (data are being transmitted in parallel), this is not always the case in the presence of noise. Many recent interfaces — including various switched Ethernet standards, Universal Serial

Bus [12, 13], SerialATA [14], FireWire [15], InfiniBand [16] — are now using low-voltage differential pairs of conductors. These standards use serial transmission and change the way data is signaled to make the communication less sensitive to electro-magnetic noise.

High Speed Serial Transceivers are devices that serialize and deserialize parallel data over a serial channel. On the serial side, they are capable of baud rates from 100 Mb/s to 11.0 Gb/s which means that they can be configured to support a number of different standards, including Fiber Channel, 10G Fiber Channel, Gigabit Ethernet, and InfiniBand. As with other the aforementioned FPGA blocks, the transceivers can be configured to work together. For example, two transceivers can be used to effectively double the bandwidth. This is called channel bonding (multi-lane and trunking are common synonyms).

In the Virtex 5 series FPGAs two types of transceivers exist, RocketIO GTX and GTP. GTX transceivers are capable of a higher bandwidth whereas the GTP transceivers are lower bandwidth and require less power. The number of transceivers vary from part to part. For example, the Virtex 5 FX130T includes 20 GTX transceivers. As with the DSP slices and `SX` series FPGAs, there are applications which require a higher percentage of transceivers to configurable logic. The `TX` (also known as TXT or HX) series FPGAs include these additional transceivers. Both the GTX and GTP transceivers are bi-directional, providing independent transmit and receive at the same time.

Xilinx includes a customizable GTX/GTP wizard to expedite adding the transceiver logic to a design or specific component. In short, it is possible to specify the data width (parallel data), frequency and channel bonding needed by the design. Design considerations are needed for systems with tight timing or resource constraints; however, a lot of the headache typically associated with high-speed integration can be eliminated.

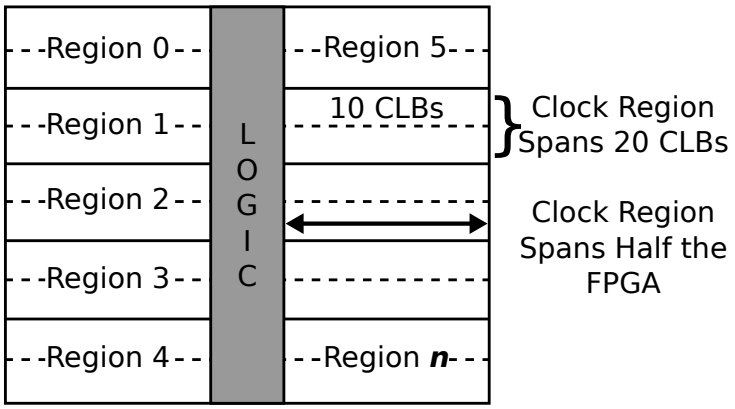


Figure 2.7: clock regions on an FPGA span 20 CLBs vertically and half of the FPGA horizontally, the number of clock regions varies by chip

2.1.5.6 Digital Clock Manager

In FPGA designs it is common to operate different cores at different frequencies. In traditional designs, any clocks needed would have to be generated off-chip and connected as an input to the system. With FPGA designs it is possible to generate a wide range of clock rates from a single (or a few) clock source(s). While it is easier to design systems with only a few different clock rates, having the flexibility to incorporate clock rates after board fabrication is compelling to designers. However, there are limitations on the number of clocks that can be generated and routed to various parts of the FPGA. A number of clock regions exist on the FPGA (varying from 8 to 24 based on the Virtex 5 FPGAs) which support up to 10 clocks domains. The FPGA is split in half and on each half a clock region spans twenty CLB. The Virtex 5 FX130T has 20 clock regions. Figure 2.7 is simple example of the clock regions on the FPGA.

To help the designer use and manage these clocks, Xilinx uses *digital clock managers* (DCM). Generally speaking, a DCM takes an input clock and can generate a customizable output clock. By specifying the multiply and divider values the frequency-synthesis output clock `clkfx` can generate a custom clock. Given an input clock `clkin` the equation:

$$clkfx = M/D * clkin$$

is used to generate the output clock. However, the DCM provides more than just generating different clock rates. A DCM is also capable of phase shifting the input clock by 90, 128, and 270 degrees. The DCM also provides a 2× the input clock rate and can phase shift this input clock by 180 degrees. It is easy to see without going into more detail that DCMs are a useful tool to generate the necessary clock(s) in FPGA designs. In short, meeting timing when designing with FPGAs can be a difficult task unless design considerations for timing and clocks are included from the beginning of the design process.

2.2 Hardware Description Languages

Now that the internals of an FPGA have been presented, the next step is to discuss how to "configure" them. This can be accomplished through the use of a *hardware description language* as a high-level language to describe the circuit to be implemented on the FPGA. The origins of hardware description languages were rooted in the need to document the behavior of hardware. Over time, it was recognized that the descriptions could be used to simulate hardware circuits on a general-purpose processor. This process of translating an HDL source into a form suitable for a general-purpose processor to mimic the hardware described is called *simulation*. Simulation has proved to be an extremely useful tool for developing hardware and verifying the functionality before physically manufacturing the hardware. It was only later that people began to *synthesize* hardware, automatically generating the logic configuration for the specified device from the hardware description language.

Unfortunately, while simulation provided a rich set of constructs to help the designer test and analyze the design, many of these constructs extend beyond what is physically implementable within hardware (on the FPGA) or synthesize inefficiently into the FPGA resources. As a result only a subset of hardware description languages can be used to synthesize designs to hardware. The objective of this section

is to present two of the more popular hardware description languages, VHDL and Verilog, along with a brief overview of some high-level languages to produce digital circuits.

2.2.1 VHDL

VHDL, which stands for VHSIC[1] Hardware Description Language, to describe digital circuits. In simulation, the VHDL source files are analyzed and a description of the behavior is expressed in the form of a netlist. A *netlist* is a computer representation of the a collection of logic units and how they are to be connected. The logic units are typically AND/OR/NOT gates or some set of primitives that makes sense for the target (4-LUTs, for example). The behavior of the circuit is exercised by providing a sequence of inputs. The inputs, call *test vectors*, can be created manually or by writing a program/script that generates them. The component that is generating test vectors and driving the device under test is typically called a *test bench*.

In VHDL, there are two major styles or forms of writing hardware descriptions. Both styles are valid VHDL codes; however they model hardware differently. This impacts synthesis, simulation, and, in some cases, designer productivity. These forms are:

Structural/Dataflow Circuits are described in terms of logic units and signals. Dataflow is a type of structural descriptions that has syntactic support to make it easier to express Boolean logic.

Behavioral Circuits are described in an imperative (procedural) language to describe how the outputs are related to the inputs as a process.

A third style exists as a mix between both structural and behavioral styles. For programmers familiar with sequential processors, the behavioral form of VHDL seems natural. In this style, the process being described is evaluated by 'executing the program' in the process block. For this reason, often complex hardware can be expressed

[1] Very high speed integrated circuit

succinctly and quickly — increasing productivity. It also has the benefit that simulations of certain hardware designs are much faster because the process block can be directly executed. However, as the design becomes more complex, it is possible to write behavioral descriptions that cannot be synthesized. Converting behavioral designs into netlists of logic units is called *High-Level Synthesis* referring to the fact that behavioral VHDL is more abstract (or higher) than structural style.

In contrast, since the structural/dataflow style describes logic units with known implementations, these VHDL codes almost always synthesize. Also, assembling large systems (such as Platform FPGAs) requires structural style at the top-level since it is combining large function units (processors, peripherals, etc.) It is also worth noting that some structural codes do not synthesize well. An example of this is using a large RAM in a hardware design. A RAM is not difficult to describe structurally because of its simple, repetitive design. However, in simulation this tends to produce a large data structure and that needs to be traversed every time a signal changes. In contrast, a behavioral model of RAM simulates quickly because it matches the processor's architecture well.

2.2.2 Verilog

Another common hardware description language is Verilog. Verilog has many similarities to VHDL as both were originally intended describe hardware circuit designs. Verilog is considered to be less verbose than VHDL, often making it easier to use, especially for designers more familiar with an imperative coding style like C++ or Java. As with VHDL, Verilog became more than just a textual representation of a circuit. Designers used Verilog to simulate circuits which eventually led to a subset of the language supporting hardware synthesis.

In Verilog, there are three major styles or forms of writing hardware descriptions. Both styles are valid VHDL codes; however they model hardware differently. This impacts synthesis, simulation, and, in some cases, designer productivity. These forms

are:

Gate Level Modeling Circuits are described in terms of logic units.

Structural Circuits are described in terms of modules.

Behavioral Circuits are described in an imperative (procedural) language to describe how the outputs are related to the inputs.

2.3 Xilinx Integrated Software Environment

The Integrated Software Environment (ISE) contains a suite of commands that can turn FPGA designs described in hardware description languages and netlists into bitstream configuration files. To gain a better understanding of the tool flow it is necessary to cover the underlying commands [17] that are called by the ISE GUI.

2.3.1 Xilinx Synthesis Tool

At the core of the ISE tool chain is the Xilinx Synthesis Tool [18] (XST). XST is used to synthesize hardware description languages into a netlist. XST is not the only synthesis tool available; however, as it is available within the ISE tool chain, it is the most obvious synthesis tool to use.

To help understand how XST works, let's begin by synthesizing a simple VHDL version of a 1-bit full adder. In addition to the VHDL file, two files are needed to run XST in commandline mode. These two files are the project file and the synthesis script file. The project file specifies all of the HDL in the project to be synthesized. It is commonly named with the extension `.prj`. In the full adder example, only one VHDL file exists so the project file simply contains:

```
vhdl work fadder.vhd
```

The three columns denote the HDL type, library name and VHDL filename. The `work` library is the default library to use.

```
run
-ifn fadder.prj
-ofn fadder.ngc
-ofmt NGC
-top fadder
-opt_mode Speed
-opt_level 1
-iobuf NO
-p xc5vfx130t-ff1738
```

Figure 2.8: sample synthesis script for XST

The XST synthesis script is commonly named with the extension `.scr`. This script contains parameters used by XST during synthesis. It is possible to run XST without a script by entering each option on the commandline; however, creating a single script is the preferred method since it reduces the redundant typing of the long series of inputs at the commandline. For the full adder example, the synthesis script is shown in Figure 2.8.

The `run` keyword will indicate to XST to execute synthesis with the following attributes. XST requires an input project filename (ifn), which is the project file we mentioned earlier, containing the HDL to be synthesized. The output filename (ofn) is the name the synthesized netlist will be given after successful synthesis. The output file format (ofmt) is set to NGC, Xilinx's proprietary netlist format.

Next is the top-level entity name (top) attribute which in our full adder example is `fadder`. Two synthesis optimization options follow, one which specifies whether XST should synthesize for speed (provide the highest operational frequency possible) or area (pack the logic as tightly as possible). The second option is for the synthesis effort level. A trade off between synthesis effort and synthesis time is made, higher levels may provide more resource efficient or frequency efficient designs at the expense of longer synthesis times. The iobuf attribute adds I/O Buffers to the top level module. In the full adder example we will choose not to insert I/O Buffers. The last attribute in this example is the FPGA part type (p). Here we specify to synthesize for the Virtex 5 FX 130T FPGA.

```
=========================================================================
*                            Final Report                               *
=========================================================================
Final Results
Top Level Output File Name         : fadder.ngc
Output Format                      : NGC
Optimization Goal                  : Speed
Keep Hierarchy                     : no

Design Statistics
# IOs                              : 5

Cell Usage :
# BELS                             : 2
#      LUT3                        : 2
=========================================================================
```

Figure 2.9: sample XST synthesis final report

There are additional commandline arguments that can be added to the synthesis script file; however, for this first XST example these are sufficient to produce a netlist. The final step is to run XST on the commandline, passing the synthesis script file as an input file.

```
$ xst -ifn fadder.scr
```

After XST completes, a report is written to a synthesis report file (srp). If there are any syntax errors XST will report approximately where in the file the line exists that failed synthesis. This is similar to compiling a C binary. Understanding the report is an important tool for the designer when trying to identify resource consumption and timing analysis. Under the `Final Report` heading, Figure 2.9, is a list of the basic elements (BEE'S) needed to represent the digital circuit. In our design 2 3-LUTs are needed for the two outputs, sum and carry-out, in the full adder.

Another important section in the synthesis report file is the device utilization summary section, Figure 2.10. For the specific FPGA device the number of slices, LUTs, flip-flops, BRAMs, etc. are listed, giving the designer an approximation to the amount of resources the design requires.

```
Device utilization summary:
---------------------------
Selected Device : 5vfx130tff1738-3

Slice Logic Utilization:
 Number of Slice LUTs:                  2  out of  81920     0%
    Number used as Logic:               2  out of  81920     0%

Slice Logic Distribution:
 Number of LUT Flip Flop pairs used:    2
   Number with an unused Flip Flop:     2  out of      2   100%
   Number with an unused LUT:           0  out of      2     0%
   Number of fully used LUT-FF pairs:   0  out of      2     0%
   Number of unique control sets:       0

IO Utilization:
 Number of IOs:                         5
 Number of bonded IOBs:                 0  out of    840     0%
```

Figure 2.10: sample XST synthesis device summary report

```
Timing Summary:
---------------
Speed Grade: -3

   Minimum period: No path found
   Minimum input arrival time before clock: No path found
   Maximum output required time after clock: No path found
   Maximum combinational path delay: 0.400ns
```

Figure 2.11: sample XST synthesis timing report

The synthesis report also provides some rough timing information, Figure 2.11. These numbers are not accurate because they do not consider the actual placement of the circuit on the FPGA. It does provide an approximation to the minimum period and maximum combination delay the circuit can obtain. Looking at the report our full adder example was purely combinational, there is no clock and as a result no minimum period. For larger designs when trying to meet timing, the synthesis report can help identify which component is causing the timing error.

For large systems with many components and subcomponents XST can be used to synthesize the entire design into a single netlist or it can be used to synthesize individual components in a hierarchical fashion. Figure 2.12 depicts this synthesis

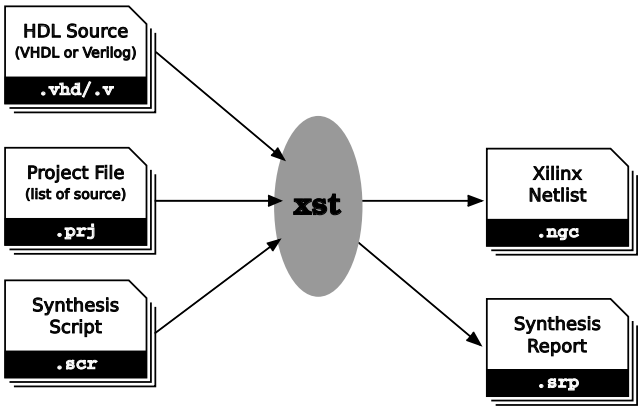


Figure 2.12: Xilinx synthesis flow

flow for a design. While each approach has their merits, the second approach is more often used as it provides more immediate and useful feedback for each component. This approach stems from the philosophy of bottom-up design where components are created (typically in HDL) from the ground up. It is only natural to synthesize in the same fashion, generating intermediate netlists and producing individual synthesis reports. The designer can view these reports to determine which of the components need to be redesigned in the event of resource or timing constraints.

For example, consider a design consisting of three components, a top-level component and two low-level components. XST would be used to synthesize the top-level and two low-level components individually. The top-level component synthesis would treat the two low-level components as `black boxes`. The immediate advantage of such an approach is to minimize the number times each component needs to be re-synthesized. If one of the low-level component's HDL is modified, only that low-level component requires being synthesized again, unless the modification changes the entity's generic or port list, forcing the top-level component to change its instantiation of the design. These are akin to how libraries are in C/C++ during compile time.

2.3.2 Netlist Builder

After XST is used to generate the system's netlist(s) the next step is to combine the netlists and any specific design constraints to produce a single netlist file. This

is typically accomplished through the use of the following two commands, NGCBuild and NGDBuild.

2.3.2.1 NGCBuild

If the designer has chosen to synthesize each component individually in a hierarchical manor, the next command needed is called `ngcbuild`, which combines multiple netlist files into a single `ngc` netlist. NGCBuild opens the top-level netlist and matches any subcomponents with existing netlists. NGCBuild does not produce errors or warnings when one or more subcomponent's netlists are not found. In the previous example, the top-level component netlist may be combined with one of its two subcomponents in the event only one of the two components has been synthesized. NGCBuild can be called again to combine the missing subcomponent netlist when it is available. This is similar to the component level synthesis approach mentioned earlier in that a component's netlist is constructed as its subcomponents become available. An example of the NGCBuild commandline execution is:

```
$ ngcbuild input/system.ngc output/system.ngc -sd implementation
```

The input to NGCBuild is the top-level netlist (in this example it is `input.ngc`). The output netlist that is generated by NGCBuild is the second parameter (`output.ngc`). The flag `-sd` specifies the search directory to look for any component's netlist to be combined to generate the output netlist.

2.3.2.2 NGDBuild

Once all of the components have been synthesized `ngdbuild` is used to generate a Xilinx Native Generic Database (NGD) file. The NGD file is used to map the logic contained within the netlist to the specific FPGA device. NGDBuild takes a single NGC file (created by NGCBuild) and any device, netlist or user constraints file. User constraints would include specific FPGA pins to be used for components such as RS232 UARTs and off-chip memory like DDR2 modules. NGDBuild will produce errors during runtime if any of the netlists are missing in the input NGC file or if there are unbound user constraints. An example of NGDBuild is:

```
$ ngdbuild -p v5fx130t -bm system.bmm -uc system.ucf system.ngc system.ngd
```

The commandline flags used by NGDBuild are `-p` to specify which FPGA device to generate the ngd file for, `-bm` to specify the block memory file which is a listing of all of the BRAMs in the system, and `-uc` to specify the constraints file listing all of the design's constraints (timing, IO, IO Standards). The final input is the NGC file generated by NGCBuild. The single output is the NGD file that will be passed on to Map, the next stage in the tool flow.

2.3.3 Map

The Xilinx program `MAP` takes an NGD file and maps the logic to the specified FPGA device. MAP performs a design rule check (DRC) to uncover physical or (if possible) logical errors in the design. After passing the DRC, the logic described by the netlist is mapped to the FPGA device's components (such as BRAM, IOBs, and LUTs). MAP also trims any unused logic or netlists. Unlike synthesis which can only trim internal signals if they are not used, MAP analyzes the entire system to determine if any logic is unnecessary. As a result, after MAP completes, the resource utilization reported is a more accurate representation than the synthesis report. Previously, the synthesis report's resource utilization was used to give an approximate first order estimation of the resources needed by the design. Now using MAP's report the actual resource utilization is given. When finished, MAP produces a Native Circuit Description (NCD) file which can be used by the next tool to place and route the design to the target FPGA. An example of MAP's commandline is:

```
$ map -o system_map.ncd -pr b -ol high system.ngd system.pcf
```

The MAP flags are used to give the designer more control over what MAP placements are or are not run. Typically, a designer will be able to use the default options; however, there are cases when a nearly fully occupied FPGA design may required additional constants to MAP successfully. The `-o` flag sets the output file name, in

this case the resulting NCD file is named `system_map.ncd`. The `-pr b` option specifies to pack flip-flops and latches in both input and output registers, while the `-ol high` flag is the overall effort level for the placement algorithm, high is used to achieve the best placement at the expense of longer MAP runtimes. The input file, `system.ngd`, is the NGD file generated by NGDBuild. We already stated that `system_map.ncd` is the output from MAP, but in addition is `system.pcf`, the physical constraints file which are constraints placed during the design creation.

2.3.4 Place and Route (PAR)

The Xilinx program `PAR` actually consists of two programs, Place and Route. These two programs are commonly run in series, so Xilinx has combined them into a single command. PAR takes the NCD file generated by MAP and runs both a placement and routing algorithms to generate a routed NCD file. To begin, Place tries to assign components into sites (LUTs, BRAMs, DSP48E slices, etc.) based on any specific constraints (i.e. use pin location P38 to output the transmit (TX) line of the RS232 UART component) to maximize resource utilization while minimizing component distances (which will make routing and meeting timing requirements easier). Placement occurs through multiple phases (passes) and produces a placed NCD file.

After Place, Route is run to connect all of the signals for the components based on the timing constraints. Routing is also a multi-phase operating resulting in a routed NCD file. After PAR completes, a timing analysis is performed to verify the design has met timing and if not, produce a short log to help the designer identify and fix the errors. As the design size and clock rates increase, timing becomes more constrained and difficult to meet. It is possible to run PAR with different commandline arguments in order to meet timing. PAR consumes the most time during the execution of all of the tools (as the design increases). Identifying fast *build* computers to run PAR on will reduce the amount of "idle" time before a design can be tested on the FPGA.

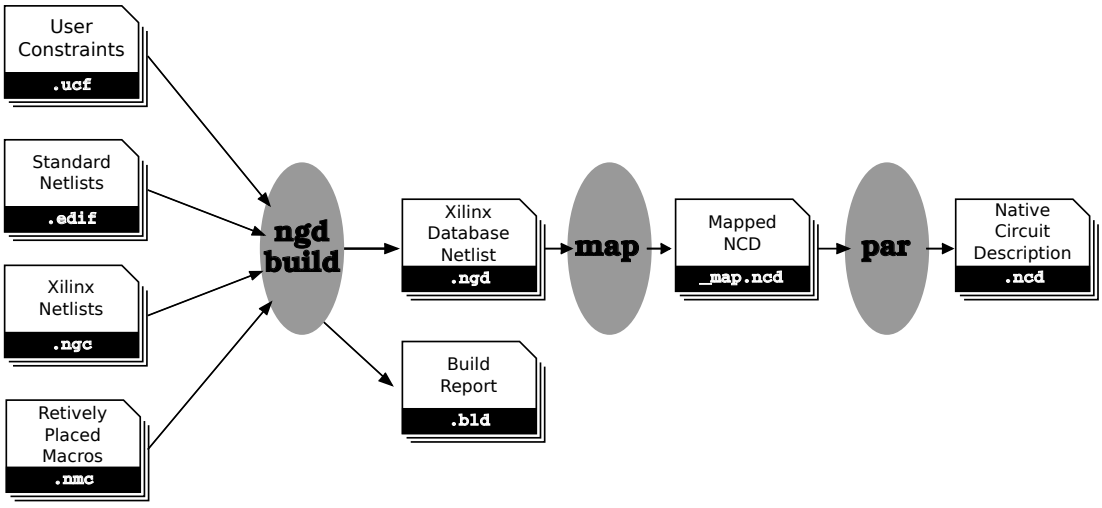


Figure 2.13: Xilinx implementation flow

An example of PAR commandline is:

```
$ par -ol high system_map.ncd system.ncd system.pcf
```

PAR takes as input the NCD file and PCF files from MAP and generates an output `system.ncd` file which is the placed and routed native circuit description.

Figure 2.13 illustrates the flow from netlists to a fully placed and routed design. This flow picks up after the synthesis to netlists flow that is run for each component in the design. These flows are used to help picture the process from source files (HDL) to just before generation of a bitstream, which will be covered next.

2.3.5 Configuration Bitstream Generation

The final command in generating a configuration file for the FPGA is called `bitgen`. BitGen produces a configuration `bitstream` for the specific Xilinx FPGA device. The bitstream (.bit) file contains configuration information (proprietary to Xilinx) which is downloaded and stored into the SRAM cells of the FPGA device. The concept of *programming* an FPGA is when the bitstream is downloaded to the FPGA, at which point the digital circuit is realized on the FPGA fabric. An example of the BitGen commandline is:

```
$ bitgen -f bitgen.ut system.ncd system.bit
```

The BitGen command can take as input a parameters file (`-f bitgen.ut`) which can be used to specify configuration information. The output `system.bit` is the configuration bitstream that can be used to "program" the FPGA.

These short sections have covered the process from synthesis to bitstream in an effort to be able to better understand the entire ISE flow. The proposed work relies on this flow as a foundation for creating bitstreams; however, the process may not be followed exactly as described above. These proposed differences will be presented in more detail in the following chapters.

2.4 Reconfigurable Computing Cluster

The Reconfigurable Computing Cluster Project [19] is investigating the role (if any) FPGAs have in the next generation of very large scale parallel computing systems. Faced with growing needs of computational science and the existing technology trends, the fundamental question is: *how to build cost-effective high performance computers?* To answer this question, the RCC project identified four research areas to investigate (which recently have gained attention [20]) namely: memory bandwidth, on-chip and off-chip networking, programmability, and power consumption. While the focus here is on the networking, the three remaining categories clearly play a significant role and, as a result, will also be present in the discussion.

There are numerous aspects to FPGAs that make them especially useful for high performance computing. One advantage is that because a single FPGA is now large enough (and has a rich enough set of on-chip IP), it is able to host an entire system-on-chip. This eliminates many of the peripheral components found on a standard commodity PC board which in turn offers size, weight, and power advantages. Also, by reducing the size of the node, there is a secondary advantage in that it reduces the distance between nodes enabling novel networking options. Research also indicates that FPGA floating-point performance will continue to rise (both in absolute terms and relative to single-core processors [21]).

Certainly, there are potential pit-falls that could limit the expected performance gains of such an approach. Until recently, FPGAs have had the reputation of being slow, power hungry, and not very good for floating-point applications. However, these generalizations are based on a snap shot in time and depend heavily on the context in which the FPGA is employed. Moreover, while a number of HPC projects are using (have used) FPGAs, they have been used as "compute accelerators" for standard microprocessors. The RCC project's proposed approach is fundamentally different in that the FPGAs are promoted to first class computation units that operate as peers. The central hypothesis of the project is that the proposed approach will lead to more cost-effective HPC system than commodity clusters in the high end computing range.

Ultimately, the goal of this project is to investigate how FPGAs could be used towards assembling high performance computing systems. Furthermore, the project aims to generate a collection of reusable modules to aid in the development of designs being implemented on such a system comprised of FPGAs. Towards this goal, the RCC project is a multi-disciplinary project, bringing together engineers and computational scientists from several Universities to test real, computationally challenging science applications on a prototype cluster. Teaming computer engineers and domain scientists on every application is not an effective, long-term solution; however, it does provides invaluable hard data for computer engineers evaluating the design while advancing our collaborator's science program.

The RCC's initial investigation began in late 2007 with the assembly of, *Spirit*, a small scale cluster of 64 all-FPGA compute nodes (with no discrete microprocessors in the design) arranged in a 4-ary 3-cube network topology. Each compute node consists of a Xilinx ML410 development board [22] with a Virtex 4 FX60 FPGA, 512 MB of DDR2 memory, Gigabit Ethernet and a custom high speed network (among several other peripherals). Figure 2.14 depicts a simple representation of the organization of the cluster. The server node is a commodity x86 server used to manage the cluster

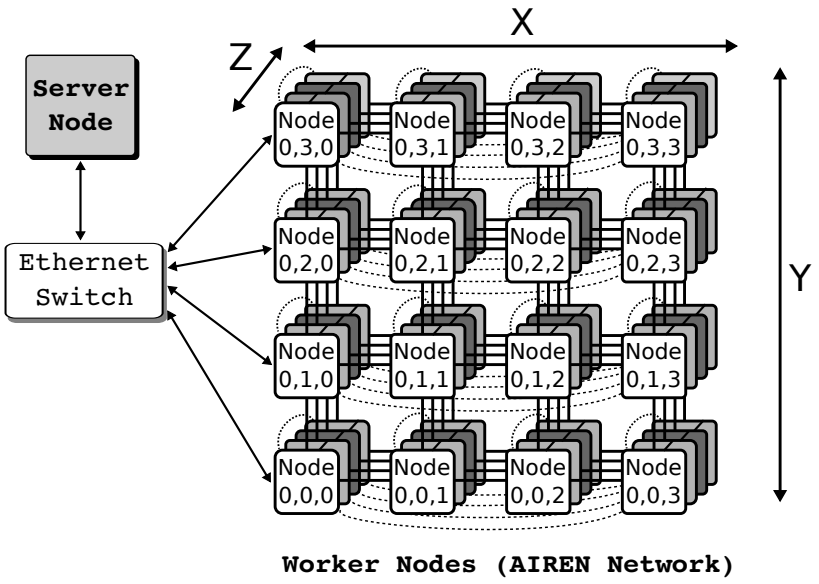


Figure 2.14: the Spirit cluster consists of 64 FPGAs connected in a 4-ary 3-cube through a high speed network (AIREN)

and provide a network filesystem for the cluster. The server node connects to the cluster through a commodity Gigabit Ethernet switch to which each FPGA node also connect.

Each FPGA node is essentially a "blank slate." Its functionality is determined after it has left the factory and most devices can be repeatedly reconfigured as needed. In Spirit, all of the FPGAs are blank at power on. Each node loads a default configuration that boots a simple embedded system design. This system allows it to receive an application-specific configuration bitstream over a secondary Gigabit Ethernet network. (This network is just used for administrative traffic, such as rsh/ssh, ntp, NFS, etc.) Figure 2.16 illustrates a typical configuration which consists of one or more processors, custom compute cores, memory controllers, on-chip interconnects, and off-chip network interfaces. Details regarding the network will be discussed in Section 2.5.

2.5 Spirit's Integrated On-Chip and Off-Chip Network

At the heart of the Spirit cluster is the Architecture Independent REconfigurable Network (AIREN) which is an integrated on-chip/off-chip network that not only

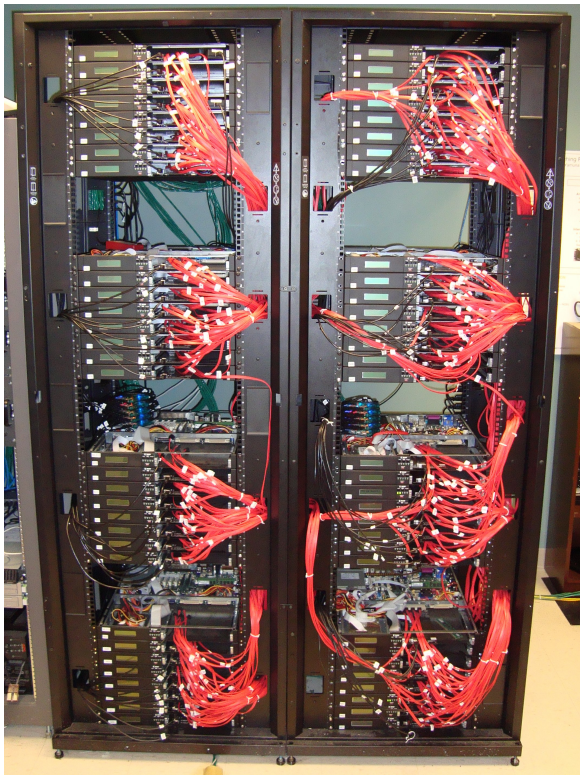


Figure 2.15: 64 node Spirit cluster at UNC Charlotte

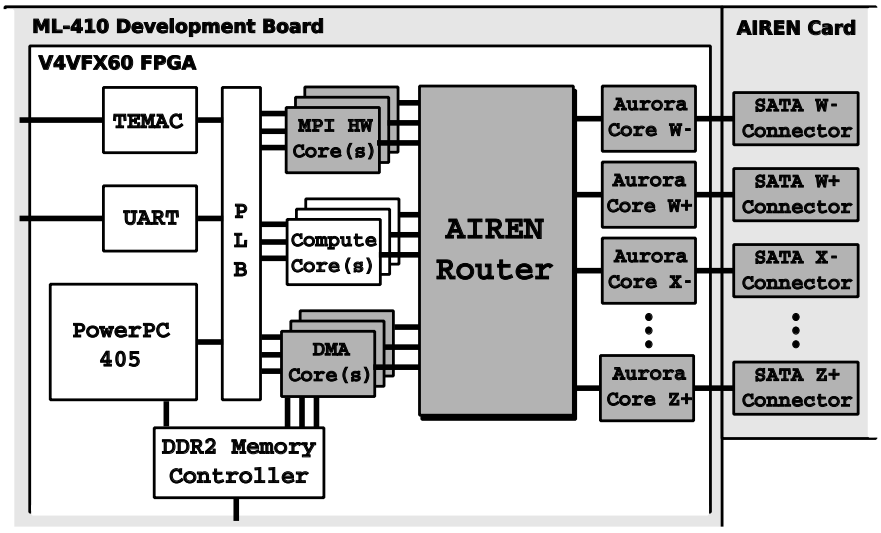


Figure 2.16: typical base system configuration

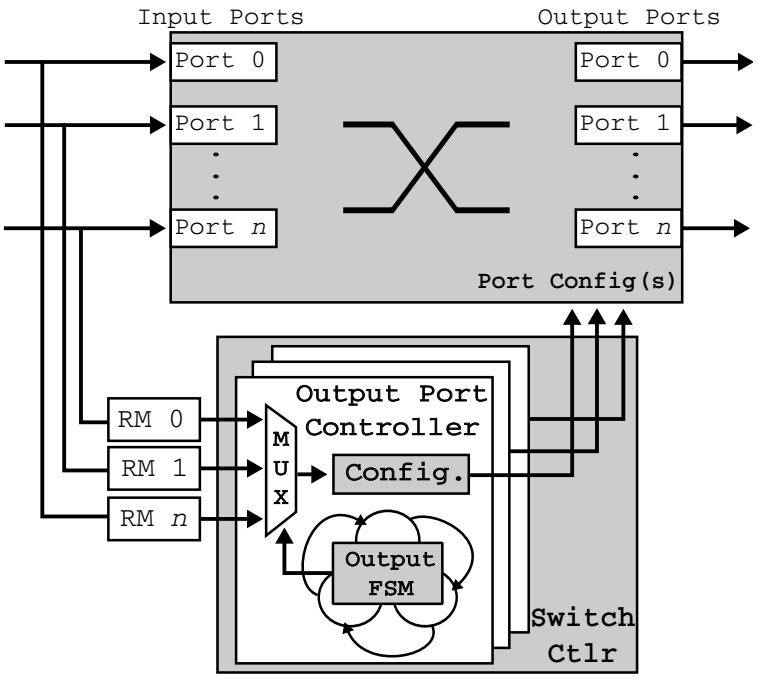


Figure 2.17: AIREN router block diagram

supports efficient core-to-core communication, but node-to-node as well. Figure 2.16 shows a high level overview of the FPGA's system-on-chip. On the FPGA, AIREN spans several IP cores, of which the AIREN router is the most central component. The original implementation of AIREN has been published in a short paper in 2009 [23]. Several designs have relied on the AIREN integrated network and have been published accordingly [24, 25, 26, 27, 28, 29].

AIREN offers several configurability options in order to provide the desired network behavior. These include the ability to change the number of input and output ports (radix) of the switch, the data width, operating frequency, whether the switch has software controlled routing or hardware accelerated routing, the dimensionality of the off-chip network (ring, mesh, cube, etc), and the routing methodology. These details and more regarding the AIREN network will be provided within this section.

2.5.1 AIREN Router

Figure 2.17 details the internal components and functionality of the AIREN router. The router consists of a single crossbar switch implemented in the FPGA's pro-

grammable logic along with routing decision modules that inspects the header of each packet and passes the routing decision to the switch controller. The switch controller manages the connections and deals with contention for output ports. The crossbar switch is implemented as a generic VHDL model to allow easy customization of the number of ports needed for a given implementation. The signals that are switched include 32 bits of data along with 4 bits of flow control signals. On-chip, cores connect to the switch for high bandwidth, low latency communication. Off-chip, cores communicate through a set of special ports which interface to the FPGAs multi-gigabit transceivers for communication with other FPGAs.

2.5.2 AIREN Interface

One of the first architectural enhancements to the AIREN router is changing from the use of an in-house interface (AIREN network interface) to connect compute cores to the network. A consequence of that interface was a less than optimal latency of 0.08μs across the switch. More recently, work has been done to migrate to a more widely used and accepted interface standard supported by Xilinx known as LocalLink [30]. With the adoption of this standard, compute cores can be quickly connected to the network. In addition, LocalLink provides flow control which reduces the need to incorporate buffers throughout the network, freeing up scarce on-chip memory resources (BRAM). Furthermore, LocalLink supports frames, which means headers and footers are more easily identifiable and results in a reduction the latency across the switch to 0.02μs. Figure 2.18 provides a simple diagram of how the LocalLink standard is incorporated into the network and specifically can allow compute cores to be chained together and/or connected to the AIREN router.

It is important to emphasize the AIREN network does not require the use of the crossbar switch. In fact, compute cores can be directly connected together to form a pipeline or stream of compute cores. That is, data output from one core is fed as input to the next core. Since each application may require different communication

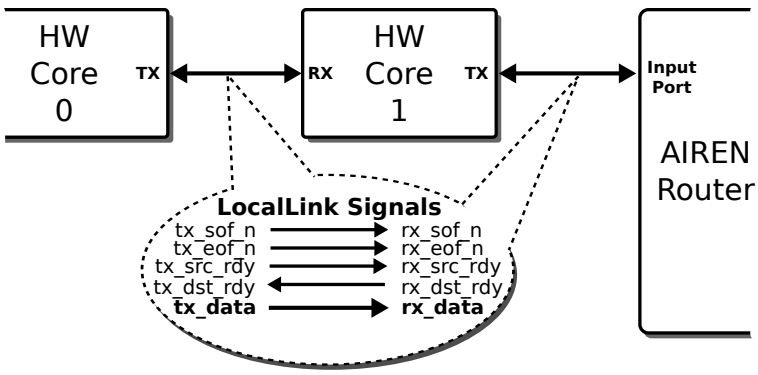


Figure 2.18: simple illustration to show the Xilinx LocalLink standard implemented within the AIREN network to support both core-to-core and core-to-router connectivity

paths, being able to assemble systems with direct connects would result in the best performance. Ultimately, many applications require variable connectivity of communication paths. Therefore, the AIREN network's on-chip network employs a single stage full crossbar switch.

2.5.3 AIREN Routing Module

Each input port connects to both a port on the crossbar switch and to a routing module hardware core. The routing module examines the header of each incoming packet to determine its destination. Each node receives a unique `node id` and each core has a `core id` that is only required to be unique on each node; a decision that was made to simplifying routing to first route to a node and then route to a core. The routing module looks at the packet header and identifies the packet's destination in terms of node ID and core ID. If the packet's node ID matches the local node's ID, then the core ID is used to route the packet to the corresponding core. If the IDs differ, the routing module directs the packet to the corresponding node's off-chip port, depending on the routing algorithm. The default routing is a simple dimension-ordered routing algorithm to determine which out going port the message is routed to [31].

The routing module is configurable to support different off-chip routing algo-

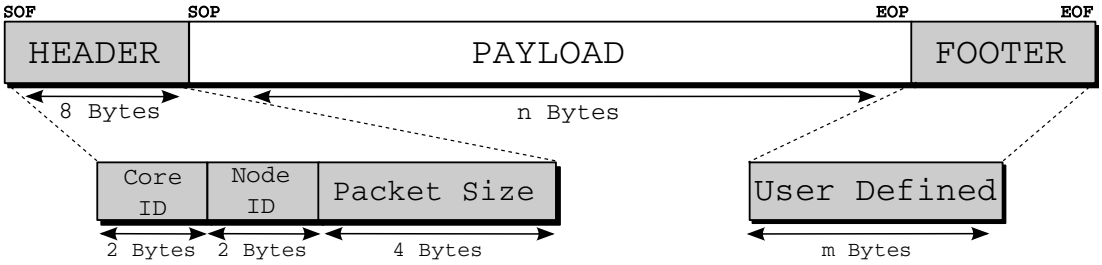


Figure 2.19: AIREN packet structure

rithms, such as dimension order or wormhole routing. Once a routing decision has been made, the routing module passes the routing request to the switch controller and must wait until it is granted access to the crossbar switch. When an input port is granted access to the output port, it owns that port until the packet is finished.

A packet includes within the first 8-byte header both the node and core id, as can be seen in Figure 2.19. The LocalLink start-of-frame (SOF) signal marks the beginning of a packet. The routing module analyzes the header to make the routing decision, otherwise the routing module is a passive component in the system. At present no footer is used; however, for future applications to use such a feature would require no modifications in the AIREN network. Finally, when the LocalLink end-of-frame (EOF) signal is asserted, the routing module can de-assert its request for the output port and the switch controller is free to reconfigure the connection to another input port.

2.5.4 AIREN Switch Controller

Once the routing decision has been made and the output port identified, the switch controller configures the crossbar switch to connect the input and output ports. By itself, the crossbar switch merely connects inputs to outputs with a single clock cycle of latency to register inputs to outputs. However, to control which inputs are connected to which outputs requires a switch controller. The AIREN network is configurable to support either a software controller or a hardware controller. The software controller enables the processor to set the connections within approximately

0.07μs. For systems with minimal changes to the communication path, the software controlled switch is appealing as it offers low resource utilization.

Of course, as the radix increases so too does the demand on the processor to make routing decisions. The switch can be controlled by hardware by connecting each input port to a routing module. Each routing module can make its own routing decision in parallel and notify the switch controller. The switch controller monitors each routing module and configures the switch based on input requests and output ports availability. Requests for separate output ports can be handled in parallel. To deal with contention, a simple priority encoder is used to give pre-determined ports a higher priority. This mechanism can also be customized to support other arbitration schemes with minimal effort.

2.5.5 On-Chip/Off-Chip Network Integration

The AIREN network card was designed in house to connect the nodes of the cluster. Each link in the direct connect network is a full duplex high-speed serial line capable of transmitting/receiving at 8 Gbps (4 Gbps in/4 Gbps out). Each FPGA node in *Spirit* has eight links for a theoretical peak bandwidth of 64 Gbps to/from each node. The network was physically implemented by fabricating a custom network interface card (see Figure 2.20) that routes eight of the FPGA's integrated high-speed transceivers to SATA receptacles (we use SATA cables but not the SATA protocol). The AIREN network card has additional hardware used to manage the cluster, configure the FPGAs, and debug active designs.

The off-chip network is centered around the AIREN network card. Since each node can connect up to eight other nodes, a variety of network topologies are possible. The experiments reported here use a 4-ary 3-cube (four nodes in each of the three dimensions). Within the FPGA, each MGT is interfaced through the Xilinx Aurora protocol [32]. Aurora is a component released by Xilinx which wraps the multi-gigabit transceiver present on most of the new Xilinx FPGAs in an simple LocalLink

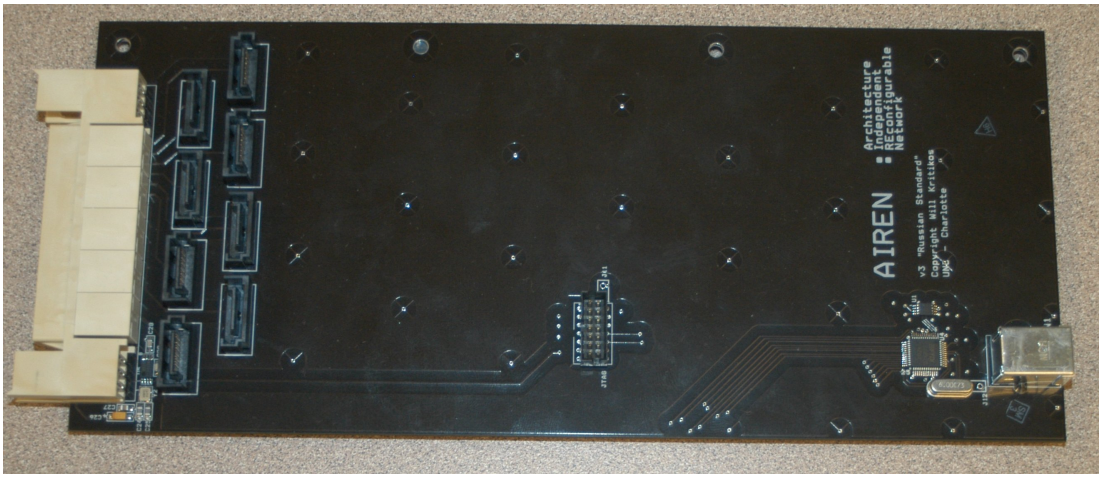


Figure 2.20: AIREN network card

interface. To this core the AIREN network adds custom buffers and flow control to more efficiently support back pressure between nodes. The Aurora core is configured to support bi-directional transfers of 32-bit at 100 MHz, resulting in a peak bandwidth of 4.00 Gbps. To this, Aurora uses 8B/10B encoding which further reduces the peak bandwidth to 3.2 Gbps.

2.5.6 AIREN Resource Utilization

There are three key parameters that can change the performance and resource requirements of our network core. First is the number of ports. This is the dominant parameter because, in general, the resources required to implement a full crossbar switch grows exponentially with the number of ports. The second parameter is the port bitwidth; it defines the number of bits transmitted in parallel each clock cycle. The resources of the crossbar grows linearly with the number of bits per port. The third parameter is clock speed. Technically, all of the devices we consider have a maximum clock speed of 500 MHz; however, in practice, unless the design is hand optimized frequencies between 100 and 200 MHz are typical. In Table 2.1 the resources consumed by a software controlled and a hardware controlled AIREN network. The purposes is to report on the scalability of the approach given the Virtex 4 FX60 FPGA device resources. The key difference is the 4-input lookup table (4-LUT)

Table 2.1: AIREN resource utilization (V4FX60)

	Software Routing		Hardware Routing	
# Ports	**# Slice FFs (%)**	**# 4-LUTs (%)**	**# Slice FFs (%)**	**# 4-LUTs (%)**
4	433 (0.85%)	669 (1.32%)	305 (0.60%)	655 (1.30%)
8	703 (1.39%)	1,559 (3.08%)	511 (1.01%)	2,009 (3.97%)
16	1,263 (2.49%)	5,589 (11.05%)	964 (1.91%)	8,228 (16.27%)
32	2,471 (4.89%)	20,757 (41.05%)	1,933 (3.82%)	33,603 (66.46%)

Table 2.2: largest 32-bit full crossbar switch possible in 15% of the device

Xilinx Part	**CMOS**	**Year**	**percentage**	**# ports**
Virtex 2 Pro 30	90nm	2002	15%	16
Virtex 4 FX60	90nm	2004	15%	20
Virtex 5 FX130T	65nm	2009	15%	35
Virtex 6 LX240T	40nm	2010	15%	70
Virtex 7 855T	28nm	2011	15%	84

utilization that is needed for the switch controller when performing hardware routing. In contract, the software routing requires slightly more slice flip-flops (FFs) so the processor can interface with each output port configuration register.

When considering trends, it may be more instructive to increase the ports on the switch proportional to the growth in the number of logic resources. When considering different FPGA devices (different CMOS technologies) the resources needed to support a high radix full crossbar switch on an FPGA dramatically decreases. So much so, that current Virtex 6 and emerging Virtex 7 devices make a full crossbar switch approach such as AIREN highly feasible. This is seen by fixing the switch to consume no more than 15% of the resources, as seen in Table 2.2. Since the behavior of the switch is very predictable and increasing the number of ports per switch will only reduce contention, these data suggest that the proposed approach is not just acceptable for the current generation of FPGA devices, but also for the foreseeable technologies.

2.5.7 AIREN Performance

First, we test the network performance in isolation. Network layer messages between hardware and software are identical in the AIREN network. However, the

Table 2.3: hardware send/receive latency through one hop

Sender/Receiver Pair	**Latency (μs)**
hw-to-hw (on chip)	0.02
hw-to-hw (off chip)	0.80
sw-to-hw (on chip)	0.15
sw-to-hw (off chip)	0.98
sw-to-sw (off chip)	2.00

ability of a core to generate or accept a message depends on the core. If a software task is the intended recipient, there is a significant amount of overhead (a processor interrupt, context switch, data, and more copies) compared to a typical hardware core (that is usually designed to produce or consume the message at network line speeds). There is also a speed difference if the message stays on-chip versus off-chip. This is because the network layer message is transmitted using a data link layer, Aurora, which handles transmission details (such as 8B/10B encoding, clock correction, etc.) Hence, we test five different combinations. The latencies are measured by performing a ping-pong test, measuring the round-trip time for a message to be transmitted from source to destination and then sent back. The round-trip time is then divided by two. The various latencies for a single hop are shown in Table 2.3. Using the hardware core to hardware core testing infrastructure, we looked at how the latency scaled with multiple hops as well. The results are summarized in Table 2.4. The results show that as the number of hops increases, the latency per hop is decreasing. This suggests that the dimensionality of the network can be trusted to predict the overall latency of the network. This is an extremely important result because it speaks to the overall scalability of the design. (Note: all of these tests were conducted without any other jobs running on the machine, thus there was no outside contention.)

To measure the bandwidth of the network, we varied the message length and repeated the ping-pong test. The measured bandwidth is plotted against the message length in Figure 2.21. Although the channel transmits at 4 Gbps, the data link layer performs 8B/10B encoding, a technique used to keep the sender and receiver

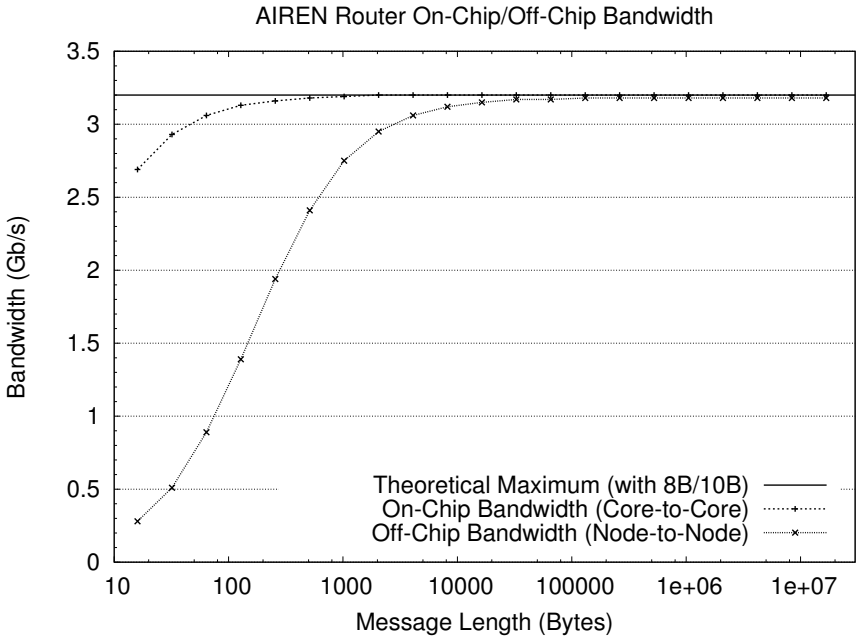


Figure 2.21: AIREN's On-Chip and Off-Chip measured bandwidth with hardware based routing

Table 2.4: node-to-node latency with AIREN router

Hops	**Latency**	**Latency/Hop**
1	0.81 μs	0.81 μs
2	1.56 μs	0.78 μs
3	2.31 μs	0.77 μs
4	3.08 μs	0.77 μs

synchronized. This, plus other communication protocol overhead, will decrease the effective data rate. Thus, after encoding, the peak theoretical bandwidth is 3.2 Gbps. Based on the data, messages on order of 10,000 to 100,000 bytes long approach optimal throughput. For on-chip bandwidth tests, messages are sent from one core to another core on the same node through the crossbar switch.

CHAPTER 3: RELATED WORK

Research in the area of high performance reconfigurable computing has taken off over the past few years. With the increase in resources with each new generation of FPGAs, researchers are finding new ways to outperform previous implementations as well as explore new areas that were previously restricted by the technology. Section 3.1 covers some of the important historical efforts as well as current projects in the high performance reconfigurable computing area. Furthermore, as the amount of resources increases and more heterogeneous systems are constructed, so to are tools to try and support the scalability of the design. Section 3.2 details the efforts in this research topic. This is followed by Section 3.3 which discusses some of the current productivity tools available to designers when creating or modifying hardware designs. This chapter concludes with Section 3.4 providing background on various performance monitoring and system monitoring tools and projects.

3.1 High Performance Reconfigurable Computing

Almost since the introduction of FPGA devices in 1984, researchers have investigated ways of using FPGAs in High-Performance Computing. It was the explicit goal of the Splash-2 project [33] and other contemporary projects [34]. Carefully engineered FPGA-based solutions have frequently been compared against the fastest computers available [35]; however these early examples demonstrated the potential performance and downplayed the development costs. The easiest way to introduce an FPGA accelerator is to create an add-on card that goes into a peripheral bus but these co-processors suffer from the fact that they do not share the same memory hierarchy as the processor. For many applications the transfer-of-data costs counter any computational speed gains.

One of the earliest efforts to incorporate FPGAs into MPI-based parallel computing was the Adaptable Computing Cluster Project that emerged in 2001. That project put an FPGA on the network interface card between the peripheral bus and the media access controller of the network. The principle idea was that simple functions (implemented on an FPGA) could operate on MPI messages in transit. This eliminated the transfer-of-data cost since the programmer-initiated messages already had to pass through the network. Results showed that significant improvements in overall system performance [36, 37] were possible. SRC adopted a similar approach (FPGA between the processor and switch) although SRC does not use MPI. Octiga Bay (now part of Cray, Inc.) put the FPGAs on the backplane of the XD-1 [38] and Silicon Graphics, Inc. has a similar technology (RASC) [39]. These systems support MPI but the programming model as a co-processor have access to the processor's NUMA memory hierarchy. Recently, plug-in co-processor solutions have appeared [40, 41]. These solutions literally replace processors with FPGA devices on multi-processor mainboards. This may prove better suited to ameliorate memory transfer costs.

When it became possible to instantiate multiple soft processor cores on a single silicon chip, a number of researchers considered MPI a natural choice and several researchers have investigated various approaches. For example, The University of Queensland (Australia) assembled multiple Microblaze software processors with a direct connect network on a single FPGA. A minimal set of MPI function calls were implemented to compile and execute simple MPI programs.

Recently, the field of HPC using FPGAs has been growing rapidly. Let's first cover some of the current research in the field of multi-FPGAs and FPGA cluster computing. These include RAMP, Maxwell, SMILE, TMD-MPI, QP, Axel, and Novo-G [42, 43, 44, 45, 46, 47, 48]. These projects, like the RCC's Spirit cluster, seek to use many FPGAs networked together in the hopes to further exploit the FPGA's

potential on-chip parallelism in order to solve complex problems faster than before.

The RAMP project [49] is a large multi-institution, university/industry collaboration interested in emulating future multi-core/many-core ICs with FPGAs. In particular, the RAMP Blue effort at Berkeley has assembled a system of 32 BEE2 boards to simulate 1000+ processor cores running MPI applications on a Linux-based system. RAMP Blue has asserted that they are not interested in building High-Performance Computing machines from FPGAs [50]. Rather, they are focused on emulating future fixed-function integrated circuits.

Maxwell uses the multi-gigabit RocketIO transceivers on a Virtex 4 to build a 8-ary 2-cube network between their FPGAs. This network is only used for direct neighbor to neighbor communications. The general node to node communication is handled by the 32 host CPUs via 1000 Mbit Ethernet with a 32 port switch [43].

The SMILE project uses an on-chip peripheral bus as its on-chip communication method and a simple ring network for the off-chip network topology [44]. Again the options for the study of off-chip network topologies are limited by the physical connectivity available with the SMILE project. Recently, the SMILE cluster was updated to use Virtex 5 FX parts; however, the size of the cluster remains at 16 nodes and the interconnection is still ring based with TCP/IP support to a central switch.

TMD-MPI [45] presents an integrated on-chip and off-chip network with additional integration for MPI. Similarities between AIREN and TMD-MPI exist at many levels, but key differences are in the on-chip network implementation, interface between the on-chip, off-chip network, scalability of the network and the integration of the processor(s) into the network.

The Quadro Plex (QP) cluster [46] developed by the National Center for Supercomputing Applications (NCSA) at the University of Illinois at Urbana-Champaign, is a 16-node cluster where each node includes a dual-core AMD 2.4 GHz CPUs, 4

NVIDIA Quadro FXS600 GPUs and 1 Nallatech H101-PCIx FPGAs. With a theoretical peach performance of 23 TFLOPs (single precision), 96% come from the GPUs, 2.6% from the processors and only 1.4% come from the FPGA. The Center is currently developing Phoenix [51] to program the cluster at a high level of abstraction to effectively utilize the resources of heterogeneous processing elements. Since its publication there has been little presented on Phoenix nor has there been any recent updates regarding the QP cluster.

Axel [47] is another heterogeneous cluster that includes both FPGAs and GPUs connected to host processors through a PCIe bus. The cluster consists of 16 nodes where each node includes an AMD Phenom Quad-Core CPU, an Nvidia Telsa C1060 GPU, and a Xilinx LX330 FPGA all in a 4U chassis. The cluster uses a Hardware Abstraction Model (HAM) to describe the available computational resources available in the system. This includes the type of computation, the amount of local memory and the communication interconnection.

The Center for High Performance Reconfigurable Computing (CHREC) is a research collaboration between several academic and commercial institutions which is investing high performance computing using FPGAs [48]. As part of this research NOVO-G, a 192 65nm FPGA cluster (Altera Stratix-III E260), has been assembled for testing purposes. Four FPGAs are connected together on a single PCB which is connected via PCIe to a general purpose processor. Two of these PCIe cards are connected with a processor to form a node in the cluster. The cluster is connected together with 20 Gb/s DDR InfiniBand. Each FPGA on a single PCB are connected together at 25 Gb/s.

This section highlights the growing trend of using FPGAs, in some form, for exploring future generations of high performance computing systems. Therefore, it is important to investigate ways to improve the hardware design process which in turn could help similar projects to the ones listed in this section. While this work focuses

on the Reconfigurable Computing Cluster project at the University of North Carolina at Charlotte, the knowledge, experience, and tools developed may be disseminated to a large volume of other hardware designers.

3.2 Existing Methods for Scaling Designs

Since the first production of FPGAs, research has been ongoing in how to map designs to the available resources and how to partition designs across multiple FPGAs in the event that a design exceeds the amount of available resources. Partitioning a design across a number of FPGAs has been under exploration since the early 90's. Woo and Kim presented an efficient method of partitioning circuits for multiple FPGAs [52] in 1993. At the time, the amount of resources on a single FPGA limited the designs that could be mapped to a single chip. Woo and Kim developed a method called MP2 for partitioning networks into multiple blocks with size and pin constraints. Since then, a number of efforts has been made to map designs to multiple FPGAs [53, 54, 55, 56, 57], which focus on the connectivity between FPGAs to pin based. That is, the FPGAs are connected together on a single PCB or through riser boards that share pins from one PCB to another. A major limitation of the partitioning approach is that the FPGAs must be directly connected which can limit the overall scalability of the system.

There have also been a number of attempts to either automate the process of utilizing a number of heterogeneous resources or to scale a system to those resources. When considering partitioning alone a system must be more tightly coupled for the resources to be shared in the system. The following approaches loosen the connectivity between systems and aim instead for a heterogeneous pool of resources. Partitioning, as was presented earlier, does not apply in these cases because of the diversity of the resources and their connectivity.

Ong et al. presented [58] which investigates the ability to automatically map applications to multiple Adaptive Computing Systems (ACS). The effort focused

on trying to reduce the development time when designing for a large amount of resources. The CHAMPION software was developed as a design environment to provide automatic mapping of applications in the Cantata graphical programming environment to ACSs.

Porrmann et al. present RAPTOR [59] as a scalable platform for rapid prototyping on FPGA-based cluster computing. The goal of RAPTOR is to enable designers a testing infrastructure to develop and experiment with designs in a much more cost effective manor than fabricating the design. Furthermore, RAPTOR offers orders of magnitude performance gains over simulation of large scale designs.

At the CHREC an automated scheduling and partitioning algorithm has been presented in [60]. The objective of the algorithm is to support multiple node reconfigurable systems. The algorithm is a two stage process which first creates an initial schedule and then second iteratively searches through the schedule for an optimal schedule. Specifically, the first stage employees a list-based scheduling technique which is fed into a modified Kernighan-Lin heuristic which analyzes a set of moves for each unfixed node in a task graph.

At the Center for Supercomputing Applications (NCSA), Pant et al. developed Phoenix as a runtime system designed to be executed on processors integrated with compute accelerators. Programmability of the system requires scheduling of tasks across the available resources. Phoenix is designed to support scaling applications from a single core (CMP) to potentially all cores in the system. Little documentation is available on the specifics of the Phoenix system or on how the system integrates with the heterogeneous processing elements in the system.

At Imperial College work with the Axel cluster has led to the development of a framework for programming a heterogeneous cluster of resources [61]. The proposed framework includes an execution model for applications to describe computation to collaborative accelerators. There also exists a modular programming model to allow

new types of accelerators to be used by the application without requiring the original application to be recompiled.

3.3 Productivity Tools

Tools of some form are needed to help the designer manage the complexities associated with hardware design, such as timing requirements, resource limitations, routing, etc. While Chapter 2 discussed conventional like synthesis, map, and place-and-route tools, these tools are just the beginning of what are available and what is to come. Tools released by vendors, such as Altera and Xilinx, cover a wide range of functions. There are component generators [62] to build mildly complex hardware cores, such as single-precision floating point units and FIFOs, saving development time. More recently, there tools such as Xilinx Base System Builder (BSB) Wizard [63] and Altera's System-on-Programmable-Chip (SoPC) Builder [64] help the designer construct a customizable base system with processors, memory interfaces, buses, and even some peripherals like UARTs and interrupt controllers. The designer can quickly create a base system using the tools and then add their own custom logic to the design, and in a short order be up and running on an FPGA. There are even tools that can help a designer debug running hardware similar to the use of logic analyzers in a microelectronics lab [65, 66].

Even with these, and many more, tools at the hardware designer's disposal system development takes time. As part of an FPGA Tool Flow Studies workshop held in June of 2008, members of the CHREC presented their take on the existing limitations of the tools and how to improve productivity [67, 68]. As FPGA capacity continues to increase, so too does the productivity gap. The workshop investigates how software development has been able to improve so quickly whereas hardware development still is using low-level tools (HDL,Schematics,netlists).

Software development's productivity comes from, among others, high-level languages, software reuse (libraries), structured development processes, and automa-

tion. Furthermore, software development benefits from fast compile times (100s of compiles per day) whereas hardware development compile times (synthesis,map,par) are at least an order of magnitude slower. CHREC proposes three focuses to improve productivity. Namely, abstraction, code reuse and verification. Abstract does not simply mean some high level language abstraction (like C-to-Gates), but synthesizing for multiple FPGAs and for parallel environments (like GPUs, Cell, etc.). Code reuse is also important, but more than just libraries and tools to generate code automatically. Code reuse is presented as interoperability between devices (and vendors). Designers are all to often stuck with Xilinx or Altera because the code has been written to support one vendor over the other. Writing more transparent code to mitigate obsolescence is necessary. Also discussed is interface synthesis which is closely related to the work presented here. That is to say, develop tools to synthesize circuit-specific interfaces for resources with little to no effort by the designer. Finally, verification is listed, where a design can be quickly checked for correctness. If a bug exists, there should be a mechanism to identify the bug and correct for it quickly, with minimal effort from the designer.

There are also projects investigating automatic code generation for VHDL, such as vMAGIC [69]. VHDL Manipulation and Generation Interface (vMAGIC), is a Java library which reads, writes, an manipulates VHDL code with the goal being to provide hardware designer's with tools to reduce development time. While the project show promise, the tools are more suited for enabling a designer to write in a higher level language (Java) rather than having to write low-level HDL. Presented along side vMAGIC is Hardware-in-the-Loop Development Environment (HiLDE), a cycle-accurate testing framework for performing FPGA-in-the-Loop simulations. The evaluation of the vMAGIC tools to generate a bitstream has yet to be demonstrated, along with any discussion on how designs can be scaled.

The work by [70] covers similar explorations as this work by building productivity

tools for a designer. Whereas this work focuses on high performance computing and scaling a design to an increasing amount of resources, Koch et al. investigate run-time reconfigurable FPGA systems. Both works focus on productivity as a metric ; however this work has not explored run-time reconfiguration. Instead, this work looks at the same problem (resource utilization) in a different way. This work assumes available resources will increase and as a result, the designer needs tools to enable productive utilization of the additional resources. Conversely, run-time reconfiguration assumes FPGA resources are scare and the current application is too large to fit in the available resources. Therefore, run-time reconfiguration changes the functionality of part, or all, of the FPGA depending on the current state of the application.

3.4 Performance Monitoring

This section highlights performance monitoring related works. While not all are specific to FPGAs this offers a brief glimpse of some relevant projects. In [71], the Owl-system monitoring framework is presented that uses hardware monitors in the FPGA fabric to snoop system transactions on memory, cache, buses etc. This is done to avoid the performance penalty and intrusiveness of software based monitoring schemes. While the same system can be adapted for fault detection as well, it essentially only monitors software behavior from hardware and not the applications within the FPGA itself. Along similar lines [72] has developed a performance analysis framework for FPGA-based systems. This does an automated application specific run-time measurement to provide a more complete view of the application core's performance to the designer. Source level (HDL) instrumentation is used to parse code and insert logic to extract desired data at runtime. TimeTrial [73, 74] explores performance monitoring for streaming applications at the block level keeping it language-agnostic, especially when dealing with different platforms and clocks. FPGAs have also been used to emulate and speed up netlist level fault injection and fault monitoring frameworks for building resilient system on chips [75].

CHAPTER 4: DESIGN

The focus of this work is an investigation into the feasibility of productively scaling designs to an increasing amount of available resources. To address this a *Systematic Design Analysis* flow (SDAflow) is presented, aimed at analyzing a design and determining its capability to be scaled across available of resources. The result of which is to help a designer focus on optimizing the design for a small set of resources rather than continually redesigning as the amount of available resources increases or to indicate performance drains that the designer should consider improving. Therefore, the goal is to show continued performances gains across the available resources without requiring the designer to manually redesigning the system for the resources.

Furthermore, with the widely distributed resources it is possible to provide a designer with a set of potential configuration candidates that can exploit different characteristics of the cluster. This is due to the fact that an FPGA cluster is tightly integrating on-chip processors, on-chip and off-chip memory, and networking. With the addition of the custom high-speed network the cluster of nodes presents as a single large FPGA rather than a large group of small FPGAs.

The Systematic Design Analysis flow considers transistors to be a cheap commodity, especially within an FPGA where the resources are reusable. Therefore, the proposed solution is targeted to efficiently use the resources; however, it is understood that a fully hand tuned system designed for the cluster would likely outperform any automated process. This stems from the fact that today's synthesis, placement and routing tools for FPGAs are still maturing and as was the case with the first software compilers it is possible for a designer to construct a system on a cluster to yield higher performance. The cost is the additional engineering effort to design the system. As

a result, a system that uses an acceptable amount of additional resources (say 18 FPGAs compared to an ideal 16 FPGAs) while achieving acceptable performance is allowed. That is, the cost of the additional resources will amortize the cost of engineer's additional efforts.

4.1 Systematic Design Analysis

While on the surface the proposed solution appears to be the creation of an automated tool, the actual solution is to develop an approach to take existing designs, analyze their resource usage, performance, interfaces, and develop a set of candidate configurations that the designer can choose from when implementing the design on a larger cluster of FPGA resources. While several automated tools have been developed as part of this work which are targeted to save the designer time and engineering effort, these tools are to work in concert with the various stages of the design flow. The Systematic Design Analysis flow (SDAflow), as is graphically illustrated in Figure 4.1, would ideally help automate the process and simplify the design space exploration to produce the candidate set of estimated performance and efficiency numbers, but such automation is not a requirement, nor necessarily an artifact of this proposed work.

The discussion of the Systematic Design Analysis flow is presented next which consists of the following structure. Each stage in the flow (Project Assembly, Component Synthesis, etc.) is described in four subsections. The first subsection is an overview of the stage and its purpose within the flow. Next, the supporting tools created for the stage are described. In the third subsection short examples are given about how the tools are used. Finally, a summary of the stage is presented. Furthermore, Figure 4.1 includes the corresponding subsection for each stage to help identify graphically where each stage fits in the Systematic Design Analysis flow.

4.1.1 Project Assembly

The first stage assembles all of the source HDL files into projects for both the single node synthesis and for the static HDL profiling stages. The single node synthesis

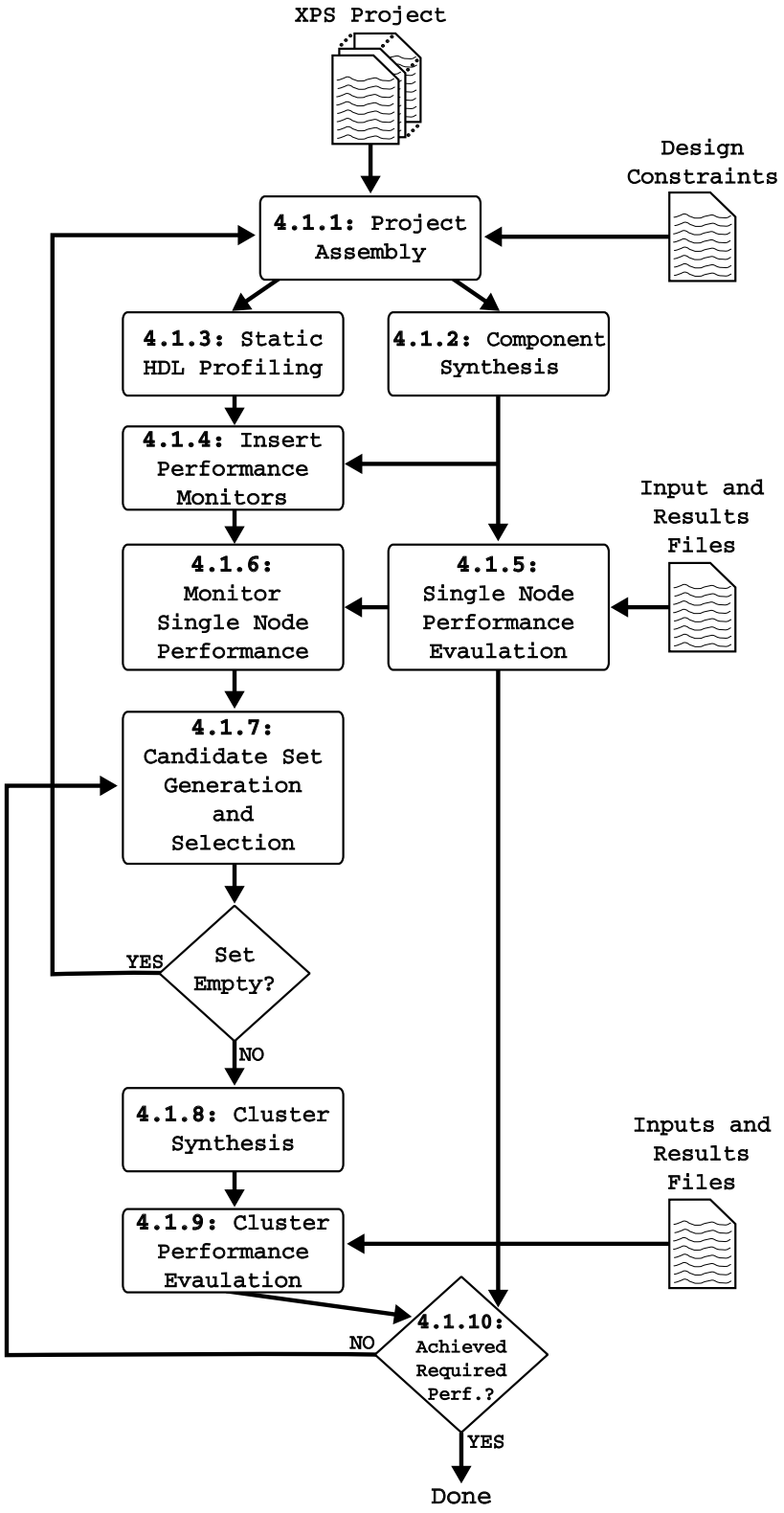


Figure 4.1: A high-level diagram of the Systematic Design Analysis tool flow

project can support the Xilinx Synthesis Tool (XST) [18]. If the designer has specific configurations pertaining to their hardware design, it would be at this stage to include those configurations. Otherwise the project will be optimized for resource utilization. The project for the static HDL profiling stage deconstructs the hardware design into subcomponents to be profiled individually and as a whole.

This stage also requires the designer give information regarding the node types and their configuration. This can be supplied via a configuration file which needs to at least specify the FPGA types and network configuration. This can become quite complex; however, with an FPGA integrated on a development board it is easy to know the pin configuration and other board specific information. For now this work will focus on a small set of boards (Xilinx ML410, Xilinx ML510, Xilinx XUPV5) and the user can specify which node is of which board type. Then as more board types are added to the boards set, it can be scaled to cover a wider range of nodes. More details pertaining to the resources and configurations used in this work can be found in Chapter 5.

4.1.1.1 Supporting Tools

The `Generate Systems` tool has been developed to aid the designer with the Project Assembly stage. The tool is written in Python and operates on the designer's initial base system. A requirement for this tool to function is that the base system be developed with Xilinx Platform Studio (XPS) [76]. The Generate Systems tool then executes the Xilinx Platform Generator tool (PlatGen) [63] which creates the XST synthesis project files for all of the components as well as the top-level HDL wrapper files. Once PlatGen completes, the Generate Systems tool parses the synthesis project files and assembles a new directory, currently called the `RCS_TOOLS` directory. Within the `RCS_TOOLS` directory the file hierarchy shown in Figure 4.2 is generated.

4.1.1.2 Tools Example

The tool is run from with the XPS project directory with the following command:

```
RCS_TOOLS
|-- data
|   |-- system_parse_results
|   |   |-- <component parse results>
|   |   |-- <component resource utilization>
|   |   |-- system_parse_results
|   |   `-- system_resource_
utilization
|   `-- performance_monitor_src
|       |-- head_perf.c
|       `-- head_perf.h
|-- hdl
|   |-- <hdl_libraries_vw_xy_z>
|   `-- work
|       |-- <top-level entity HDL wrappers>
|       `-- system.vhd
|-- netlists
|   |-- <generated component netlists>
|   |   `-- <intermediate component NGC files>
|   |-- <intermediate system NGC files>
|   |-- <intermediate system NGD files>
|   |-- <intermediate system MAP files>
|   |-- <intermediate system PAR files>
|   |-- <intermediate system BitGen files>
|   `-- system.bit
|-- scripts
|   |-- <component XST projects>
|   |   |-- <component SCR file>
|   |   |-- <component PRJ file>
|   |   `-- <component Makefile>
|   `-- Makefile
`-- Makefile
```

Figure 4.2: file hierarchy created by Generate Systems tool

```
$ ./generate_system.py system.xmp
```

The Xilinx Microprocessor Project (XMP) file contains specific information about the XPS project. The Generate Systems tool uses the XMP file to identify the Microprocessor Hardware Specification (MHS) file and any hardware core repositories locations. The MHS file is then used by the PlatGen tool to generate the synthesis scripts and top-level entity HDL wrappers. Running PlatGen for a Xilinx ML410 development board with Virtex 4 FX60 FPGA:

```
platgen -p  -xc4vfx60ff1152-11 -lang vhdl -lp /pcore_repository
```

The Generate Systems tool calls PlatGen internally, so it is unnecessary for the designer to do so manually. After PlatGen completes the Generate Systems tool finishes after generating the `RCS_TOOLS` directory structure.

Within the `data` directory are subdirectories that will eventually store results from the next two stages of the Systematic Design Analysis flow, namely `Component Synthesis` and `Static HDL Profiling`. More details about these stages and their supporting tools will be reported in Section 4.1.2 and Section 4.1.3 respectively. The `hdl` directory contains the HDL source files for the entire project listed in their respective libraries. This also includes the top-level entity HDL wrappers and the project's top-level entity, `system.vhd`. The `scripts` directory includes all of the individual component's XST synthesis project files that are necessary to create the component's netlist. The `SRC` file is the synthesis script where specific details about the component are set. Specifically, the device type, synthesis optimization flags, and generic parameters are set here. When migrating to other devices as part of the Systematic Design Analysis flow these scripts are modified accordingly. The `PRJ` file lists all of the source HDL files that are part of the component relative to the `HDL` directory. The `netlists` directory will store the synthesize netlist as part of the Component Synthesis stage as well as the intermediate MAP, PAR, and Bitstream Generation (BitGen) files which is part of the Single Node Evaluation stage.

4.1.1.3 Summary

In short, the Project Assembly stage prepares the original hardware design that is to be evaluated by the Systematic Design Analysis flow for scalability and performance capabilities. The work done in this stage is automated by the Generate Systems tool. Once completed a new project directory will be created, `RCS_TOOLS`, which along with the original XPS project and the input and result files for performance evaluation are all that is needed for the Systematic Design Analysis flow to continue to function.

4.1.2 Component Synthesis

To establish a baseline for a performance comparison against the future cluster implementation, a single node system is synthesized. The single node consists of the original hardware core plus any necessary system infrastructure, such as buses, bridges, processors, and memory controllers. The system will have already been run through the Project Assembly stage and the necessary synthesis scripts and the `RCS_TOOLS` directory will already be in place.

Since it is possible for the design to be optimized for a variety of configurations, this stage requires the designer to supply any specific configuration options that would normally be present in their single node design. This process may be iterated over with different parameters if deemed necessary. After synthesis of the components has finished, the synthesis reports will provide important information regarding the system, including subcomponents, resource utilization, timing requirements, and behavior. All of the configuration information and synthesis results are passed onto the performance monitor insertion stage.

4.1.2.1 Supporting Tools

Three tools have been developed to specifically support the designer in the component synthesis stage. These tools will enable to the designer to automatically synthesize, parse, and aggregate the individual component utilization, resource utilization, and timing information data.

The first tool is the `Iterative Component Synthesis` tool, which is written in Python. Its purpose is to run the synthesis scripts that were created during the Project Assembly stage with the Generate Systems tool. Synthesis can be a time consuming process depending on the host machine and the synthesis tools; therefore, these tools can increase a designers productivity by running together automatically without intervention from the designer, allowing the designer to work on other projects.

The second tool is the `Parse Component Synthesis Reports` tool, also written in Python, and it is used after all of the components of the system have been synthesized. After synthesis, each component generates a synthesis report file (SRP) which contains a wealth of information regarding the specific component. This tool's job is to collect this data and to feed it forward to other stages in the system. Specifically, the component utilization and resource utilization are fed to the Static HDL Profiling stages because they identify registers, FIFOs, Block RAMs, and finite-state machines (FSM) in addition to all of the components and subcomponents to are part of the specific components hierarchy.

The third tool used is the `Aggregate System Synthesis Data` tool, again written in Python, used to aggregate all of the data collected as part of the Parse Component Synthesis Reports data. The entire system data, which includes the interconnects, processors, memory controllers, and network interfaces are identified in addition to the designer's custom compute cores. The aggregate information will be available going forward through the rest of the Systematic Design Analysis flow.

4.1.2.2 Tools Example

To quickly demonstrate the functionality of the Component Synthesis stage and the tools developed for the stage, the following short example has been created. This example follows after the Project Assembly example where an existing system already has been run through the Generate Systems tool and the `RCS_TOOLS` project directory

has been created.

Beginning in the `RCS_TOOLS` directory the Iterative Component Synthesis tool runs XST on each of the component scripts found in the `scripts` subdirectory. Each subdirectory also contains a Makefile which can be used to simplify synthesis execution. The `make` process can be further expedited through the use of the `jobs` flag (-j) at the commandline which exploits a multi-core processor's capability to execute multiple synthesis jobs simultaneously. Individually, each component is synthesized through the Makefile by:

```
$ make -C synthesis/collatz_core_0_wrapper
```

which in turn calls:

```
$ xst -ifn collatz_core_0_wrapper.scr -ofn collatz_core_0_wrapper.srp
```

Alternatively, the entire system can be synthesized at once in parallel through the use of the `-j` flag:

```
$ make -C synthesis -j $NUM_PROCS
```

To use the Parse Component Synthesis Reports tool on a specific component all that is necessary is to run the tool on the generated synthesis report file):

```
$ ./parse_comp_srp.py collatz_core_0_wrapper.srp
```

which generates the report in Figure 4.3, an abridged report is shown for brevity.

The Parse Report tool can also be used to generate a list of the associated components in the system. For example, Table 4.1 lists the resource utilization for each individual component in the Collatz Core system which includes eight Collatz Cores, one secondary bus (PLB 0), and one bridge. This information is useful in understanding the resource utilization of the interconnect infrastructure and to help identify scalability. This data is also analyzed during the Candidate Set Generation stage.

```
collatz_core_0_wrapper:
  Number of Slice Registers:  233 / 69120
  Number of Slice LUTs:       323 / 69120
  Minimum period:             4.738ns
  Maximum Frequency:          211.071MHz
  Components:
    1.  plb_slave_attachment
    2.  user_logic
    3.  collatz_kernel
        Registers (collatz_kernel):
          64-bit register for signal <n>
          32-bit register for signal <steps_i>
        FSMs:
          <FSM_0> for signal <fsm_cs>
```

Figure 4.3: sample output of Parse Report tool

Table 4.1: sample system utilization output from Parse Report tool

Component	**Slice FF**	**4-LUTs**	**BRAMs**	**Timing (MHz)**
Collatz Core 0	233	400	0	188.893
Collatz Core 1	233	400	0	188.893
Collatz Core 2	233	400	0	188.893
Collatz Core 3	233	405	0	188.893
Collatz Core 4	233	400	0	188.893
Collatz Core 5	233	400	0	188.893
Collatz Core 6	233	400	0	188.893
Collatz Core 7	233	405	0	188.893
Collatz Core Total	**1864**	**3210**	**0**	—
PLB 0	131	241	0	333.333
PLB Total	**293**	**777**	**0**	—
Bridge 0	696	1014	0	164.204
Bridge Total	**696**	**1014**	**0**	—
System	**4114**	**6148**	**32**	**159.033**

Finally, the Aggregate System Synthesis Data aggregates this data into a Python data structure to be used in figure stages of the design. Specifically, the tool uses a Python module known as `Pickle` for serializing and de-serializing a Python object structure [77]. This allows the results to be quickly exported and imported by other tools without requiring re-parsing or re-running the tools.

4.1.2.3 Summary

Overall, the Component Synthesis stage is designed to understand how the original system utilizes the available resources. This allows the Systematic Design Analysis flow to create candidate configurations to use the remaining resources in an efficient manor. The work done in this stage is automated by the three tools: Iterative Component Synthesis, Parse Component Synthesis Report, and Aggregate System Synthesis Data and the aggregate data is passed through a Python `pickle` to the future stages in the flow.

4.1.3 Static HDL Profiling

During this stage, each HDL file will be analyzed to collect information to be used by the Performance Monitors Insertion stage. As the hierarchy of the hardware design is analyzed this stage will collect the types of interfaces between cores, state machine information, and will try to identify latency and bandwidth sensitive signals and components. HDL profiling is similar in principle to software profiling (i.e. gprof) in that all of the critical information is first collected and then when the system is run it can provide runtime performance information. Section 4.3 covers HDL profiling in more detail. Synthesis information will also be used from the Component Synthesis stage to aid in this stage and the supporting tools.

4.1.3.1 Supporting Tools

The Static HDL Profiling stage uses three tools to more autonomously profile the original design. These tools have all been written in Python. Furthermore, the aggregated synthesis information from the Component Synthesis Stage is imported

```
## Entity Name of VHDL File
self.entity = ""
## Architecture of VHDL File
self.arch = ""
## Libraries: List of Libraries/Use together
self.libraries = []
## generics: Key = Generic Name / Value = (Generic Type, Init Value)
self.generics = {}
## ports: Key = Port Name / Value = (Port Direction, Type, Init Value)
self.ports = {}
## signals: Key = Signal Name / Value = (Signal Type, Initial Value, Size)
self.signals = {}
## components: Key = Component Name / Value = Token for Component
self.components = {}
## fsms: Key = FSM Name / Value = FSM States
self.fsms = {}
## constants: Key = Constant Name / Value = (Constant Type, Value)
self.constants = {}
```

Figure 4.4: VHDL Parser Python data structure declarations

for these tools and the resulting profiling data is output for future stages in the Systematic Design Analysis flow.

The first supporting tool is a more general tool that is actually used by several future tools in this work. The `VHDL Parser` tool parses a given VHDL file and generates a Python pickle with an internal data structure consisting of several dictionaries, lists and strings. The Python declaration of these data structures can be seen in Figure 4.4. The decision to create a custom parser rather than use existing parsers is due to the tight integration with the other tools in the Systematic Design Analysis flow.

The second tool is the `System Parser` which uses the VHDL Parser and iteratively parses the VHDL files in the system. This includes the different interfaces, such as bus slave, bus master, direct memory access, and LocalLink [30]. These interfaces are fed to the Performance Monitor stages to insert and monitor their efficiencies. Also identified are the specific states of a FSM, software addressable registers, and FIFO and BRAMs.

The last tool is the `Parse PCORE` which performs a slightly different function. It parses the existing Xilinx PCORE directory's Microprocessor Description (MPD),

```
Entity: collatz_kernel
Architecture: imp
Generics:
Ports:
  Inputs:
        clk   : std_logic
        rst   : std_logic
        clear : std_logic
        valid : std_logic
        din   : integer
  Outputs:
        rdy   : std_logic
        rfd   : std_logic
        steps : integer
Registers:
        steps_i : 32-bit
        n       : 64-bit
FiniteState Machines:
        FSM0 : FSM_TYPE
               States: idle, calc, done
```

Figure 4.5: sample output of VHDL Parser tool

Peripheral Analysis Order (PAO), and Black Box Description (BBD) files along with any Xilinx CoreGen (XCO) project files. This information is needed as part of the Candidate Configuration Generation and Selection stages. The data is again stored in Python pickle files. All of the pickle files for this stage are stored in the `RCS_TOOLS/data/system_parse_results` directory.

4.1.3.2 Tools Example

A brief demonstration of these tools is performed on the `collatz_core` that will be discussed in Section 6.4. The Collatz core consists of three VHDL files with a bulk of the work contained within the `collatz_kernel.vhd`. Parsing this VHDL file with the VHDL Parser is done by:

```
$ ./vhdl_parser.py collatz_kernel.vhd
```

which produces the output in Figure 4.5.

From this it can be seen that the Collatz Kernel has five input signals, three output signals, two registers and one FSM with three states.

The System Parser applies the VHDL Parser to all of the VHDL files in the system and collects comparable information. Missing from the Collatz example are components, such as BRAMs and FIFOs, which this core does not have. For brevity the output of the entire System Parser is excluded from this document; yet, the tool is called by:

```
$ ./parse_system.py RCS_TOOLS
```

Finally, the PCORE Parse tool is used to parse the Xilinx specific MPD, PAO, BBD and XCO files. Much of the data collected from these files should correctly overlap with the VHDL and System Parser tools. For example, the MPD file contains the different ports and generics for the top-level entity of a hardware core. Yet, in addition to this data, more general information can be collected. This includes default values for the generics that would typically overwrite the generics at the top-level entity. Also included are the various Xilinx support bus interfaces, such as the PLB Slave, Master, LocalLink and NPI to the MPMC. This collected data is especially useful for the Candidate Set Generation stage which will apply modifications to these files to change the size of components like FIFOs as well as changing the various interfaces used.

```
$ ./parse_pcore.py collatz_core_v1_00_a
```

In addition, the PCORE Parser also can be used to generate a GUI for generating synthesis scripts for the specific PCORE. This is useful for intermediate development by a designer. Figure 4.6 depicts this tool for the top-level entity of the Collatz core.

4.1.3.3 Summary

The Static HDL Profiling stage is designed to further understand how the original system's internal components interface as well as to combine with the Component Synthesis stage's data for a more thorough understanding of the system. This allows the Systematic Design Analysis flow to create candidate configurations to use the

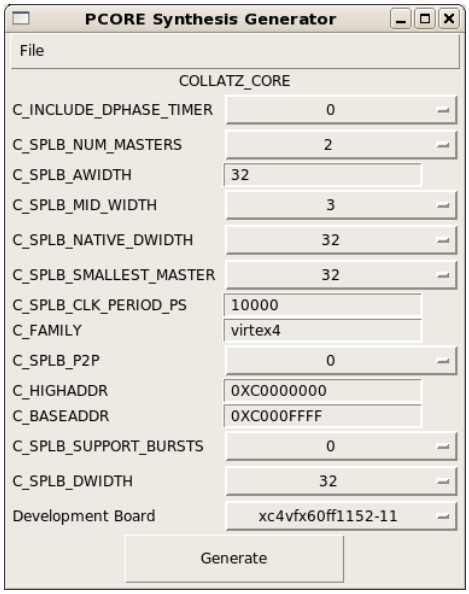


Figure 4.6: Parse PCORE Synthesis Generator GUI

remaining resources efficiently as well as generating new configurations with different interconnections and interfaces. The work done in this stage is completed with the three tools: VHDL Parser, System Parser, Parse PCORE and the data structures are passed through a Python `pickle` to the future stages in the flow.

4.1.4 Insertion of Performance Monitors

The purpose of the performance monitor cores are to gain runtime information such as bandwidth, latency and utilization about the single node system. Section 4.4 details the Performance Monitoring Infrastructure and lists the proposed monitor cores in more detail. This stage relies upon the information generated in the Static HDL Profiling stage to identify and recommend the appropriate monitors. The recommendation is done through a support tool to be discussed within this Section.

When a performance monitor is created there is a set of criteria that must also be included to allow the recommendation to take place. For example, there is a PLB Slave Interface performance monitor which specifically monitors reads and writes to the hardware core's slave registers. During profiling all signals are identified; however, until these signals are matched against a list of predetermined signals, there is no

specific way to identify when those signals are being written to. Another example considers finite state machines. Once an FSM has been identified by the system, it is trivial for the respective performance monitor to be recommended for insertion.

Over all, these monitors are designed to collect performance data with minimal invasion of the system. This can be verified during the next stage, the Monitor Single Node Performance stage. Even though recommendation of monitors is presented to the designer, no tools exist to automatically insert specific monitors into the HDL. Furthermore, it is possible for the designer to add custom monitors in addition to those recommended. While a tool to insert these monitors does not exist, a tool to insert the necessary Performance Monitoring Infrastructure exists and will be discussed within this section. In addition, collecting the performance monitoring data has also been automated through a separate tool. The performance monitor results are relied upon heavily during future Candidate Set Generation and Selection stages.

4.1.4.1 Supporting Tools

To aid in the selection of specific performance monitors the `Performance Monitor Recommendation` tool is used which parses the data collected by the preceding stages in the Systematic Design Analysis flow and given a list of available performance monitors, can recommend specific monitors. Presently, the recommendation does not directly insert these monitors in their respective components places. Instead, the monitors are recommended and the designer determines the best way to insert them into the core. This is to preserved the structure of the original hardware core. Future work may determine this structure is less sensitive and therefore insert these recommended tools accordingly. An example of the Performance Monitor Recommendation tool on the Collatz core is as follows:

```
$ ./perf_mon_recommend.py collatz_core.vhd
```

with an output shown in Figure 4.7.

```
Recommended Performance Monitors:
Top-Level Entity: collatz_core
collatz_core:
|-- plb46_slave_single_i
|   `-- NONE
|-- user_logic
|   |-- Utilization Monitor
|   |-- Interrupt Timer Monitor
|   `-- PLB SLV IPIF Monitor
`-- collatz_kernel
    `-- Finite State Machine Profiler
```

Figure 4.7: sample output of Performance Monitor Recommendation tool

The second tool used in this stage is the `Performance Monitor Insertion` tool, which purpose is to insert the performance monitoring infrastructure in both the component being evaluated and to add the supporting infrastructure to the rest of the base system. While a more detailed description of the performance monitoring infrastructure is presented in Section 4.4, a brief summary is presented here. In total the infrastructure consists of the following:

- system monitor hub
- side band network interface(s)
- context interface
- performance monitor hub
- performance monitor core(s)

The performance monitoring infrastructure is further depicted in Figure 4.8 where monitors can be inserted for the PLB slave and master bus interfaces, the BRAMs and FIFOs and the FSMs.

It is with the insertion of the performance monitors that the original design is first modified. The Performance Monitor Insertion tool must modify both the hardware PCORE, but also the original hardware design's XPS base system. The design is modified at the PCORE and MHS level and a second project is created in order to maintain the original as a baseline for comparison. To demonstrate this, the follow

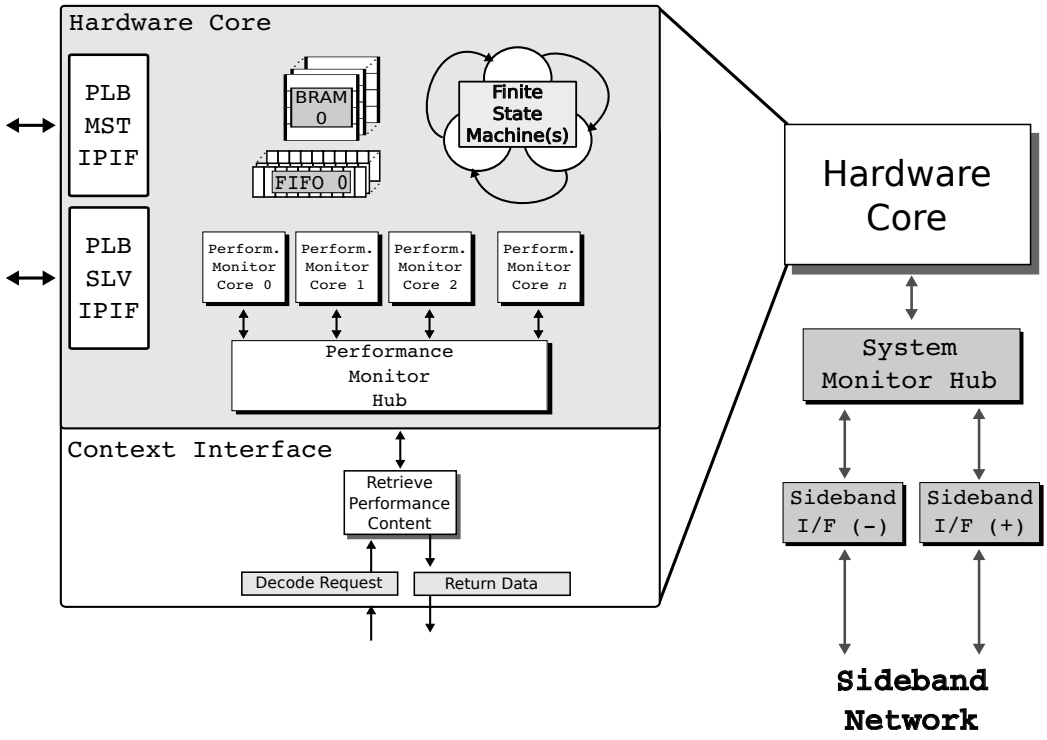


Figure 4.8: block diagram of inserted performance monitor infrastructure

example is presented.

4.1.4.2 Tools Example

Once the specific monitors have been identified the component instances must be inserted into their respective VHDL files. The performance monitors are centrally located in the PCORE repository of Systematic Design Analysis under:

`performance_monitors_v1_00_a`

A designer simply needs to include the library for the monitors in their VHDL file and instantiate the monitor. For example:

```
library performance_monitors_v1_00_a;
use performance_monitors_v1_00_a.all;
```

then all that is left is to insert the monitor with a component instance, for example the VHDL instance of the FSM performance monitor inserted as part of the Collatz kernel is shown in Figure 4.9. In Figure 4.10 a block diagram of the entire Collatz

```
fsm_mon : fsm_profiler_perf_mon
  generic map (
    CHIPSCOPE       => FALSE,
    NUM_STATES      => 4
  )
  port map (
    ila_ctlr        => (others => '0'),
    clk             => clk,
    rst             => rst,
    enable          => prof_enable,
    -- FSM State Value
    cur_state       => fsm_value,
    -- Monitor Interface
    ll_rx_sof_n     => p1_rx_sof_n,
    ll_rx_eof_n     => p1_rx_eof_n,
    ll_rx_src_rdy_n => p1_rx_src_rdy_n,
    ll_rx_dst_rdy_n => p1_rx_dst_rdy_n,
    ll_rx_data      => p1_rx_data,
    ll_tx_sof_n     => p1_tx_sof_n,
    ll_tx_eof_n     => p1_tx_eof_n,
    ll_tx_src_rdy_n => p1_tx_src_rdy_n,
    ll_tx_dst_rdy_n => p1_tx_dst_rdy_n,
    ll_tx_data      => p1_tx_data
  );
```

Figure 4.9: example of VHDL instance for FSM Profiler Monitor

Core connected via the crossbar switch is shown with performance monitors inserted (more details regarding this implementation can be found in Chapter 6).

After inserting the performance monitor cores, the performance monitor hubs is added. The hub is responsible for aggregating all of the performance monitor cores data and sending it to the monitoring head node in a single requests, therefore reducing the number of ports added to the hardware core. The hub is inserted with one port per performance monitor core. Again, this performance monitor hub is inserted manually at present since the number of monitor cores is non-deterministic presently. With the performance monitor cores and hub inserted, the Performance Monitor Insertion tool modifies the PCORE and the XPS project file to add support for the context interface, the system monitor hub, and the side band network channels that are part of the entire monitoring infrastructure.

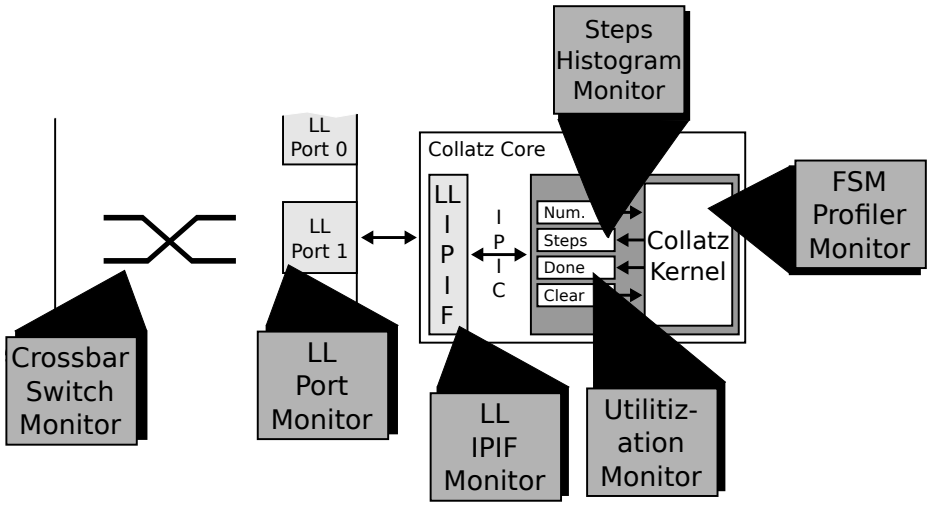


Figure 4.10: example of performance monitors inserted in Collatz Core

4.1.4.3 Summary

To summarize, the Performance Monitor Insertion stage is responsible for recommending monitoring cores and inserting the performance monitor infrastructure into the original design, thereby creating a new design to be evaluated during the Monitor Single Node Performance stage of the Systematic Design Analysis flow. Recommendations are made based off of Static HDL Profiling data and modifications to the PCORE and XPS project are automated for the designer.

4.1.5 Single Node Performance Evaluation

Once the single node system has been synthesized it will be run on a single node to gather performance numbers. This stage may not be necessary if the performance numbers can be provided by the designer. Of course, these numbers are application specific as are the specific tests which generate the numbers, but it is reasonable to require the designer to supply both a benchmark of inputs and expected results files so that the scalability performance can be compared. Presently, no tools exist to collect the data due to the variable nature of each application. Instead, the designer must supply a mechanism to retrieve the data for comparative analysis or run the tests and identify the specific results. For the case studies in this work, the most

common performance metric being evaluated is execution time for a predetermined set of inputs. The case studies list the single node performance as part of their respective results and analysis sections.

4.1.5.1 Supporting Tools

There are no specific tools developed for the Single Node Performance Evaluation stage. Instead, the designer is required to provide the input data and test application(s) to evaluate the system. Then the designer needs to specify what is the comparison metric (speedup, execution time, resource utilization, etc.) for future candidate configuration comparisons.

4.1.5.2 Tools Example

There are no examples to demonstrate.

4.1.5.3 Summary

This stage should be the most familiar to the designer since the functionality of the system and runtime performance is being evaluated without any modifications to the system. As a result, the designer should quickly move past this stage, or possibly bypassing it completely if existing data for comparison already exists.

4.1.6 Monitor Single Node Performance

With the monitors added to the single node system it is now possible to synthesize the design and run it to collect the runtime data. This stage relies on the same testing infrastructure that was provided by the designer during the Single Node Performance Evaluation stage. The performance monitors should not impact the performance when compared to the original system since the monitoring infrastructure operates independently of the original design's processor. In addition to the performance companions, each performance monitor's data is retrieved and is used to determine, if possible, the overall scalability based on the resource utilization, memory bandwidth, network bandwidth, and I/O bandwidth.

Since each performance monitor's outputs may differ, the goal of this stage is to

present the results to the designer in the form of a print statement per each element of the monitor core's data structure, the details of which are described shortly. For the initial set of performance monitors listed in Section: 4.4 the data collected can be forwarded to the next stage, Candidate Set Generation, in order to recommend to the designer the configurations that may yield the best performance. However, because it is possible for the designer to insert new monitors that the Systematic Design Analysis is unaware of, it is currently not possible to completely automate the analysis of the retrieved data for all monitors. As a result, the designer will be responsible for analyzing custom monitors output data.

4.1.6.1 Supporting Tools

To assist collecting monitoring data, the `Performance Monitor Collection` tool has been written in Python. Ths tool serves two purposes. First, the tool parses the newly created system to identify all of the performance monitors inserted in the system. This is done after running the Generate Systems tool. Starting with the top-level entity in the system.vhd file, the tool walks through and identifies the System Monitor Hub and all components connected to it. From each of these connects it is possible to locate the Context Interface (CIF) in each component. The CIF connects to the Performance Monitor Hub which aggregates all of the Performance Monitor Cores. Once collected the tool then is able to perform its section purpose. At this point the Performance Monitor Collection tool is able to assemble the entire system's performance monitoring data structure for the head node to use to collect the runtime data. This data struct is a C struct. In addition to the generation of this struct are C subroutines for the head node to automatically collect and report (currently via `printf()`) each monitor's data back to the designer. This data can also be stored to a file through the Reconfigurable Computing Systems Network Filesystem when the head node is running Linux. In this case, the Performance Monitor Collection tool can be used to parse the results. Presently, the more common mechanism to store

```
typedef struct {
  unsigned int command;
  unsigned int node_id;
  unsigned int num_cores;
  unsigned int num_mons_0;
  sw_0_counter_perf_mon sw_0_counter_perf_mon_0;
  sw_0_fsm_perf_mon sw_0_fsm_perf_mon_1;
}perf_core;
```

Figure 4.11: generated perf_core C struct

the monitoring data is through the use `minicom` which when started with the `-c` flag and a filename captures the terminal's output to the file.

4.1.6.2 Tools Example

To briefly demonstrate the Performance Monitor Collection tool, the following example with the Smith/Waterman is used. The details of the Smith/Waterman can be found in Chapter 6. Once the performance monitor cores and supporting monitoring infrastructure have been inserted during the previous stage the Performance Monitor Collection tool is run:

```
$ ./collect_perf_mon.py RCS_TOOLS
```

First, the tool parses all of the HDL in the system to specifically identify the aforementioned monitoring components. For this simple Smith/Waterman example two monitors are identified, a counter for the PLB slave IPIF and the FSM profiler. The search for the monitors is possible due through top-down analysis of the components and their respective sub-components. Finally, the tool produces two files, `perf.h` and `perf.c` which can be compiled into the head node's C application to retrieve the performance data. An example of the top-level C struct for the Smith/Waterman core is shown in Figure 4.11. Both of the performance monitor cores also have C structs created to store their performance monitor data. These structures are shown in Figure 4.12 and Figure 4.13.

The Performance Monitor Collection tool also generates the required access functions needed by the head node to collect the performance monitor cores data and

```
#define FSM_0_NUM_STATES 29
typedef struct {
  unsigned int mon_num_words;
  uint64 state[FSM_0_NUM_STATES];
}sw_0_fsm_perf_mon;
```

Figure 4.12: generated PLB slave IPIF C struct

```
#define NUM_WR_CE 14
typedef struct {
  unsigned int mon_num_words;
  uint64 Bus2IP_WrCE[NUM_WR_CE];
}sw_0_counter_perf_mon;
```

Figure 4.13: generated C FSM profiler C struct

store them into its respective struct. Then, the data can be written or to a file and/or printed out to the terminal. Figure 4.14 gives an example of the output for a performance monitor that the designer would see during the data collection and reporting of the Performance Monitor Collection tool. The specific output will be discussed in more detail as part of the Smith/Waterman case study in Chapter 6.

Table 4.2 demonstrates in table form the FSM performance monitors output for the Collatz Core. The output is given to the designer in similar form to the GNU Profiler [78] tool that many software developers are familiar with. Presently, the Systematic Design Analysis flow does not autonomously use this data for future candidate configurations generation, it is mostly provided for the designer to quickly understand how the FSM behaves when running with real input data. The designer could then ideally evaluate the necessity to improve the individual state or entire state machine's performance.

Table 4.2: sample output in table form of FSM Profiler

State	**Time**	**Percentage**
IDLE	22.695 μs	2.07 %
CALC	54.601 μs	4.98 %
DONE	1019.103 μs	92.95 %

```
Smith/Waterman Core 0 SLV PLB IPIF Write Registers Monitor:
  perf.sw_0_counter_perf_mon.Bus2IP_WrCE[0]:   186
  perf.sw_0_counter_perf_mon.Bus2IP_WrCE[1]:   0
  perf.sw_0_counter_perf_mon.Bus2IP_WrCE[2]:   2095745
  perf.sw_0_counter_perf_mon.Bus2IP_WrCE[3]:   2095838
  perf.sw_0_counter_perf_mon.Bus2IP_WrCE[4]:   2095838
  perf.sw_0_counter_perf_mon.Bus2IP_WrCE[5]:   2095838
  perf.sw_0_counter_perf_mon.Bus2IP_WrCE[6]:   2095838
  perf.sw_0_counter_perf_mon.Bus2IP_WrCE[7]:   2095838
  perf.sw_0_counter_perf_mon.Bus2IP_WrCE[8]:   0
  perf.sw_0_counter_perf_mon.Bus2IP_WrCE[9]:   2095838
  perf.sw_0_counter_perf_mon.Bus2IP_WrCE[10]:  0
  perf.sw_0_counter_perf_mon.Bus2IP_WrCE[11]:  0
  perf.sw_0_counter_perf_mon.Bus2IP_WrCE[12]:  0
  perf.sw_0_counter_perf_mon.Bus2IP_WrCE[13]:  0
```

Figure 4.14: sample output of PLB slave IPIF WrCE performance monitor

4.1.6.3 Summary

Overall, the Monitor Single Node Performance stage is responsible for parsing the system and determining which monitors have been added. Once identified this stage also is responsible for generating the monitor collection software infrastructure used by the head node to retrieve the performance monitoring data. The Performance Monitor Collection tool has been developed to perform these time consuming tasks for the designer. The data can be printed out for the designer to see in real time and/or stored to a file for analysis in future stages of the Systematic Design Analysis flow.

4.1.7 Candidate Set Generation and Selection

Due to the configurable nature of FPGAs it is possible to assemble many different systems with different interconnections, memory interfaces, and network interfaces. Furthermore, each application may behave differently when scaling the design beyond a single core and/or node. Therefore it is necessary to collect a set of potential configuration candidates. When scaling the design beyond a single node this work will focus on initially supporting only homogeneous node types (each bitstream is the same for each node), but as more complex applications demand a more diverse set of nodes, it will become a design space consideration to explore each possible

configuration and its final placement in the cluster. This is only necessary when the design has been finally run on the cluster and it does not meet the projected performance or the user supplied cutoff.

The various configuration options that are currently under consideration include the following. Modifications to the internal components, such as increasing or decreasing internal FIFOs. Replacing bus master interfaces that access off-chip memory with direct memory access interfaces. Inserting centralized direct memory access when an application has been identified as using programmable I/O to transfer data to a hardware core. Converting a design between a direct connect network, crossbar switch, and a shared bus interconnect to improve cross-section bandwidth of the interconnect or to reduce resource utilization. Utilizing the AIREN high-speed/low latency integrated on-chip/off-chip network in place of traditional fast Ethernet. Migrating designs to use different FPGA devices, as would commonly be the case when updating a design to use the newly available FPGAs.

Any one of these modifications could require weeks, if not more, of the designer's time to implement. Therefore, the as part of the Systematic Design Analysis flow several supplemental tools have been developed to assist the designer in implementing these alternative designs. While some tools do require the designer to manually modify the software application, say in order to support central DMA, all of the tools have been constructed such that minimal manual hardware modifications are required. The tools which require hand modifications will be noted.

While the designer can include any number of custom candidate configurations to the Systematic Design Analysis flow, this section will highlight how the previous stages have specifically contributed to the candidate set generation. One important note, just as with the Insert Performance Monitors stage, this stage does not automatically call the following tools to build the configurations. Instead, the top-level tool performs an analysis of the collected data and recommends which configurations

might yield the best performance. This allows the designer an opportunity to add or remove configurations based on their own knowledge or experience with the design.

4.1.7.1 Supporting Tools

Several tools have been developed as part of this stage to improve the designer's overall productivity by reducing the amount of time a designer must spend manually creating and evaluating these configurations. Moreover, while the tools presented here are the culmination of a vast amount of hardware engineering knowledge and experience, it would be trivial to create and insert additional tools or to modify existing tools as the Systematic Design Analysis flow and the target hardware mature.

The first tool is the `Candidate Configurations Recommendation` tool, which is responsible for parsing all of the static HDL profiling data, component synthesis resource utilization data, performance monitoring cores data, and the single node performance evaluation data. Presently, this data is all stored (with the exception of the single node performance evaluation data) in Python `pickle` data structures for easy accessibility. The Candidate Configurations Recommendation tool (demonstrated in the next section) makes recommendations based on specific parameters that are set as follows.

Resource Utilization During the Component Synthesis stage individual component resource utilization is identified. The tool analyzes the resources to determine if any resources are still available. In the event sufficient resources are available (a software programmable parameter which is currently set to at least 10% of the resources remaining) the tool will recommend replication of the specified hardware core. Replication involves instantiating additional parallel copies of the hardware core in the design. This is currently supported in designs that are connected through the Processor Local Bus or connected through a crossbar switch. The replication occurs through the use of the `PLB Replicate PCORE` tool and `Crossbar Switch Replicate PCORE` tool. To identify which kind of connection the design uses, the static HDL pro-

filing data is used. If there are insufficient resources available, the design is migrated to a larger FPGA device. This currently includes the Xilinx ML410, Xilinx ML510 and Xilinx XUPV5 discussed in more details in Chapter 5. Migration is automated through the use of the `Migrate to ML510` and `Migrate to XUPV5` tools. It is possible that a design does not automatically migrate due to missing Xilinx CoreGen project files or other netlists that are specific for one device. In these cases the designer must complete the migration process manually.

Static HDL Profiler While part of the static HDL profiling information is used to determine the interconnection types for buses and crossbar switch, the profiler also provides insight into how the system is connected to off-chip memory and the network. For example, if a component is connected to the PLB as a bus master, then it may be possible to use the `Bus Master to DMA` tool. However, since it is not known with static HDL profiling along if the component only access off-chip memory or other components on the bus, performance monitoring data is also necessary. The profiler helps the Candidate Configuration Recommendation tool identify performance monitoring data described next.

Performance Monitoring Data Using the resource utilization and HDL profiled data to understand which kind of components are being used and how they are interfaced, the Candidate Configuration Recommendation tool can isolate specific monitoring data and make specific candidate configuration recommendations. For example, if a hardware core has a FIFO that has been shown to never reach full capacity, it is possible to replace this FIFO with a smaller FIFO which would consume fewer resources, through the `FIFO Replacement` tool. Likewise a FIFO size could be increased if it is determined that it is full for a long duration of time. Similarly, if a design is found to have the processor use programmable I/O to read data from off-chip memory and write it to the hardware core the `Central DMA Insertion` tool can insert the necessary hardware into the design and recommend how to interface

with this new core through sample C code.

The designer ultimately runs these tools manually as part of this work, but a fully automated system could be created if so desired. The benefit of the designer's contribution at this stage is in the event of an error, the designer can address the errors per each configuration as part of each tool. While an error would ideally not occur it must be considered probable.

This stage also presents the designer some initial information based on these configuration candidate sets. The designer can modify the candidate sets and/or input additional design constraints based on the preliminary results. For example, this stage can present the designer numbers in terms of scalability or performance, such as "it will take x resources to scale to y cores" or "the performance of scaling to y cores will be..." To be useful, the designer needs to have feedback at all levels and to decide how much effort is needed to obtain the desired performance.

4.1.7.2 Tools Example

As part of this work nine tools have been developed to aid the designer in scaling the design. These tools are used throughout the four case studies presented in Chapter 6. This section will describe each tool in more detail along with providing simple functional examples. This section does not further describe the Candidate Configurations Recommendation tool as it was already described in sufficient detail.

PLB Replicate PCORE Tool Through the use of the PLB Replicate PCORE tool a design created with Xilinx Platform Studio can quickly generate large scale designs were a specified PCORE is replicated along with any necessary buses and bridges. The tool operates at the MHS level of the XPS project currently. The tool is written in Python and is used by:

```
$ ./plb_replicate_pcore.py system.mhs collatz_core_0 127
```

where the first parameter is the MHS file, the second parameter is the core to be replicated and the last parameter is the number of times to replicate the core. For

this example the collatz_core_0 instance will be replicated 127 times resulting in 128 parallel Collatz cores. Furthermore, because the PLB is limited to supporting at most 16 slave hardware cores, bridges are added along with supplemental buses automatically. The bridges are also configured with the correct address ranges. This saves the designer time and reduces the changes for an error due to the complex address ranges.

Crossbar Switch Replicate PCORE Tool Similar in concept to the PLB Replicate PCORE tool, the Crossbar Switch Replicate PCORE tool can scale a hardware core across the crossbar switch. Unlike the PLB, the crossbar switch does not require the use of additional switches or bridges. Instead, the crossbar switch simply interfaces with each new hardware core until maximum number of ports is reached. Presently, an additional tool has been developed to increase the number of ports of the crossbar switch, called `Scale Crossbar Switch`. To first scale the crossbar switch all this is necessary is:

```
$ ./scale_crossbar_switch crossbar_switch_v1_00_a 96
```

which augments the existing crossbar switch to now support 96 input/output port pairs. While it is feasible to scale indefinitely, the resource utilization of the particular FPGA will set an upper limit on the scalability of the crossbar switch. Once the number of ports has been increased it is possible to scale a particular hardware core already connected to the crossbar switch:

```
$ ./crossbar_switch_replicate_pcore.py system.mhs collatz_core_0 95
```

which scales the number of Collatz cores to a total of 96.

PLB to Crossbar Switch Tool If a design has been identified as a PLB based system through static HDL profiling the PLB to Crossbar Switch tool can replace the bus infrastructure with a crossbar switch. This tool not only must augment the existing base system, but also modify each PCORE that is connected to the PLB. This

is accomplished through the replacement of the PLB Slave IPIF component which is responsible for translating PLB bus requests to IPIC signals the hardware core interfaces with. In its place is the LL IPIF component which translates the crossbar switch signals to the IPIC signals. No modifications are required by the designer to the original hardware core. In fact, the PLB to Crossbar Switch tool modifies the core at the HDL, MPD, and PAO levels. For example, augmenting a hardware core:

```
$ ./plb_to_ll_pcore.py collatz_core_v1_00_a
```

allows the hardware core to be included in crossbar switch designs. The next step is to use the PLB to Crossbar Switch tool to modify the existing base system, replacing the PLB and exchanging the bus interfaces for crossbar switch interfaces:

```
$ ./plb_to_crossbar_switch.py system.mhs
```

FIFO Replacement Tool One simple tool that has been heavily utilized is the FIFO Replacement tool. Its purpose of increasing or decreasing the size of a FIFO while retaining the rest of the FIFOs parameters enables a designer to focus less on the fine grained resource utilization issues and more on functionality. For example, a common design practice is to build a system an introduce buffers on the interface; however, the sizes of these buffers is not always optimized or even removed later even if they are completely unnecessary. Therefore, this tool can be used in concert with the FIFO Utilization performance monitor to help generate candidate configurations with larger or smaller FIFOs. In one case this tool enables four additional hardware cores to be instantiated on what was considered a fully utilized design (BLAST case study). This tool relies upon the existence of the Xilinx ChipScope project file for the FIFO. If the project file does not exist it is possible to have the tool generate a new project FIFO given parameters from the designer:

```
$ ./modify_fifo_depth.py sw_core_v1_00_a fifo.xco 1024
```

will change the depth of the FIFO to 1024 elements. Presently, changing the FIFO width is not permitted. There is also an ability to use distributed RAM (in LUTs) in place of BRAM; although none of these FIFOs have been evaluated as part of this work.

Bus Master to DMA Tool A significant performance and productivity tool is the Bus Master to DMA tool. When the static HDL profiler identifies the hardware core as having a PLB master interface the Insert Performance Monitor stage can insert a monitor to evaluate the utilization of this interface. Specifically the monitor evaluates the number of requests issued, the amount of data transferred, the address ranges transferred to and from, and the latency of the transfers. From all of this information it is possible to identify if the interface can be replaced with a direct memory access interface. Similar to the LocalLink IPIF replacement, the Bus Master to DMA tool extracts the PLB master IPIF component and replaces it with a DMA component. The hardware core still issues requests to the DMA component as if it were interfacing with the bus; however, the DMA component is directly connected to off-chip memory which offers lower latency and higher bandwidth to memory. Most importantly, the designer does not need to modify the existing hardware core or the base system, it is all handled by the Bus Master to DMA tool. For example:

```
$ ./modify_pcore_bus_master.py blast_v1_00_a
```

will generate a new top-level entity with the existing PLB slave interface and all internal components except in place of the PLB master interface will be the DMA component. Furthermore, the DMA signals needed to interface with the memory controller are also inserted. In addition to the HDL modifications, the tool modifies the MPD and PAO files as well. To modify the MHS file an additional Python script has been created which walks through and replaces the components master PLB interface and exchanges it for the new DMA interface along with augmenting the memory controller to connect to the DMA channel of the hardware core:

```
# Hardware Core Instance
BEGIN sw_core
 PARAMETER INSTANCE = sw_core_0
 PARAMETER HW_VER = 3.00.b
 PARAMETER C_BASEADDR = 0xc8c00000
 PARAMETER C_HIGHADDR = 0xc8c0ffff
 BUS_INTERFACE SPLB = plb
 # AGS: Interface to MPMC DMA Channel
 BUS_INTERFACE XIL_NPI = BUS_MASTER_NPI
 PORT mpmc_clk = DDR2_SDRAM_mpmc_clk_s
END

BEGIN mpmc
 PARAMETER INSTANCE = DDR2_SDRAM
 PARAMETER HW_VER = 4.03.a
 PARAMETER C_NUM_PORTS = 4
 # AGS: Added PIM for DMA Channel
 PARAMETER C_PIM3_BASETYPE = 4
 PARAMETER C_PIM3_DATA_WIDTH = 32
 BUS_INTERFACE MPMC_PIM3 = BUS_MASTER_NPI
 # AGS: Remainder of Instance Truncated
```

Figure 4.15: sample of MHS created by Bus Master to DMA tool

```
$ ./modify_base_bus_master.py system.mhs
```

will produce changes found Figure 4.15.

Central DMA Insertion Tool An improvement on the performance of programmable I/O, where the processor performs the entire request, is to use a DMA controller. Xilinx provides a *Central DMA Controller* [79] as a core in its EDK IP Core Repository. The processor issues a DMA request to the controller which in turn handles the DMA transaction on behalf of the processor. Figure 4.16 depicts the layout of such a system. The DMA controller can support:

- Processor initiated read from memory and write to hardware core
- Processor initiated read from hardware core and write to memory
- Hardware core initiated read from memory and write to hardware core
- Hardware core initiated read from hardware core and write to memory
- Hardware core to hardware core transfers

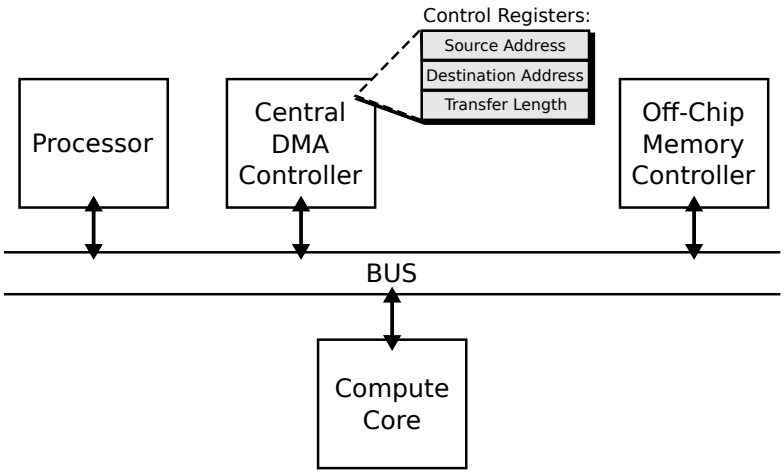


Figure 4.16: central DMA controller offloads memory transactions from processor

It is important to understand that *memory* in the term "direct memory access", does not necessarily mean off-chip memory. In fact, transfers between hardware core's on-chip memory is also possible. While evaluating the performance monitor data it if a component is found to have numerous writes to a specific register it is possible that the designer has introduced Programmable I/O into the system. To improve performance a central DMA core can be added to the system and requests to memory can be off-loaded from the processor. At this time, the designer must manually modify the software application to take advantage of this design. However, the Central DMA Insertion tool can take care of inserting the necessary hardware into the existing design along with providing boilerplate code for standalone C access. For example:

```
$ ./insert_central_dma.py system.mhs
```

will produce instantiate the Central DMA core in the MHS file and produce the standalone C code needed to access the core which includes the necessary C structures and access patterns.

Migrate to FPGA Tools Two migration tools have so far been developed to take existing designs and migrate the systems to other FPGAs. These two tools are the Migrate to ML510 and Migrate to XUPV5 tools. Both of these tools assume the

migration occurs from the Xilinx ML410 development board. How this tool works is by moving specific cores to the new system. For example:

```
$ ./migrate_ml510.py system.xmp
```

will parse the XMP file and identify the location of the MHS files and any subsequent repositories. Then the ML510 base system will be constructed that resembles the ML410 base system (matching as best as possible memory ranges, peripheral devices, and overall structure). If a particular component cannot be migrated, the user will be notified. The purpose of this tool is to aid the designer when newer FPGA devices become available. Rather than start the design from scratch, these tools can help a designer quickly run on the new FPGA. It is then possible for the designer to analyze the performance through the use of the Performance Monitoring Infrastructure and tune the performance as part of the Systematic Design Analysis flow.

4.1.7.3 Summary

As part of the Candidate Set Generation stage nine tools have been developed to aid the designer in exploring a wide space of different configurations. Parameters used in the generation of these configurations include component synthesis resource utilization results, static HDL profiling results, and runtime performance monitoring data. The Top-level tool is the Candidate Configurations Recommendation tool which simply analyzes the aforementioned results and prompts the designer with the various candidate configuration options. The designer can then select a tool to use to generate the specific candidate configuration and synthesize the design for evaluation, as will be discussed in the next stage, Cluster Synthesis.

4.1.8 Cluster Synthesis

For designs that consist of homogeneous nodes, a single project file is generated based on the selected candidate and synthesized for the cluster. This process is similar to the single node synthesis stage at the beginning of this design analysis flow. For heterogeneous nodes, one project design per node type is required. This may increase

the build time for the cluster; however, the execution time of the analysis is not under evaluation with this research.

4.1.8.1 Supporting Tools

No new tools have been developed for cluster synthesis. Instead, the Project Assembly stage's Makefiles are used, even if the design is a heterogeneous design requiring multiple different synthesis runs, the Makefiles handle.

4.1.8.2 Tools Example

There are no tools to demonstrate.

4.1.8.3 Summary

The Cluster Synthesis stage synthesizes one of the candidate configurations to be evaluated on the cluster of a available resources. This stage is repeated for each configuration that is to be evaluated. It is possible for these stages to be performed in parallel per each configuration if the compute resources are available.

4.1.9 Cluster Performance Evaluation

Once the design has been synthesized and a bitstream has been created the design is run on the cluster. Again, it is up to the designer to supply a benchmark for input and a results file to compare for the expected output. The complicated portion of this stage is that the resulting performance is based on the designer's supplied benchmarks. Caution should be taken by the designer not to provide insufficient tests. Furthermore, a design that is not optimized to scale or an application that is not ideally suited for having multiple parallel hardware cores may not fully take advantage of the cluster of available resources.

4.1.9.1 Supporting Tools

While no supporting tools have been developed in general, there are a number of tools that aid in the preparation, delivery, and setup of the cluster of resources. These tools work on top of the existing FPGA Session Control (FSC) [80] which is part of the Reconfigurable Computing Cluster project's Spirit cluster. FSC provides

```
# Build ACE File (assume homogeneous design)
make ace
# Request 32 FPGAs
make request n00-n31
# Power on 32 FPGAs
make up n00-n31
# Upload ACE Files to Nodes
make upload system.ace n00-n31
# Boot ACE File on each node
make boot n00-n31
# Once Node Boots Insert Device Drivers
make insmod all_dd n00-n31
```

Figure 4.17: sample commands used during Cluster Evaluation stage

commandline access to control the cluster remotely and has been developed to power on/off nodes, upload FPGA configuration files (Xilinx ACE files), and boot the nodes into a specific ACE file. At this point the FPGA is programmed and any supplemental software applications are loaded into memory. A generalized Makefile has been constructed to enable a designer to quickly perform these tasks with minimal effort. Once the system is booted, if it is running Linux additional steps may be necessary before the application can be run and the performance can be evaluated. Most often, this includes the insertion of device drivers needed by the application to access the custom hardware accelerator cores. The Makefile also supports this, although the specific device drivers need to be identified by the designer. One specific application, BLAST, uses a custom Python application to perform a more verbose evaluation of the hardware. This application will be discussed in more detail as part of the BLAST case study in Section 6.2.

4.1.9.2 Tools Example

Figure 4.17 offers a brief example of using the Makefile to construct the ACE configuration file, request and power on the nodes, upload and boot the ACE file and insert the necessary device drivers. Other custom tools developed per each application as part of this work or preceding this work will be presented in each application's respective case study.

4.1.9.3 Summary

The Cluster Performance Evaluation stage is responsible for running the test applications over the newly generated configurations. The results are collected for analysis in the final stage of the Systematic Design Analysis flow.

4.1.10 Performance Analysis

With both the baseline (single node) and cluster performance numbers gathered a straightforward comparison can be made to determine if any performance gains were actually realized with the cluster. In the event the design satisfies the designer's requirements, the Systematic Design Analysis flow has completed its task and the designer is left with a design that meets the requirements. However, in the event the performance is below the anticipated performance system should return to the candidate state and select the next viable candidate. Performance information is returned to this stage as well to help improve the likelihood that the next chosen candidate will yield the expected performance. Finally, in the event that no configuration is found to meet the performance requirements, the flow should fail and terminate. Before doing so, the design analysis should provide the designer with the best configuration candidate and any information that could help the designer modify the original single core implementation to yield better performance during the next design analysis.

4.1.10.1 Supporting Tools

No tools have been created to aid in the performance analysis. It is the responsibility of the designer to fully determine if the candidate configuration has met the requirements for performance or any other metric.

4.1.10.2 Tools Example

There are no examples to demonstrate.

4.1.10.3 Summary

The final stage in the Systematic Design Analysis flow asks the simple question, "did the candidate configuration achieve the required performance?" It is up to the

designer to answer this question unless a specific metric (such as speedup) is provided. Of this analysis can be more complex than just comparing two numbers. Ultimately, this stage will either terminate or return back to the Candidate Set Selection stage where another candidate is selected and evaluated. Chapter 5 and Chapter 6 will discuss how the Systematic Design Analysis flow is to be evaluated as part of this work.

4.2 Tool Development

As Section 4.1 has shown, several tools have been developed to support the Systematic Design Analysis flow. This section aims to briefly highlight how and why some of these tools were developed. In total 22 tools, scripts and other forms of automation have been created as part of this work. By providing a designer with these tools several of the repetitious tasks can be avoided. Furthermore, analysis can be done in a simple turn key solution when collecting profiling and resource utilization data. Ultimately, the question that tool development boils down to is, why create tools? This section will cover three tools that have been heavily utilized as part of this work. Some code snippets will be included; however, all of the code will be available on the Reconfigurable Computing Systems Lab website [81].

4.2.1 Generate Systems Tool

The most utilized tool is the generate systems tool. Every project that is to take part in the Systematic Design Analysis flow passes through this tool. As mentioned earlier, this tool assembles the `RCS_TOOLS` directory out of which the remainder of the project runs. The generate systems tool is written in Python to take advantage of the `pickle` module to pass data structures efficiently between tools.

Before coding began, it was identified that the exising XPS project directory was going to be insufficient for the needs of the project. Most important was the reliance on the Xilinx MHS and MSS files and PlatGen to actually generate the top-level system entity. Since this work relies upon the fact that the design should be stable,

```
## Current Working Directory
cwd = os.getcwd()
## Run PlatGen (already parsed XMP for device specific data)
os.system(‘‘platgen -p  -xc4vfx60ff1152-11 -lang vhdl -lp /pcore_repository’’)
## Retrieve list of all Components in system (based on PRJ files)
components_list = get_components_list(cwd + ‘‘/synthesis’’)
## For each component create XST project
for component in components_list:
    ## For each component retrieve HDL and XST scripts
    component_hdl = copy_hdl_and_scripts(component, cwd)
    ## Genrate XST Project and Makefile
    generate_component_makefile(component_hdl, component)
## Generate Top-level Project
generate_synthesis_makefile(components_list)
```

Figure 4.18: code snippet for Genreate Systems tool

changes to the system should be made by the Systematic Design Analysis flow rather than the designer.

As a result, the tools needs to collect all of the source HDL such that it can be used for static HDL profiling and so that any modifications to the source code are done to copies of the design, not the original hardware core. Once called the generate systems tool will first execute PlatGen in order to generate the necessary synthesis scripts. Once complete, the tool parses the generated synthesis directory to collect and parse all of the component synthesis scripts. After which, each component's HDL and synthesis project files are copied into the `RCS_TOOLS` directory along with a Makefile that is used laster to synthesize the design. The synthesis project files are a valuable resource for the next few stages of the Systematic Design Analysis flow because all of the pertinent information that was stored in the MHS file and the PCORE's custom MPD and PAO files are now stored in these scripts.

Figure 4.18 provides a short section of code that makes up the systems tool. From this section the tool will run PlatGen and parse the synthesis scripts to identify the components in the system along with the path to all of the HDL used by the system. Finally, the top-level and subcomponent synthesis projects.

4.2.2 Performance Monitor Recommendataion Tool

Skipping head, the next tool to better understand is how exactly the performance monitors are identified. One challenging tasks for any hardware designer, but especially one who is just getting started in design is to understand what to monitor. Ideally, the designer will try and monitor everything. Ultimately, the designer needs a mechanism to recommend monitors based on the system as if a senior designer were the one making recommendations. The recommendation tool is written in Python and looks at both the static HDL profiling results along with the component synthesis results. Then, the tool runs through and tries to match those conditions to existing performance monitor parameters. For the purposes of this work, the monitors that are considered are as defined in Section 4.4.6.

From the Python pickle that was created by the static HDL profiler stage (VHDL Parser tool) the tool currently looks for three characteristics. First, it looks at each component's entity description to identify interfaces to the PLB as a slave or master, or to the crossbar switch via LocalLink. If detected the tool will recommend the user insert the specific monitor for that core. Next, each component instance is evaluated to identify if any are FIFOs for the FIFO utilization monitors. Third, all of the component's FSMs are analyzed and if any are found, the FSM profiler monitor is recommended. The tool also tries to locate memory interfaces (such as the multi-ported memory controller), network interfaces (such as Ethernet and AIREN), and even processors (PowerPC and MicroBlaze).

The goal for this work is to provide a set of monitors and an mechanism to identify when these monitors could be included in a design. Then, as this work matures additional monitors can be added. Presently, it is up to the designer to take the recommendation and actually insert these monitors. Another tool, the Performance Monitor Insertion Tool actually is responsible for inserting the monitoring infrastructure. Figure 4.19 shows a snippet of code used to identify the FIFO performance

```
# Open static HDL profiling pickle
hdl_data = load_pickle(‘‘hdl_prof_data.pickle’’)
# Retrieve Components from hdl_data
comp_dict = hdl_data[‘‘components’’]
# Walk through component dictionary to identify FIFOs
component in comp_dict:
    if (‘‘FIFO’’ in component.name):
        # Get FIFO declaration and all instances
        declaration,instance = comp_dict[component]
        ## Insert FIFO into Recommend FIFOs List
        for instance in instances:
            rec_fifo_list.append(instane)
```

Figure 4.19: code snippet for Performance Monitor Recommend tool

monitor.

4.2.3 PLB Replicate PCORE Tool

Finally, the PLB Replicate PCORE tool is used during the cadidate configuration generation stage to scale an existing PLB based design on a single node. This tool was developed to simplify the process a designer goes while trying to determine the exact number of cores that will fit in the design. Furthermore, a designer that is calculating the number of cores based on the available resources has to also deal with constraints such as the PLB can only support up to sixteen slaves, otherwise additional buses must be inserted. The PLB Replicate PCORE tool simplifies this process by managing the PLBs, bridges, address ranges, and generics. The tool was developed by observing designers replicating cores using a text editor and opening the MHS file. From there, the designer would simply copy and paste the hardware core the requisite number of times all the while adjusting the address range and other instance specific parameters. When scaling only to a few cores this manual process is fairly reliable. However, when the design scales to tens over a hundred cores this is a very time consuming process. This tool automates this process and handles the bus and bridge manipulations as well. Figure 4.20 shows a short code snippet of the tool as it considers the number of PLBs and bridges already in the system.

```
## Replicate Core - Add Buses and Bridges if Necessary
for i in range(0,replicate_number):
    ## Unique Instance Number per Core
    instance = ("%x" % i).upper()
    new_pcore_instance = pcore_instance[:-1] + instance
    if ((i % MAX_CORES_PER_BUS) == 0):
        ## Need to add new bus
        plb_slave = plb_slave[:-1] + bus_num
        new_fd.write("BEGIN plb_v46\n")
        new_fd.write(" PARAMETER INSTANCE = " + plb_slave + "\n")
        new_fd.write(" PARAMETER C_NUM_CLK_PLB2OPB_REARB = 100\n")
        ## Version specific to Xilinx 10.1 toos for now
        new_fd.write(" PARAMETER HW_VER = 1.03.a\n")
        new_fd.write(" PORT PLB_Clk = sys_clk_s\n")
        new_fd.write(" PORT SYS_Rst = sys_bus_reset\n")
        new_fd.write("END\n")
        ## Need to add new bridge to primary bus
        new_fd.write("BEGIN plbv46_plbv46_bridge\n")
        new_fd.write(" PARAMETER INSTANCE = " + bus_num + "_bridge\n")
        new_fd.write(" PARAMETER HW_VER = 1.01.a\n")
        ## Set Base/High Address Range
        new_baseaddr = base_addr[:-7] + bus_num.rjust(2,"0") + "00000"
        new_highaddr = base_addr[:-7] + bus_num.rjust(2,"0") + "FFFFF"
        new_fd.write(" PARAMETER C_RNG0_BASEADDR = " + new_baseaddr + "\n")
        new_fd.write(" PARAMETER C_RNG0_HIGHADDR = " + new_highaddr + "\n")
        new_fd.write(" BUS_INTERFACE SPLB = plb_prime\n")
        new_fd.write(" BUS_INTERFACE MPLB = plb_" + bus_num + "\n")
        new_fd.write("END\n")
    ## Finally Replicate PCORE on this bus
    new_fd.write("BEGIN " + pcore_name + "\n")
    new_fd.write(" PARAMETER INSTANCE = " + new_pcore_instance + "\n")
    new_baseaddr = base_addr[:-6] + instance.rjust(2,"0") + "0000"
    new_highaddr = base_addr[:-6] + instance.rjust(2,"0") + "FFFF"
    new_fd.write(" PARAMETER C_BASEADDR = " + new_baseaddr + "\n")
    new_fd.write(" PARAMETER C_HIGHADDR = " + new_highaddr + "\n")
    ## Write rest of Generics
    for generic,value in generics_list:
        new_fd.write(" PARAMETER " + generic + " = " + value + "\n")
    ## Write out bus interface
    new_fd.write(" BUS_INTERFACE SPLB = " + plb_slave + "\n")
    ## Write all of Ports
    for port,value in ports_list:
        new_fd.write(" PORT " + port + " = " + value + "\n")
    new_fd.write("END\n")
```

Figure 4.20: code snippet of the PLB Replicate PCORE tool

4.3 HDL Profiling

In order to identify portions of the code and/or components that can be moved to/from various places in the cluster the ability to perform HDL profiling is needed, but not just at the simulation level. This will give more information about runtime and help tune the system to the available resources. For example, consider how to best connect cores needing to communicate, should it be a bus, a mesh network, a crossbar switch? While each may work as an infrastructure, one may provide a significant performance and/or resource utilization advantage. Specifically the profiler will need to at least look at:

1. memory interfaces
2. network interfaces
3. latency sensitivity
4. resource utilization

4.3.1 Memory Interfaces

Static profiling/analysis of a hardware core's memory interfaces can provide a great deal of insight into the system. For example, if there are a number of on-chip memory interfaces versus off-chip memory interfaces, or if a number of concurrent accesses is required the location of the memory could be moved closer to farther from the compute core. The interface information will be fed into the performance monitors generator which can insert specific monitors for when the system is actually running.

As another example, if a system has a bus master interface and is issuing many requests to off-chip memory the profiler can identify this by looking at Bus Master transactions to the specific memory range which is provided at least by Xilinx in the MHS file and track the number of accesses. The designer may also be interested in the bandwidth, which can be used to calculate the peak bandwidth statically by the width of the data lines and the maximum operating frequency when the design is

synthesized, or the latency sensitivity of the design. Then it is possible to determine if it is more efficient to use a different interface, such as a direct connect to memory.

4.3.2 Network Interfaces

With large systems it is inevitable that hardware cores on one node will need to communicate with cores on another node. Perhaps not all applications will require this, but supporting a tight integration between the on-chip and off-chip networks will make it possible to do so. Ideally, the designer would want to know during HDL profiling if cores will be needing to communicate with other cores. This is kind of complicated because in some designs there may only be one core in the initial application. As a result it can become difficult to determine how these cores will need to communicate with each other.

Instead, what this work proposes to do is to use the network as a black box to the user and develop the appropriate interface based on the static profiling information. As a case in point for the case study for BLAST, Section 6.2, did not require modifications to the original BLAST core to use the network. Instead as part of the Systematic Design Analysis, small translation cores have been developed to allow the BLAST core to "think" it was interfacing directly with the Hardware Filesystem, even though it is connected directly to the network.

4.3.3 Latency Sensitivity

One important question that needs to be addressed when profiling the core is "how susceptible the core is to latency?" If we have core's that have little or no demands on latency, it becomes a good candidate to exist on a different node in the cluster. Alternatively, if two cores need to communicate (in a transaction-based interface, not streaming or pipelined) it is better to move them closer together. Latency is tied to utilization. If a specific set of signals used to communicate with another core are under utilized or a lot of time is spent waiting for data and/or ack signals, it is pretty apparent that the latency is having an effect on the core.

4.3.4 Resource Utilization

From the HDL/hardware profiling we also need resource utilization numbers. This will give us a better understand of how much a resource consumes and what the timing requirements are. By synthesizing a core and its subcomponents we can make broad claims of how many cores could possibly fit on a single FPGA. This gives us an upper bound of the scalability of a particular hardware core. It also lets us know how the underlying hardware core is using those resources. If we need a bunch of BRAMs to store data, but the data accesses are sparse or, as with BLAST, we can pipeline those accesses, maybe we can reserve parts of the chip of entire chips to support the BRAM/resources for that core.

Ultimately, this information is used to help narrow down where and how many performance monitors should be included in the design. This work will try and minimize the amount of effort it takes a designer to realize their design on a large cluster of FPGAs. If they can focus on making it work efficiently on a single node and we can then recommend ways to scale it across a cluster, perhaps that will save them time and engineering effort. Furthermore, if we can support debugging and performance analysis quickly then it will be easier for the system to be tested, modified and retested.

4.4 Performance Monitor Infrastructure

4.4.1 Overview

To explain the Performance Monitor Infrastructure, the high-level organization of a parallel computer composed of FPGA nodes (with an additional node used for system and performance monitoring) is presented. Next, the details of the protocol and the design of the IP cores used to support it are explained. Finally, a tool is described that has been developed to automate the process of inserting the necessary infrastructure into existing hardware cores. Performance monitoring is one facet of a large System Monitoring Infrastructure that is under development in the Reconfigurable

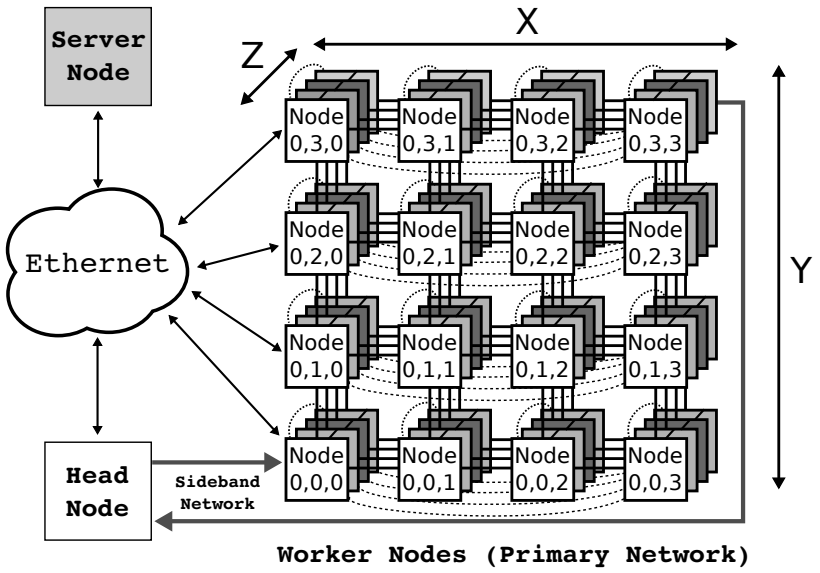


Figure 4.21: nodes and networks in cluster

Computing Systems lab [27, 28].

The monitoring infrastructure consists of several IP components that spans a variety of elements both in the system and across the cluster. The infrastructure to support the monitoring system is comprised of three types of nodes and two networks, as illustrated in Figure 4.21. Node types include a server node, a head node, and worker nodes. With the exception of the server, all nodes are Xilinx ML410 development boards with Virtex 4 FX60 devices. Currently, 64 FPGAs are connected via six direct-connect links to create the *Spirit* cluster [19] through the primary custom high-speed 3 dimensional torus network [23]. Two more links on the custom network board are used to form the sideband network.

4.4.2 Networks in Monitoring Infrastructure

There are two networks in the Spirit cluster. To differentiate between the two consider the following. The *primary network* is a high-speed, low latency integrated on-chip/off-chip network designed for the cluster of FPGAs. The network supports up to eight 4.0 Gb/s bi-directional channels with an $\approx$0.8 μs latency between nodes. The nodes can be wired to construct any conceivable eight channel network; however, the

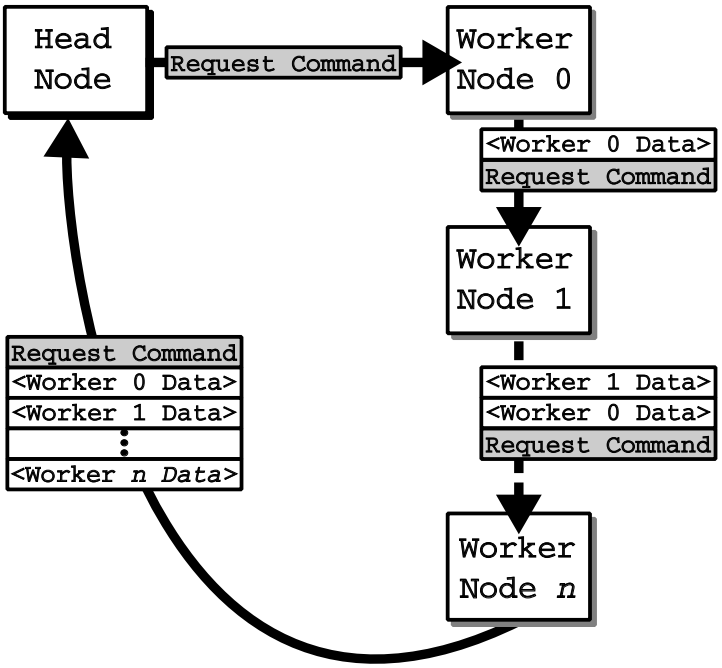


Figure 4.22: dataflow of commands issued by head node to worker node in system monitor's sideband ring network

current configuration uses six of the eight channels to create a 4-ary 3-cube (four nodes in each of the three dimensions) for a total of 64 nodes as is shown in Figure 4.21. It carries MPI messages via a custom Linux device driver and administrative TCP/IP traffic via another device driver. For the purposes of this work, the primary network is used during the cluster scalability evaluations.

The system monitor's *sideband network* uses the two remaining channels on the network card. Currently, the sideband network is connected in a ring, a decision to be noninvasive to the primary network. The head node issues requests across the ring and worker nodes respond to the requests by appending their data onto the end of the command packet. This can be seen in Figure 4.22. This work will rely on the sideband network in its ring configuration to retrieve performance monitor data from nodes and cores under evaluation as part of the Systematic Design Analysis flow.

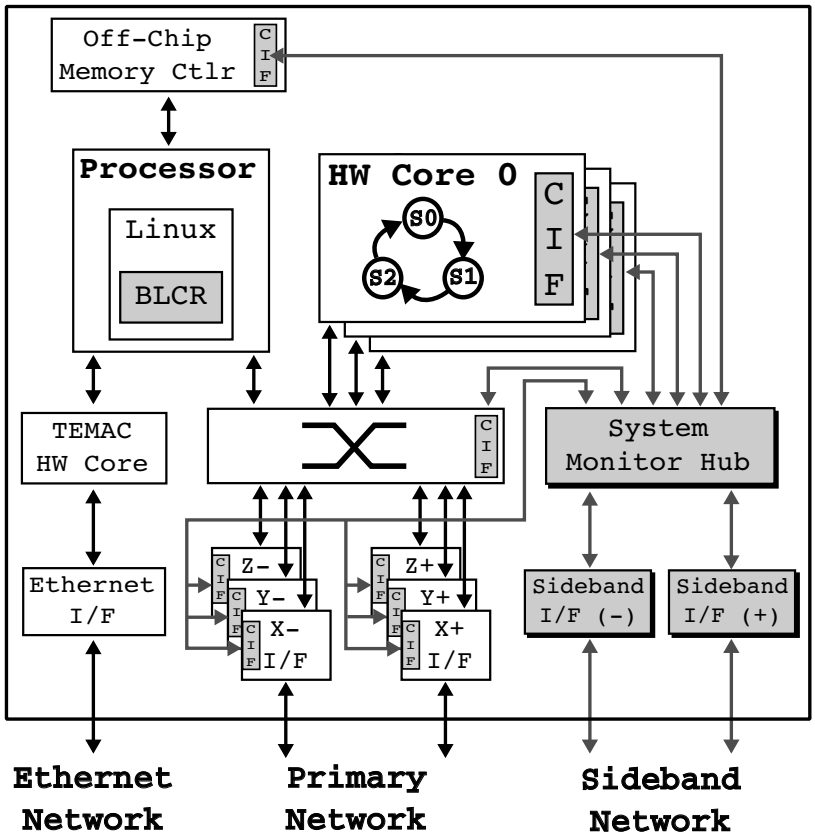


Figure 4.23: block diagram of FPGA node's monitoring system

4.4.3 System Monitor Hub

Figure 4.23 provides a high-level overview of the system-on-chip running on each FPGA and its integration with the *system monitor hub*. The system monitor hub is implemented as a generic VHDL entity with a configurable number of ports and also acts as an intermediary to decode incoming requests arriving on one of the two ports of the sideband network. The request is issued by the head node and contains the specific command to be performed, as is listed in Table 4.3. The system monitor hub then issues the subsequent request across the appropriate ports to the hardware cores being monitored. Each core replies to the request with the specific status, context or performance information. Finally, the system monitor hub returns the aggregated information out the sideband network, to the head node. The resource utilization of the system monitor hub can be seen in Table 4.5.

Table 4.3: system monitor request commands

Command	Description
GET_ALL_STATUS	Get all node's status data
GET_NODE_STATUS	Get one node's status data
GET_CORE_STATUS	Get one core's status data
GET_NODE_CONTEXT	Checkpoint node's context data
GET_CORE_CONTEXT	Checkpoint core's context data
SET_NODE_CONTEXT	Restart (load) node's context data
SET_CORE_CONTEXT	Restart (load) core's context data
GET_NODE_PERF	Get node's performance monitor data
GET_CORE_PERF	Get core's performance monitor data
GET_MON_PERF	Get one monitor's performance data
PAUSE_ALL	Pause all node's hw execution
PAUSE_NODE	Pause one node's hw execution
PAUSE_CORE	Pause one core's hw execution
START_ALL	Start all node's hw execution
START_NODE	Start one node's hw execution
START_CORE	Start one core's hw execution

Table 4.4: resource utilization of CIF

Resource Type	Occupied	Total Available	% Used
Number of Slice Flip Flops:	12	50560	0.024 %
Number of 4 input LUTs:	73	50560	0.144 %
Number of FIFO16/RAMB16s:	0	232	0.000 %

4.4.4 Context Interface

The Context Interface connects a hardware core to be monitored to the system monitor hub using the Xilinx LocalLink specification. The CIF is used to read status information (which can be used to identify the health of the component), get/set context information (for checkpoint/restart functionality), and read performance monitor information. For the purposes of this work only the performance monitor functionality will be evaluated. An example of the CIF is shown in Figure 4.24. The resource utilization of the context interface component is shown in Table 4.4.

To assist programmers wanting to include system monitoring into existing designs the CIF Generator, a graphical tool, has been created to instrument existing VHDL code. (Figure 4.25 shows a screen shot of the GUI.) By reading in a user's VHDL source code, the CIF Generator identifies the state-holding elements and generates a

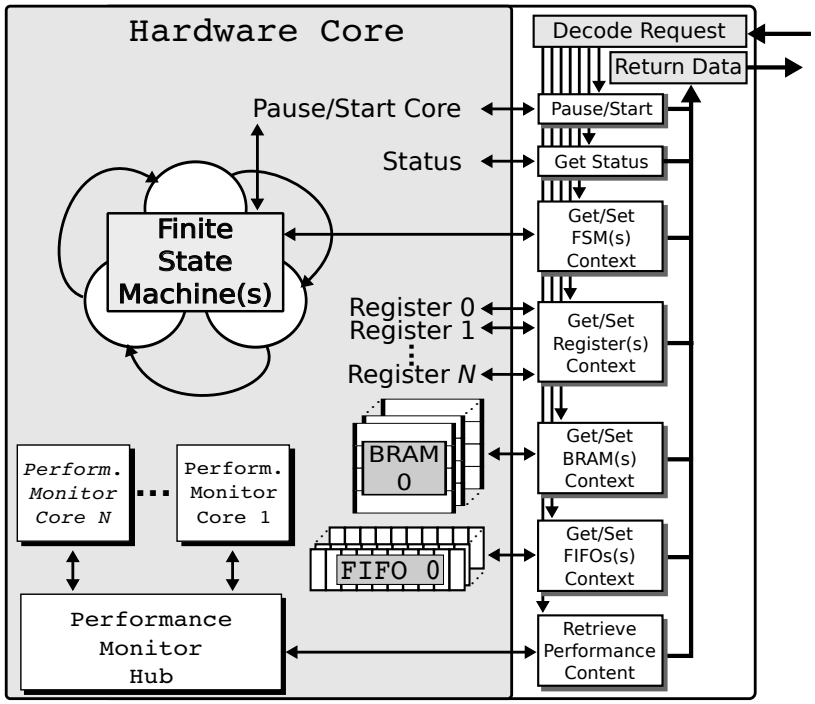


Figure 4.24: block diagram of hardware core and its CIF

list of signals, components, and possible state machine identifiers. Then, `xst` is run on the hardware core to further verify registers and FSMs. Next, the user selects the data to be saved during a checkpoint. The tool then generates a custom Context Interface wrapper for the accelerator core.

When finished, a new top-level entity is generated that encapsulates the newly created `cif.vhd` file, which interfaces with the system monitor hub, along with a modified version of the original hardware core to support access to the status, registers, BRAMs, FIFOs, and FSMs along with the necessary write enable signals for setting the context and pausing the execution. The CIF Generator reports the new hardware core's timing estimates in relation to the original core, although no optimizations are performed to improve timing.

4.4.5 Performance Monitor Hub

Connected to the Context Interface is the Performance Monitor Hub which in turn connects to performance monitor cores within the hardware core. The performance

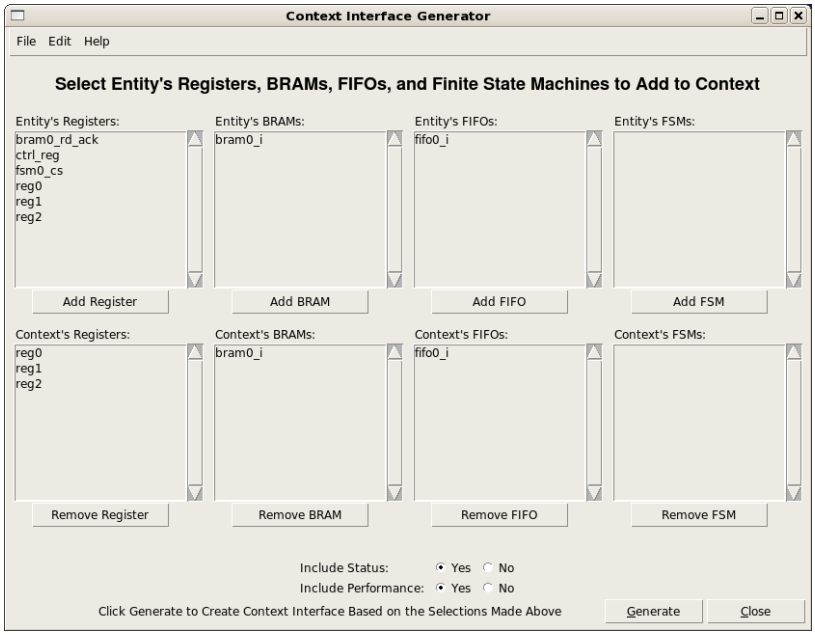


Figure 4.25: Context Interface Generator GUI

Table 4.5: system and performance monitor hubs resources

	System Monitor Hub		Perf. Monitor Hub	
Ports	**FFs (%)**	**LUTs (%)**	**FFs (%)**	**LUTs (%)**
1	105 (0.21%)	482 (0.95%)	14 (0.03%)	70 (0.14%)
2	105 (0.21%)	527 (1.04%)	17 (0.03%)	78 (0.15%)
4	109 (0.22%)	658 (1.30%)	21 (0.04%)	153 (0.30%)
8	111 (0.23%)	896 (1.77%)	21 (0.04%)	250 (0.49%)
16	113 (0.22%)	1345 (2.66%)	23 (0.05%)	419 (0.83%)

monitor hub is instantiated within the hardware core, an example of which can be seen in Figure 4.24. Requests for performance monitor data are issued by the Context Interface. The performance monitor hub then aggregates the performance monitor data and returns it back to the Context Interface. The performance monitor hub was based on the system monitor hub and is intended to be a light weight data collector, as can be seen in Table 4.5.

4.4.6 Performance Monitor Cores

Individual performance monitors serve the purpose of providing runtime performance data of the system under test. This is extremely valuable since the application's performance can differ dramatically based on the configuration of components, uti-

lization of resources, and even the different inputs into the application. For this work performance monitors will be added into the system based on the static HDL profiling and component synthesis information. The designer can also choose to add specific monitors if so needed. The system will be run with a single core and the performance monitors will collect specific information to help predict scalability across the cluster of resources.

Not every application will require the addition of every monitor core listed here and some applications may require the creation of new monitor cores. Therefore, this work is focused on the creation of an initial set of monitor cores and supporting infrastructure to support the inclusion of cores into the system. At the time that additional monitors are needed, the system should be flexible enough to quickly include these new cores.

The remainder of this section is as follows. First a useful debugging tool is introduced which makes inserting debugging components easier. Debuggers are one way to retrieve performance data of a running system; however, when the scale of the system grows the debugging infrastructure will not likely scale. Next, are several generic performance monitor cores which have been created and bundled into a VHDL library for easy inclusion into existing designs. Finally, this work includes some custom monitors that have been designed for specific interfaces or purposes. These new monitors are described last.

4.4.6.1 On-Chip Debugging

While not directly related to the idea of performance monitoring, on-chip debugging gives the designer key insight into the working system, running on the FPGA. Xilinx provides an application and tool set called ChipScope. To use ChipScope a design must add an integrated logic analyzer (ILA) which monitors a set of user specified signals to a specific component or hardware core. However, this can be a time consuming process. Therefore, this work includes the creation of a custom ChipScope

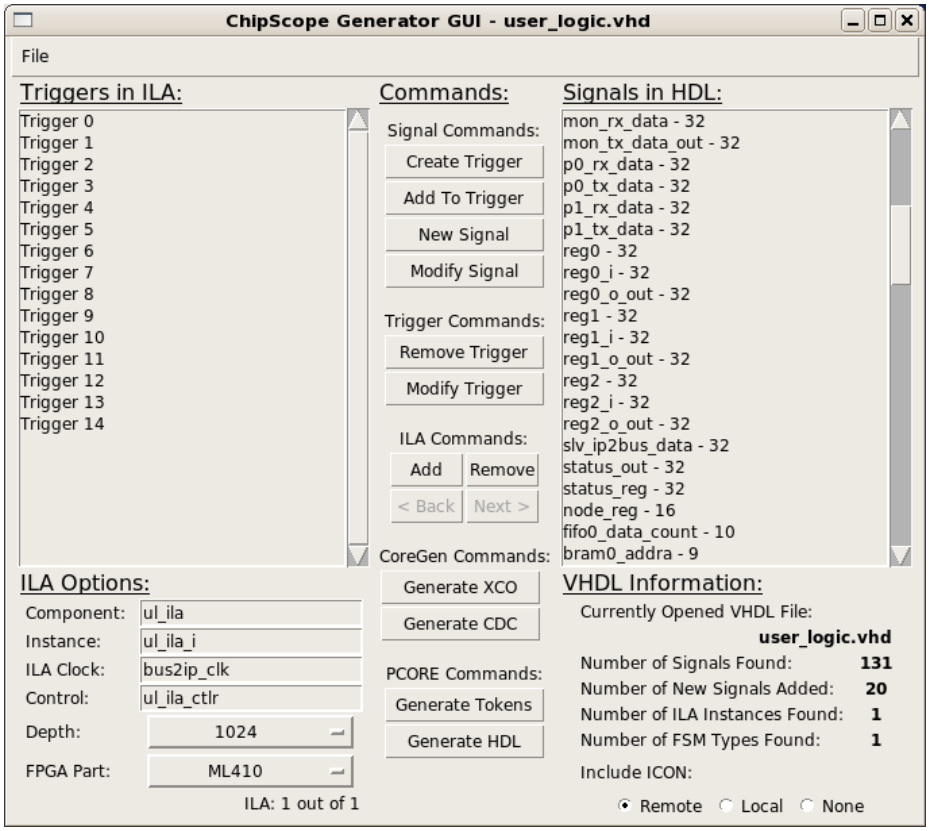


Figure 4.26: custom ChipScope integrated logic analyzer insertion GUI

ILA Insertion GUI to simplify this process. Figure 4.26 and Figure 4.27 are examples of the GUIs that have been created by the author for this purpose. This work aims to create a GUI for each type of monitor to provide the designer quick and easy access to inserting debugging cores and performance monitors into an existing system.

4.4.6.2 Counters and Timers

One of the most fundamental and frequently used monitors are of the type of counters and timers. With little resource utilization a timer and/or counter consists of a register and an adder. While the specifics of when the counter should be incremented or the timer should be started or stopped is application specific, a generic interface can be constructed to support a both the ease of use and the ease of integration into an existing hardware design.

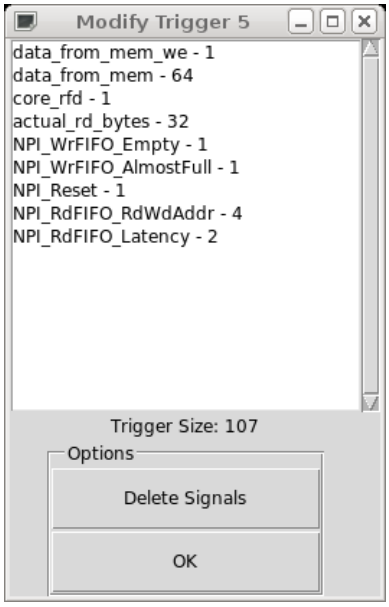


Figure 4.27: custom ChipScope modify trigger GUI

4.4.6.3 Pattern Detectors

A simple core with a FIFO that when triggered will start recording data that can be read out from a standard interface. It could be placed directly inline with an interface, such as a LocalLink connection, to allow for quick and easy debugging. Pattern detectors can also be used to assert flags or trigger interrupts in the event something occurs that is unintended. In these critical cases the system may need to know with little delay when such a event occurs and this monitor can do so without interfering with the running system.

4.4.6.4 Utilization Monitor

Another component is to calculate the utilization of some component. Similar in principle to the counter/time except the starting and stopping is coordinated by the component under test. This monitor is to be used to measure the impact on scaling the hardware designs with respect to the memory and network bandwidths. The utilization information can be combined with other information from the system to identify where the bottleneck exists to help tune the system accordingly.

4.4.6.5 FIFO Status Core

In many of the systems there are more than one FIFOs in a core and designers typically end up having to add a status register to figure out if any of the FIFOs are Full or Empty. In the case of network components this is important because if any of those FIFOs that are intermediate buffers are Full it means the system are missing data. Providing feedback about the buffers can help the system improve performance by tuning around the efficient utilization of these FIFOs.

4.4.6.6 Histogram Generator

The histogram generator core uses one or more BRAMs to store counters into bins based on the needs of the application. The monitor core will unobtrusively collect information and then after the application has run, a software process can read back the values in each bin and create a graphical histogram. Therefore, this monitor is actually a package of HDL, standalone C code to run on the processor to retrieve the data, and a Gnu Plot script to generate the histogram figure.

4.4.6.7 Latency Counter

The latency counter is a counter and a histogram together to give ideas of how much latency accesses to memory or to other cores is. In the event that the latency of a deterministic resource is required, only a counter is necessary; however, for non-deterministic resources such as off-chip memory (DRAM) the histogram will provide range of latencies that the application is subjected to during runtime.

4.4.6.8 PLB Slave IPIF Monitor

The PLB slave IPIF performance monitor combines counters, timers and utilization monitors together into a single performance monitor. The purpose of having this single monitor is to reduce the number of connections needed to the performance monitor hub. Furthermore, since a majority of the designs that will be evaluated as part of this work will have an interface to the PLB, generating this core once will allow for rapid adoption into the performance monitoring infrastructure later.

As part of the Systematic Design Analysis flow the static HDL profiler will identify whether or not a core has a PLB slave interface. If the core is identified as having the slave interface, the profiler uses the IPIC signals and generics to identify the number of read and write software addressable slave registers. Currently, the profiler pares for specific signal and generic names which are common as part of the Xilinx Create/Import Peripheral wizard [63]. These include the `Bus2IP_WrCE` and `Bus2IP_RdCE` signals and `C_NUM_CE` generic. From here counters can be added for each register (denoted by the chip enable (CE) with the counting enable signal being the read and write chip enable signals. In addition to the counters for the software addressable registers are acknowledgment timers which count the number of clock cycles the hardware core spends acknowledging the bus for read and write requests.

4.4.6.9 PLB Master IPIF Monitor

Similar in theory to the PLB slave IPIF performance monitor, the master monitors bus transactions that the hardware core is initiating. That is, when the hardware core needs to read or write data to an address range that is outside of itself. This core monitors the number of requests issued by the master, the time spend waiting for the transaction to complete, and the address ranges the requests are issued to. This monitor is useful in identifying if the hardware core is issuing a large number of requests to off-chip memory and if those requests have a long latency. As part of the Systematic Design Analysis flow those cores are then able to be replaced with a DMA interface to off-chip memory to improve latency and increase bandwidth.

4.4.6.10 Enhanced FIFO Status Monitor

The FIFO status core monitors the empty and full time of a specific FIFO. However, there is more information that can be collected pertaining to the functionality of the FIFO. Specifically, it was identified during one of the case studies that not all of the data is being consumed out of the FIFO. To verify, this enhanced monitor added additional counters to compare the number of reads and writes to and from

the FIFO.

4.4.6.11 Finite State Machine Profiler

The last major performance monitor added is built on top of the histogram monitor. Specifically, a monitor that identifies which state(s) a FSM is spending most of its time. The number of bins in this case represents the number of states in the state machine. A unique feature of the performance monitoring infrastructure is the rapid collection of runtime data, such as FSM state data, that could aid the designer in creating a more efficient system. The FSM profiler is modeled after the GNU Profiler [78].

CHAPTER 5: EVALUATION METHODOLOGY

The evaluation of this work is based on the ability to assemble and validate the functionality of the Systematic Design Analysis flow over a range of resources and across different applications. This is chosen over comparing against a designer's efforts to create a custom hand tuned cluster implementation for pragmatic and scientific reasons. Studies in this area would likely require a larger population of human subjects than we would be able to assemble.

The focus instead will be on the productivity of such an approach, meaning can the designer optimize for a single core or node instead of the cluster, and the validation will be in the ability to: (1) map an existing hardware application to a cluster of FPGA resources, (2) doing so while efficiently utilizing the available resources such that (3) a speedup can be measured per each application. The success of this work is not dictated by the quantitative degree of speedup (it is also possible that a result of this approach will be a slowdown for applications) or the utilization of the resources, but instead in the ability to address the questions associated with this approach towards productively scaling a design to the available resources.

Section 5.1 describes the various resources that are available as part of the Reconfigurable Computing Systems lab's Reconfigurable Computing Cluster project. Then in Section 5.2 a list of minimal functionality is presented for each stage in the Systematic Design Analysis flow that will be used in its evaluation in Chapter 6. Finally, in Section 5.3 a brief overview of how the Systematic Design Analysis flow will be evaluated with applications is presented.

5.1 Evaluation Infrastructure

The Reconfigurable Computing Cluster Project's *Spirit* cluster will be used for evaluation purposes. Recent renovations to the cluster have included the addition of Xilinx Virtex 5 FX and LX FPGAs which allow for a wider range of tests with a more diverse set of resources. The resources under test will include the Xilinx ML410, ML510, and XUPV5 development boards and any peripherals on these boards. Not every resource is required to be tested in this work; however, any application requiring or that could benefit from the addition of one or more of these resources should be supported.

Chapter 2 covers the basic resources found in an FPGA which include: lookup tables, flip-flops, configurable logic blocks, on-chip block memory, processors, discrete processing blocks, and high-speed serial transceivers. The following subsections will briefly list out the specific FPGA resources and peripherals that can be found on the development boards that will be included in this work.

5.1.1 Xilinx ML410

The Xilinx ML410 development board [22] consists of a Xilinx Virtex 4 XC4VFX60 FPGA on an ATX motherboard form factor. Table 5.1 lists the peripheral devices on the development board. The XC4VFX60 FPGA includes 25,280 slices each with two 4-input lookup tables and two flip-flops, 128 DSP slices, 232 18-Kb Block RAM, 12 DCMs, 16 high-speed serial transceivers, and two PowerPC 405 processors.

5.1.2 Xilinx ML510

The Xilinx ML510 development board [82] consists of a Xilinx Virtex 5 XC5VFX130T FPGA on an ATX motherboard form factor. Table 5.2 lists the peripheral devices on the development board. The XC5VFX130T FPGA includes 20,480 slices each with four 6-input lookup tables and four flip-flops, 320 DSP slices, 298 36-Kb Block RAM, 12 DCMs, 20 high-speed serial transceivers, and 2 PowerPC 440 processors.

Table 5.1: Xilinx ML410 development board resources

Quantity	Size	Type of Resource
1	512 MB	DDR2 DIMM (originally 256 MB)
1	64 MB	DDR component
1	512 MB	CompactFlash card
2	–	RJ-45 connectors to Trimode Ethernet PHYs
2	–	PCI Express downstream connectors
4	32-bit/33 MHz	PCI connectors
2	–	USB peripheral ports
1	–	parallel port
2	–	serial ATA connectors
2	–	UARTs with RS-232 connectors
1	–	IIC/SMBus interface
1	–	SPI EEPROM
1	–	JTAG / trace debug ports
1	–	flash memory interface

5.1.3 Xilinx XUPV5

The Xilinx University Program (XUP) V5 development board [83] consists of a Xilinx Virtex 5 XC5VLX110T FPGA on a custom PCB. Table 5.3 lists the peripheral devices on the development board. The XC5VLX110T FPGA includes 17,280 slices each with four 6-input lookup tables and four flip-flops, 64 DSP slices, 148 36-Kb Block RAM, 12 DCMs, and 16 high-speed serial transceivers. The XUPV5 is a logic series part which means there are no hard processors blocks within the FPGA fabric. Instead, soft processors such as the Xilinx MicroBlaze [84] must be added to the design when a processor is needed.

5.2 Systematic Design Analysis Flow Evaluation

The Systematic Design Analysis covered in Chapter 4 consists of a series of steps that begin with assembling a project and end with an overall performance analysis of a system running on the cluster of resources. Prior to evaluating the entire approach with any applications, it is necessary to first evaluate the functionality of each stage. This can be considered as a simple checklist of functionality and capabilities that

Table 5.2: Xilinx ML510 development board resources

Quantity	Size	Type of Resource
2	512 MB	DDR2 DIMMs
1	512 MB	CompactFlash card
2	–	RJ-45 connectors to Trimode Ethernet PHYs
2	–	PCI Express downstream connectors
4	32-bit/33 MHz	PCI connectors
2	–	USB peripheral ports and one parallel port
2	–	serial ATA connectors
2	–	UARTs with RS-232 connectors
1	–	IIC/SMBus interface
1	–	SPI EEPROM
1	–	JTAG / trace debug ports
1	–	flash memory interface

Table 5.3: Xilinx XUPV5 development board resources

Quantity	Size	Type of Resource
1	256 MB	DDR2 SODIMM
1	1 GB	CompactFlash card
2	–	RJ-45 connectors to Trimode Ethernet PHY
2	32 MB	XCF32P Platform Flash PROM
1	32-bit	ZBT synchronous SRAM and Intel P30 StrataFlash
1	–	USB peripheral port
2	–	serial ATA connectors
2	–	UARTs with RS-232 connectors
1	–	JTAG / trace debug ports

should be supported by each stage. These specific functions should be considered by the reader as the applications are also being evaluated. Chapter 6 presents the results pertaining to the assessment of thes function.

5.2.1 Project Assembly

The project assembly stage will need to perform the following steps:

1. input source HDL for the design
2. input design constraints file
3. parse design constraints file for:
   (a) FPGA board types in system
   (b) network configuration
4. create project for component synthesis stage
   (a) create top-level project for synthesis tool
   (b) create sub-projects for each sub-component
5. create project for static HDL profiling stage

5.2.2 Component Synthesis

Component synthesis is used for two purposes, to synthesize a design with a single component instance for a baseline performance comparison and to provide single component resource utilization. This stage must perform the following:

1. synthesize project created in project assembly stage
2. synthesize sub-projects
3. parse synthesis reports for all projects
   (a) generate resource utilization data structures
   (b) pass data structures to static HDL profiling stage

5.2.3 Single Node Performance Evaluation

The single node performance evaluation is used for a baseline performance comparison. The following steps are performed at this stage:

1. run design with test application
2. store results in data structure
3. pass data structure to monitor single node performance stage
4. pass data structure to performance analysis stage

5.2.4 Static HDL Profiling

The static HDL profiling stage is the first stage the can provide the developer important information regarding the potential performance scalability and bottlenecks of the design.

1. parse each input HDL file for:
   (a) port map signals
   (b) internal signals
   (c) internal components
   (d) fintie-state machines
2. evaluate signals for interfaces
3. evaluate resource utilization from synthesized project
4. pass data structures to insert performance monitors stage

5.2.5 Insertion of Performance Monitors

Based on the static HDL profiling and the amount of available resources remaining in the system assemble a list of potential monitor cores for the identified signals, interfaces and components. The monitors added to the system are then passed to the next stage.

1. parse data structures from previous stages

2. recommend performance monitors for insertion
3. insert performance monitoring infrastructure

5.2.6 Monitor Single Node Performance

The monitor single node performance evaluation is used to collect the monitor information of the single node performance. The following steps are performed at this stage:

1. run design with test application
2. store results in data structure
3. retrieve monitor cores results
4. parse monitor cores results
5. pass results to next stage
6. verify results match single node performance evaluation

5.2.7 Candidate Set Generation and Selection

The candidate set generation stage performs the following steps, note that only a finite number of configurations will be considered in order to not generate a restrictively large set that cannot be evaluated in any reasonable amount of time:

1. parse static HDL performance data structure
2. parse performance monitor results
3. determine possible memory configurations
   (a) slave on system/peripheral bus
   (b) master on system/peripheral bus
   (c) direct connect to memory controller
4. determine possible network configurations
   (a) number of connects needed to neighbors
   (b) configuration of off-chip network core(s)

   (c) configuration of on-chip cores to off-chip network

5. determine possible on-chip interconnect configurations

   (a) system/peripheral bus

   (b) crossbar switch

   (c) direct connect

6. generate synthesis projects for each candidate configuration

5.2.8 Cluster Synthesis

For each project that is identified in the candidate set, synthesize the project for the cluster of resources. This stage is similar to the single node synthesis, except it is possible that based on the configuration, that multiple synthesis will be run for a single configuration. The resulting bitstreams are passed to the next stage.

1. synthesize candidate for cluster of resources
2. pass bitstream to cluster performance evaluation stage

5.2.9 Cluster Performance Evaluation

The cluster evaluation stage will consist of:

1. parse input test vectors and results files
2. distribute configurations to each of the nodes in the test
3. execute the input test vectors
4. record the results
5. compare results to verify functional system
6. pass performance results to next stage

5.2.10 Performance Analysis

The final stage is to analyze the performance of the scaled system to that of the original system. This stage requires the following:

1. parse results from single node performance evaluation
2. parse results from cluster performance evaluation
3. compare performance results and generate results file

At this point a configuration from the candidate set has been run on the cluster. The designer can analyze the results in terms of raw speed up to determine if the desired performance was met. If not, the system will return to evaluate another configuration from the candidate set. The evaluation of all of these functions will be more thoroughly analyzed in Chapter 6.

5.3 Evaluation with Applications

While building the hardware infrastructure is a critical step in this effort, the success of the process cannot be easily measured without the use of some set of applications. Therefore, four experimental applications have been identified for testing with the Systematic Design Analysis flow. While it would be ideal to cover these applications and many more, it would require additional development of the new applications. Furthermore, this development would be required by the author which could introduce additional bias since the author is fully aware of the Systematic Design Analysis flow and could unintentionally develop the application to benefit this work. Therefore, to mitigate risk, applications that have already been designed by other colleagues will be used.

These applications in their present form have not been assembled to fully utilize the entire cluster of resources. In some cases, the applications have been scaled and run on the cluster; however, this work chooses to use the application's single node base system, specifically the hardware accelerated compute core, as the evaluated entity. For comparison sake, when an application has been hand tuned to run on the cluster of resources an evaluation between the Systematic Design Analysis and the hand tuned implementation will be presented as part of the application's case study.

Overall, these applications present the ability to determine if a designer opti-

mizing for a single node can obtain continued performances gains across the available resources without redesigning the system or requiring the designer to manually modify the design. As these applications have been designed by colleagues in the Reconfigurable Computing Systems lab their cited work is also listed to give the reader an opportunity to further understand the application. For more information regarding each implementation please refer to the respective work's publication. Next, in Chapter 6, each of the following applications will be presented in more detail in the form of four case studies.

Applications Under Evaluation:

1. Matrix-Matrix Multiplication
2. Basic Local Alignment Search Tool (BLAST)
3. Smith/Waterman Algorithm
4. Collatz Conjecture

CHAPTER 6: ANALYSIS

The Systematic Design Analysis flow will be evaluated with four applications in the form of four separate case studies. The first and second case studies are performed on mature applications that have been evaluated and functioning for several years in the Reconfigurable Computing Systems lab. The third and fourth case studies investigate applications that are younger, still under development, and have not been thoroughly evaluated in terms of performance and correctness. Upon completion of these case studies, an evaluation will be performed on the functionality and capabilities of the Systematic Design Analysis flow. This is to ultimately determine its ability to encapsulate the knowledge of an experience hardware designer.

Each of the four case studies are organized as follows. First, an overview of the application is presented, along with a simple example or description of the algorithm. The hardware design is presented in the second section, which is used as the base case when comparing candidate configuration performances. In the third section the implementation is presented. The implementation refers to the use of the Systematic Design Analysis flow and its supporting tools. Details regarding specific stages of the flow are given. The results and analysis section follows with a discussion of how specific stages in the flow performed. Finally, in the last section the observations made while performing the case study are presented.

6.1 Case Study: Matrix-Matrix Multiplication

Matrix-Matrix Multiplication (MMM) is a basic algebraic operation where two matrices, A and B, are multiplied together to form the resultant matrix C, as shown in Figure 6.1. Commonly this is implemented in a high-level program language as a triple-nested loop iterating over two dimensional arrays. Figure 6.2 shows a simple C

$$\begin{bmatrix} c_{1,1} & c_{1,2} & \cdots & c_{1,n} \\ c_{2,1} & c_{2,2} & \cdots & c_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m,1} & c_{m,2} & \cdots & c_{m,n} \end{bmatrix} = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{bmatrix} \times \begin{bmatrix} b_{1,1} & b_{1,2} & \cdots & b_{1,n} \\ b_{2,1} & b_{2,2} & \cdots & b_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m,1} & b_{m,2} & \cdots & b_{m,n} \end{bmatrix}$$

Figure 6.1: matrix-matrix multiplication

```
int mmm(int a[SIZE][SIZE], int b[SIZE][SIZE], int c[SIZE][SIZE]) {
    int i,j,k;
    for (i=0;i<SIZE;i++) {
        for (j=0;j<SIZE;j++) {
            c[i][j] = 0;
            for (k=0;k<SIZE;k++) {
                c[i][j] += a[i][k] * b[k][j];
            }
        }
    }
    ...
```

Figure 6.2: matrix-matrix multiplication in C

implementation of the multiplication and accumulation for each cell in the resultant matrix C. The operations for each cell in matrix C can be executed in parallel and are ideal for a custom hardware implementation. Section 6.1.1 discusses the specific implementation used in this case study; however, it should be pointed out that this matrix-matrix multiplication implementation is for demonstration purposes only and has not been integrated with any of the Basic Linear Algebra Subprograms (BLAS) such as Single precision General Matrix-Matrix Multiplication (SGEMM) [85]. Instead the purposes of including the design within this work is to study a highly parallelizable hardware core as the system scales beyond a single compute node's resources. Moreover, this work aims to improve existing performance of implementations of MMM that have already been tested on the Reconfigurable Computing Cluster's *Spirit* cluster.

6.1.1 Design

The focus of this case study is on a single-precision implementation of MMM, which can be decomposed into many Multiply and Accumulate (MAcc) steps. Fig-

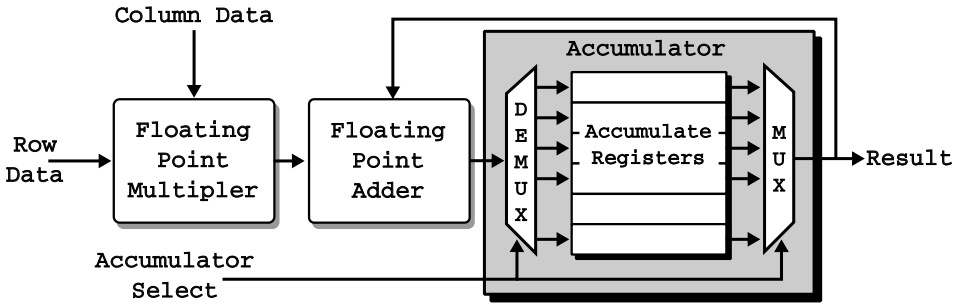


Figure 6.3: single precision multiply accumulate unit

ure 6.3 shows how the single precision multiplier and adder are combined to form one multiply accumulate unit. The two inputs for the row and column values are connected to the inputs of the multiply unit. The result of the multiplication is one of the inputs to the adder. The other input to the adder is one of the available accumulation registers. The accumulation registers are necessary to hide the latency of the adder. The number of registers is proportional to the latency of the adder. If only one accumulation register was used then the system would have to stall while waiting for the previous step's accumulation to finish. In this system the intermediate results of independent multiply-accumulate operations are being stored in the registers.

Single Precision Floating Point Unit While many FPGA based floating point units have been presented in the literature [86, 87, 88, 89, 90, 91], this design chooses to use the parameterizable floating point unit generated by the Xilinx CoreGen utility [62]. The parameters of the floating point core include precision (single, double, or custom), utilization of DSP48 primitives in the FPGA, and latency, which affects both clock frequency and resource utilization. The goal of this work is to maximize the number of multipliers and adders while maintaining at least a 100 MHz clock frequency. The single precision floating point multiply unit has been configured with a six clock cycle latency, four DSP48 slices, 146 LUTs and 175 FF. The single precision adder has been configured to have a seven clock cycle latency and use no DSP48 slices, 565 LUTs and 397 FFs. Figure 6.4 shows the GUI for the Xilinx CoreGen utility with parameters set for the Single Precision Floating Point

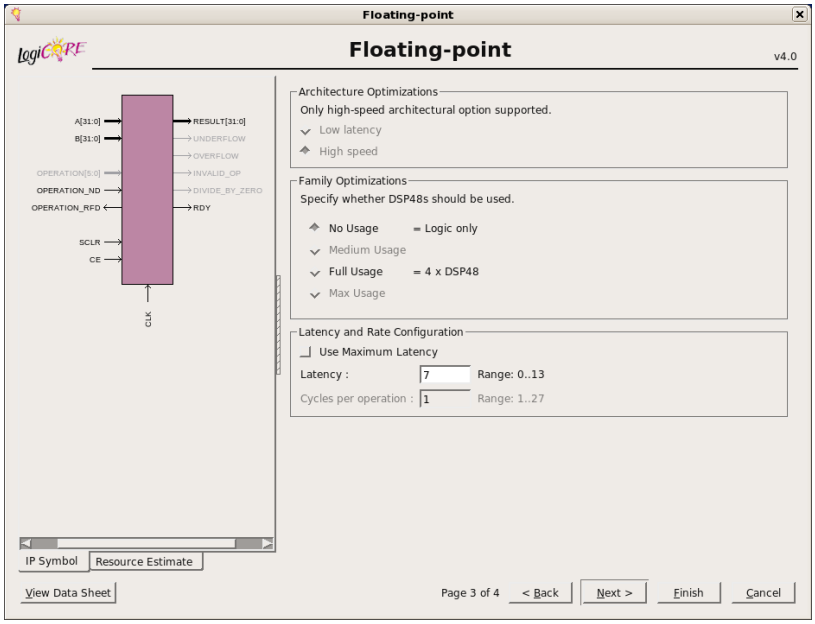


Figure 6.4: sample SP FP adder Xilinx CoreGen MMM parameters

adder. Based on the multiplier and adder resource utilization, it is possible for the design to instantiate 32 multipliers and 32 adders. This fully utilizes the DSP48 slices while using only half of the LUT and FF resources in order to maintain the 100 MHz minimum system clock requirement.

MAcc Array Figure 6.5 shows MAcc units assembled into an array to support matrix-matrix multiplication. The FIFOs around the edge of the array help to keep new data available every clock cycle. The MAcc units are connected as an variable sized array. On an FPGA the available resources limit the size of the array. For the Virtex 4 FX60 FPGA on the ML410 development board, there are 128 DSP48s. Single precision floating point multiplication and addition can consume zero to several DSPs based on the desired operating frequency and latency. Therefore, a critical design decision is the size of the MAcc array. The VHDL source to generate the MAcc array is listed in Figure 6.6. The use of `generics` and `generate` statements enables the design to scale with the available resources.

Fully Utilized MAcc Array The original MMM hardware core design approach tries to maximize the size of the MAcc array with respect to the available

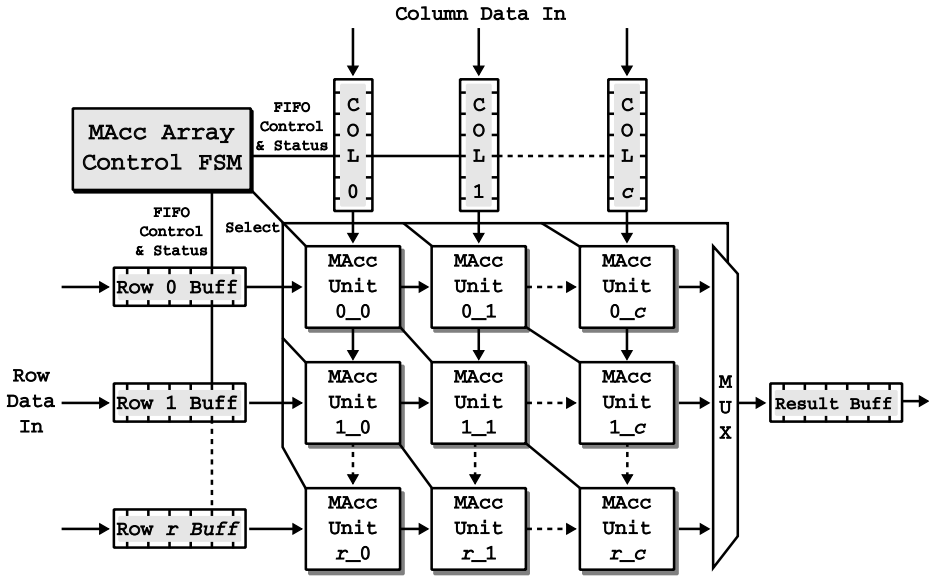


Figure 6.5: a variable sized array of MAcc units

```
-------------------------------------------------------------------
-- Multiply Accumulate Units Generate Statement
-------------------------------------------------------------------
MACC_ARRAY_COL_GEN: for i in 0 to (NUM_COLS-1) generate
  MACC_ARRAY_ROW_GEN: for j in 0 to (NUM_ROWS-1) generate
    macc_i_j : entity work.macc
      generic map (
        NUM_REG    => NUM_REG,
        S_WIDTH    => S_WIDTH,
        D_WIDTH    => D_WIDTH,
        MULT_DELAY => MULT_DELAY,
        ADDR_DELAY => ADDR_DELAY
        )
      port map(
        clk         => clk,
        rst         => rst,
        col_valid   => col_valid(i),
        col_data    => col_dout_array(i),
        row_data    => row_dout_array(j),
        macc_we     => macc_we_array(i)(j),
        macc_in     => macc_in,
        macc_select => col_select(1 to S_WIDTH),
        macc_out    => macc_dout_array(i)(j));
  end generate MACC_ARRAY_ROW_GEN;
end generate MACC_ARRAY_COL_GEN;
```

Figure 6.6: VHDL code snippet of MAcc Array

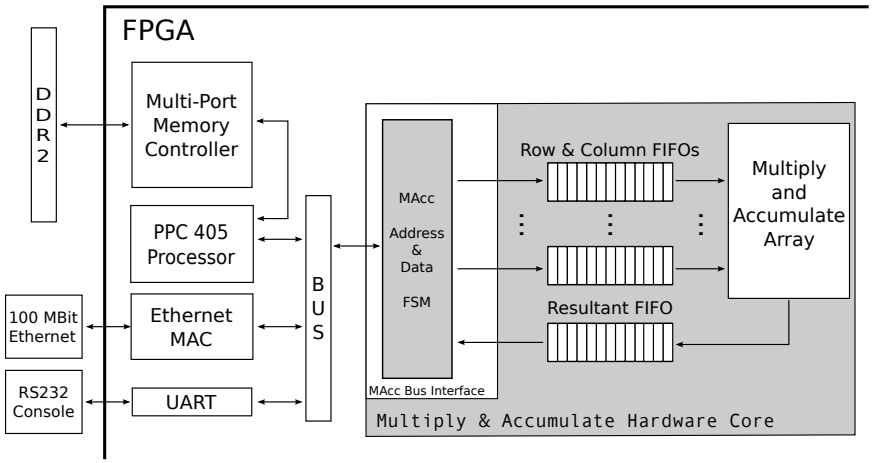


Figure 6.7: block diagram of MMM core's programmable I/O system

resources, in order to achieve the highest performance. On the Virtex 4 FX60 FPGA this results in a MAcc array of size 16×2. However, as a result of the accumulate registers in the MAcc unit, the array actually directly supports multiplication of matrices whose result is 16×16. For example, if `register 0` in any particular MAcc unit held the value for the element (x, y) in the result matrix, then the other seven accumulation registers would hold the values for the elements $(x, y+1), (x, y+2), (x, y+3), ..., (x, y+7)$. This effect is achieved by the ordering of the data in the row and column FIFOs and a controller to handle fetching new data elements from the FIFOs.

In the initial implementation the processor writes matrix A and B data into the input FIFOs, then waits for the MAcc array to complete the calculation before retrieving the resultant matrix C. This process of retrieving data for the hardware core is known as programmable I/O. Figure 6.7 depicts this base system and the top-level entity of the MMM hardware core. The Systematic Design Analysis flow will use this fully-utilized MAcc Array hardware core to determine the best candidate for scalability across the cluster of available resources. In the next section the evaluation MMM core is performed by the Systematic Design Analysis flow.

6.1.2 Implementation

6.1.2.1 Project Assembly

To start the Systematic Design Analysis flow the Project Assembly stage begins with an existing project from XPS and generates the subsystems to be used throughout the remainder of the flow using the `Generate Systems` tool. This tool is described in full detail in Chapter 4. The primary functions are to run PlatGen and to construct the `RCS_TOOLS` subdirectory which contains the necessary HDL and synthesis scripts to be used in the Static HDL Profiling and Component Synthesis stages. Presently, the system is not configured to run beyond a single node, therefore this section's evaluation is on the single node performance.

6.1.2.2 Static HDL Profiling

The second stage is to parse the system and identify the components and subcomponents of the system. This is accomplished through the `System Parser` tool, the details of which are described in Chapter 4. The MMM core is identified to contain the following subcomponents: `plb_slave_ipif`, `user_logic`, `mac_array`, 32 `mac_units`, and eighteen 32-bit×512 deep FIFOs. From these components it is identified that the MMM core connects as a slave to the PLB and there are 25 software addressable registers.

6.1.2.3 Component Synthesis

Table 6.1 lists the resource utilization for the Matrix-Matrix Multiplication core's top-level entity and key subcomponents. At the heart of the MMM core is the MAcc array which consists of 32 MAcc units. Individually, each MAcc unit only occupies a small portion of the FPGA (≈1% of the flip-flops and lookup tables and 3.13% of the DSPs); however, as the system scales to create the full MAcc array, the utilization increases to ≈57% of the flip-flops, ≈68% of the lookup tables and ≈100.00% of the DSP48s. The limiting factor at this point is not the slices or LUTs, but instead in the number of discrete processing slices. This initial design has been hand optimized to

Table 6.1: resource utilization of MMM hardware core (V4FX60)

Component	Slice FFs (%)	4-LUTs (%)	DSP48s (%)	BRAMs (%)
MAcc Unit	831 (1.64%)	984 (1.95%)	4 (3.13%)	0 (0.00%)
MAcc Array	28922 (57.20%)	34394 (68.02%)	128 (100.00%)	18 (7.76%)
MMM Core	30354 (60.03%)	35979 (71.16%)	128 (100.00%)	18 (7.76%)

fully utilize this exact number of DSPs. Also identified are the sizes of the FIFOs used as buffers to store the row and column data for the MAcc array as data is transferred to the MMM core from the processor. Each FIFO has been initially configured to 32-bit×512 elements.

6.1.2.4 Performance Monitor Insertion and Evaluation

Three sets of performance monitors have been inserted to evaluate the MMM core. First is the PLB slave IPIF where the processor is writing data directly to the MAcc array's input FIFOs. The monitor will determine the efficiency of the transfers which will be used to determine if an alternative candidate configuration exists to improve the data transfer between the processor, memory and the MMM core. The second set of performance monitors are for the eighteen input FIFOs. These monitors will analyze the capacity of the FIFOs to determine if more buffer resources should be allocated to improve future performance. Finally, a utilization performance monitor is added to determine how much time the actual core spends performing the computation as opposed to I/O time.

Currently, these performance monitors must be manually added to the design. These monitors are chosen from an assortment of available monitors that all have a common interface to a performance monitor hub. The purpose of the performance monitor hub is to aggregate all of the performance monitor core's data and to send the data to a centralized head node that is connected to the node under test through the performance monitoring system. The `Performance Monitor Insertion` tool automatically generates the necessary infrastructure in order to add the performance monitoring system to the node under test. This tool is described in more details in

Chapter 4.

The results from the single node performance monitoring indicate that the largest bottleneck in the current design is the PLB slave IPIF. The processor spends over 98% of the total execution time transferring the matrix data into or the result data out of the MMM hardware core. Section 6.1.3 shows that the performance of this initial implementation is so poor that only three small matrix sizes could even be performed, all with unfavorable results. In addition to the bus monitor and utilization results the FIFO utilization showed that while the FIFO was not fully utilized, it did set an upper bound on the overall size of the matrix to be computed. However, increasing the FIFO size on this initial implementation is not estimated to increase the overall performance, instead alternative configurations are necessary.

6.1.2.5 Candidate Set Generation, Selection, and Evaluation

Clearly, the processor writing the matrix data to the hardware core is not an ideal configuration. However, in terms of rapid development, it is better for a designer to focus on designing the actual compute core (kernel) rather than trying to improve the performance of the I/O. The Systematic Design Analysis flow is ideally suited for cases such as the MMM core since it is possible to quickly generate a set of candidate configurations that have alternative I/O interfaces. This case study will focus these alternative configurations as well as the scalability of the system on heterogeneous resources. Moreover, with the DSP limitation this case study will look at scalability across a cluster of resources which improves not only the number of available DSP slices, but the number of channels to off-chip memory as well.

Specifically, three sets of alternative candidate configurations are being evaluated with twelve total configurations implemented on the cluster of available resources. The first is the DMA implementation. The second includes the DMA interface with a fast Ethernet connection for node-to-node communication. The third is a DMA interface with a crossbar switch to connect a custom high-speed network that is part

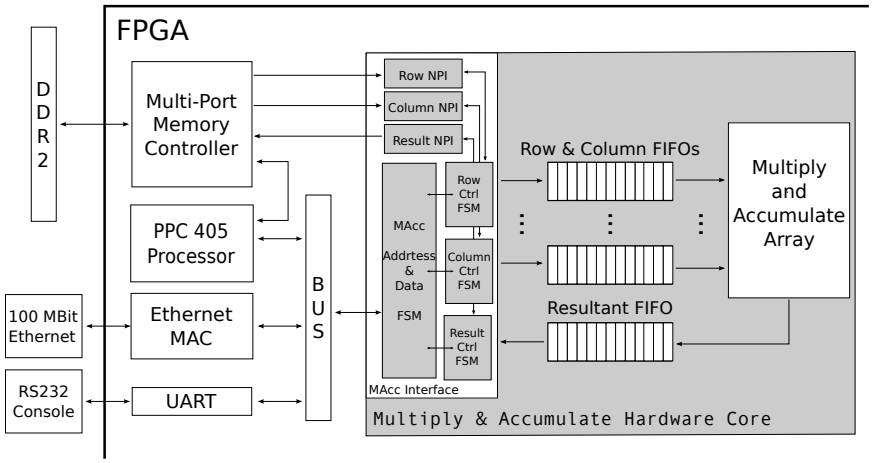


Figure 6.8: candidate configuration connecting MAcc Array to the PLB and fast Ethernet for scalability up to 49 nodes

of the *Spirit* cluster. Figure 6.8 and Figure 6.9 illustrate these two designs respectively.

In place of the programmable I/O, a custom memory interface has been developed to connect to the MMM core. This custom core acts as a wrapper around the MAcc array, allowing the array to remain unmodified by the designer. Instead, the memory interface can perform DMA transfers to try and fully utilize the memory channel's bandwidth. Since both cluster configuration candidates will utilize this interface the analysis between the programmable I/O and the DMA implementation will be done only once.

The second candidate configuration set extends the first candidate with DMA to use fast Ethernet to evaluate the performance as the amount of resources scales from a single node to: 4, 9, 16, 25, 36, and 49 nodes. Each of these configurations explores how well the system will scale as the contention for the network resource increases.

The third candidate configuration set utilizes a crossbar switch and a custom network, known as AIREN. AIREN will be used in place of fast Ethernet when evaluating the scalability of the MMM system across an variable number of FPGA nodes: 1, 4, 9, 16. AIREN is further supported by the crossbar switch which is primarily responsible for connecting the AIREN's eight network channels together

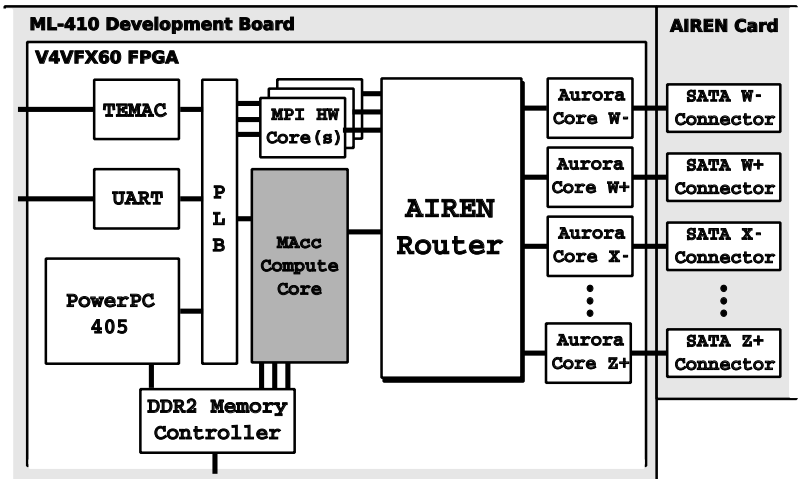


Figure 6.9: candidate configuration connecting the MAcc Array to crossbar switch and AIREN network for scalability up to 16 nodes

along with DMA channels to off-chip memory on each node. Therefore, the processor can issue MPI-like send and receive commands to transfer blocks of data between nodes.

6.1.3 Results and Analysis

The results and analysis are split into two sections. First is the single node performance which includes the original (unmodified) performance along with the DMA candidate configurations. The second section details both the fast Ethernet and AIREN candidate configurations. In total, twelve configurations are evaluated with a peak speedup of 40,438× achieved over the original hardware implementation.

6.1.3.1 Single Node Performance

The original MMM core consists of a 16×2 MAcc array which offers a theoretical peak performance of the MAcc unit is 6400 MFLOPS due to the possibility of all 32 multipliers and 32 adders to generating new results every 10 nanoseconds (100 MHz clock frequency). Real application behavior introduces pipeline latencies, off-chip memory limits the rate of computation, and other activities (operating system, instruction fetches) reduces memory bandwidth available to the MAcc unit. Furthermore, since the original configuration consists of the processor using programmable

Table 6.2: performance of original MMM core with programmable I/O

Matrix Size	Performance (MFLOPS)	Speedup
16	0.095	1.00
32	0.190	2.00
64	0.380	3.98

I/O, the performance is anticipated to be underwhelming. In fact, as Table 6.2 shows, the peak performance is calculated at a mere 0.380 MFLOPS. Furthermore, due to the FIFO size limitations the maximum matrix size that can be evaluated is 64×64. The Systematic Design Analysis flow could be used to increase the size of these FIFOs; however, considering the already low performance and the only modest speedups gained thus far, a more aggressive candidate selection is chosen.

In fact, augmenting the MAcc array's interface from programmable I/O to DMA will also allow the MMM core to perform larger matrix computations without necessarily increasing the FIFO size since the DMA interface is responsible for retrieving the sub-blocks of the matrices being computed. Also, it is anticipated that the MAcc unit will perform better for large matrix sizes because the custom memory controller can utilize longer burst sizes to fetch the data elements and fill the MAcc unit's FIFOs. The amount of data passed from the processor to the memory controller to initialize a multiplication is also constant regardless of matrix size: the processor only passes base address pointers and sizes of the matrices to the memory controller.

In terms of performance, Table 6.3 shows the measured MFLOPS for the 16×2 Macc array with DMA. The limitation on the size of the matrix is because multiplying two 4096×4096 single precision matrices requires 192 MB of memory; 64 MB for the two input matrices and the result matrix. Three results are clearly seen. First, DMA provides tremendous performance gains over programmable I/O, 3811.77 MFLOPS vs. 0.380 MFLOPS, or a $\approx$10,000$\times$ speedup. Second, the performance continues to increase as the size of the matrix increases. Third, while the speedup is impressive, the percentage of the theoretical peak (6400 MFLOPS) is at best $\approx$60%.

Table 6.3: performance of bus based MMM core with DMA

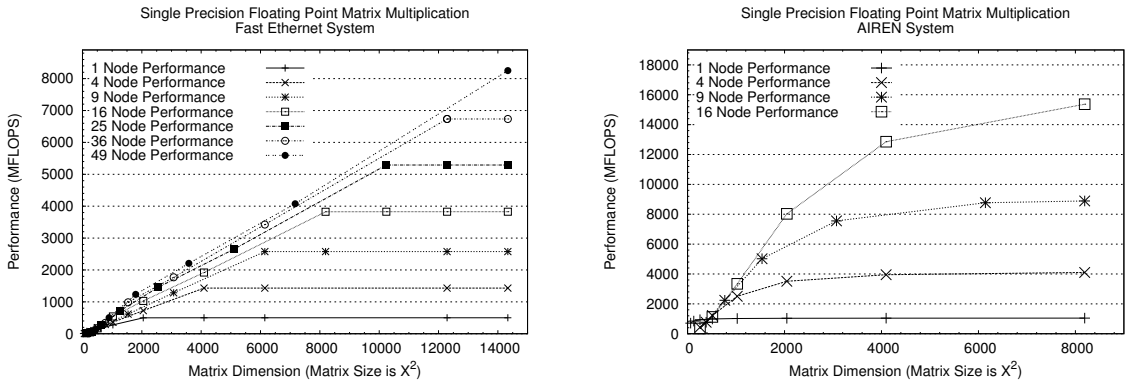
Matrix Size	MFLOPS	% of Peak	Speedup
16	839.04	13.11%	1.00
32	1431.02	22.36%	1.71
64	2102.38	32.85%	2.51
128	2726.64	42.60%	3.25
256	3197.60	49.96%	3.81
512	3498.54	54.66%	4.17
1024	3670.87	57.36%	4.38
2048	3763.60	58.81%	4.49
4096	3811.77	59.56%	4.54

6.1.3.2 Multi-Node Matrix Multiplication

Implementing the design across a cluster of FPGAs requires a control mechanism to pass submatrices between the nodes. For this the designer has supplied the Systematic Design Analysis an additional application which implements Cannon's algorithm [92]. Cannon's algorithm segments the large matrices into smaller blocks and passes these blocks around to each node to compute in parallel. In effect, these blocks could be conceptualized as a single number in a much smaller matrix. In order to support the communication, the third candidate configuration adds fast Ethernet. MPI is used as part of the application running on each node to pass the data between nodes.

For evaluation of the fast Ethernet candidate configurations, up to 49 nodes have been used as part of the *Spirit* cluster. Matrix sizes were evaluated ranging from 32×32 to $14,336 \times 14,336$. In total seven configurations were evaluated where the number of MMM cores evaluated are: 1, 4, 9, 16, 25, 36, and 49. Figure 6.10(a) illustrates the performance results from these candidate configurations in terms of MFLOPS. The graph shows the peak perform of 8252.24 MFLOPS occurs with 49 MMM cores running on 49 nodes with a matrix size of $14,336 \times 14,336$. This is the largest matrix that can be evaluated with the Spirit cluster due to limitations on the capacity of off-chip memory.

While the ≈8200 MFLOPS significantly outperforms the original implementation (21,700× speedup), this is the performance compared to a single MMM core. When



(a) **(b)**

Figure 6.10: performance of (a) 16×2 MMM core on fast Ethernet and (b) 8×2 MMM core on AIREN network

comparing to the single node DMA implementation the speedup is 2.17× less. To further evaluate this discrepancy, performance monitor cores were inserted to monitor the initial distribution time of the matrices, the final collection time of the matrices, the time spend performing matrix multiplication, the time spend sending/receiving the matrices between MMM cores, and the total runtime time. The Systematic Design Analysis flow enables the designer to quickly insert these monitors while only requiring the designer to determine the VHDL logic needed to start and stop each monitor's timer.

Table 6.4 presents these results for the 49 MMM core implementation running the largest matrix size test. The columns "Init Send" and "Final Recv" represent the times spent distributing and collecting the matrices to and from the compute nodes by the head node. The column "Send/Recv" represents the amount of time a node spent communicating with its neighbors for Cannon's algorithm. All times are reported in units of seconds. The MAcc unit is only busy with computation for at most 10% of the total execution time for the largest problem size. The bottleneck in this system is clearly identified as communication time over Fast Ethernet.

The Fast Ethernet candidate configuration leads to the last set of candidate configurations. Namely, the use of the crossbar switch and AIREN on the cluster. Even

Table 6.4: performance using Cannon's algorithm

Matrix Size	Init Send	Final Recv	Compute Time	Send/ Recv	Total Time	Total MFLOPS
224	0.08 s	0.02 s	0.01 s	0.84 s	0.95 s	23.57
448	0.34 s	0.05 s	0.03 s	0.93 s	1.36 s	132.16
896	0.47 s	0.12 s	0.12 s	2.12 s	2.84 s	505.91
1792	4.97 s	0.29 s	0.52 s	4.6 s	10.38 s	1108.28
3584	25.78 s	1.11 s	2.31 s	13.68 s	42.92 s	2145.37
7168	117.77 s	3.73 s	11.21 s	43.02 s	175.77 s	4190.55
14336	488.53 s	16.65 s	60.66 s	149.15 s	715.03 s	8241.15

with a highly parallel implementation of the MAcc Array, the "slow" fast Ethernet network stifled the performance. This candidate configuration shifts its focus away from memory and instead turns the attention on combining a high speed network with a high performance compute core. Moreover, the designs discussed thus far have all used a 16×2 (nearly fully utilized FPGA resources) MAcc Array, but used Fast Ethernet instead of the AIREN network. The single node performance yields ≈3800 MFLOPS; however, as the system was implemented on the Spirit cluster the Ethernet network quickly limited the performance as the system scaled. A significant amount of time was spent by each node waiting for the transfers of data between nodes to complete.

In the AIREN implementation, all network transfers are performed across the AIREN's high speed network. Transfers are also performed in parallel (for both matrix A and matrix b). This is due to the fact that each node has two DMA cores and can be performing an `MPI_Send` and `MPI_Recv` on each matrix in parallel. Furthermore, the hardware implementation of `MPI_Barrier` is used to speedup synchronization between nodes. However, to include the AIREN network required reducing the size of the MAcc array, from 16×2 to 8×2. This is due to the size of the crossbar switch and the inclusion of eight network channels for AIREN.

As a result the theoretical peak performance dropped from 6400 to 3200 MFLOPS. Ultimately, the sacrifice in performance may be justified based on the insufficient

performance provided by fast Ethernet. The Systematic Design Analysis was not able to automatically adjust the size of the MMM core. This modification came from the designer, since the amount of resources needed for the 16×2 implementation was greater than those available on the Xilinx ML410 development board. Fortunately, the MAcc array was designed with generics and generate statements (as previously shown in Figure 6.6) and such a modification only required the designer to adjust the number of rows to eight, the multiplier delay to three and the adder delay to four. This highlights the importance of good hardware design practices. Comparable tests were then run for the 8×2 MAcc array and the performances are presented in Figure 6.10(b).

Overall, the results show significant performance gains even over a much larger 49 node implementation with twice the compute resources (16×2 vs. 8×2). At its peak, the implementation reached over 15,000 MFLOPS, which equates to $\approx$2$\times$ speedup over the 49 node fast Ethernet implementation and a staggering 40,438$\times$ speedup over the original hardware core. Certainly it makes sense that when transferring large datasets between nodes, the higher bandwidth AIREN implementation will outperform Fast Ethernet. However, for a designer to realize the need to sacrifice single node performance to include the necessary on-chip infrastructure for the network might not be so easily identifiable, which is why the Systematic Design Analysis flow is so ideally suited.

6.1.4 Observations and Summary

Overall, the Matrix-Matrix Multiplication case study provides a wealth of interesting observations regarding the Systematic Design Analysis flow. First and foremost, all of the results presented here indicate the SDAflow does generate several candidate configurations which significantly outperform the original hardware core, with a maximum speedup achieved of 40,438$\times$. In addition to the performance gains, Table 6.5 highlights several additional observations of this case study.

Table 6.5: summary of Matrix-Matrix Multiplication case study

Stage	Details
Static HDL Profiling	correctly identified PLB slave interface found all 25 software addressable registers found all 18 FIFOs
Component Synthesis	identified resource utilization of all components scalability limited by DSP48 resource designer could re-evaluate design for DSP48 utilization
Performance Monitors	19 monitors inserted identify that FIFOs never reached full status determine that 98% of execution time waiting for data added to fast Ethernet configuration for network evaluation
Candidate Configurations	twelve configurations evaluated converted to DMA implementation converted to DMA with fast Ethernet implementation converted to DMA with AIREN implementation scaled MAcc array up to 49 parallel cores recommended MAcc array decrease for AIREN scalability
Performance Evaluation	verified original design performance of 0.380 MFLOPS DMA implementation improved original by ≈10,000× DMA with fast Ethernet improved original by ≈22,160× DMA with AIREN improved original by ≈40,000×
Other Observations	best performance comes from design with half compute resources use of existing MPI hardware cores without designer intervention avoided unnecessary candidate configurations insert performance monitors into candidate configuration

6.2 Case Study: Basic Local Alignment Search Tool

The Basic Local Alignment Search Tool (BLAST) application was originally developed at National Center for Biotechnology Information (NCBI) in 1990 [93]. The code is open source and there have been several well-known forks of the project although NCBI BLAST remains the *de facto* standard reference. The application, which is actually a collection of programs, is used to compare an unknown genomic sequence, called a *query* against an existing *subject* genomic database to identify high similarity regions between them. The BLAST software has been developed in five flavors based on the nature of the subject database and query. These are: BLASTn, BLASTp, BLASTx, TBLASTn, TBLASTx. BLASTn is the program used to make local searches between a nucleotide-based query and subject database. Nucleotides comprise of four different letters (*monomers*): A *(Adenine)*, C *(Cytosine)*, G *(Guanine)* and T *(Thymine)*.

The algorithm is divided into four major stages. The first stage involves creation of the query lookup table and overflow table. The query is run through once forming words (*w-mers*) of a *WordLength* size. The offsets of these words are stored in the query lookup table. Each word is converted into an offset into the query lookup table where its location in the query is stored. If a certain word is repeated in the query, the multiple locations are stored in an overflow table. The offset value of that word in the lookup table is updated to a pointer to the overflow table. The size of the lookup table and the overflow table is $[2^{2(WordLength)} \times 32]$ bits.

The second stage (scan), identifies exact matching words (or "hits") in the database and the query sequence. Profiling of the sequential code with a sample queries and databases suggest that the scan stage consumes ≈78% of the total execution time. These hits are forwarded to the third stage (ungapped extension). Each hit is extended to the left and right by the stride length number of letters (stride is another user-supplied parameter to the application). Scores are incremented by one for every

exact match found. All hits that score greater than a threshold score set by the user are sent to the final stage, gapped extension. Depending upon the user's selection, gapped extension is done using either the Needleman-Wunsch algorithm[94] or the Smith-Waterman algorithm[95]. The ungapped extension typically consumes another 12% to 16% of the total execution time.

These two stages, scan and ungapped extension, have been implemented in a custom compute hardware accelerator core. Since these stages have been ported to hardware, modifications were made to the query lookup and overflow tables such that the data alignment is better suited for an FPGA based implementation. The adjacent entry to every offset value stored in the query lookup and overflow tables, store a byte of left and right data (i.e. four letters to the right and four letters to the left) to the word. These augmentations made to the lookup and overflow tables provide data to the hardware core to dynamically calculate the ungapped extension score for every hit located in the subject database. These additions lead to doubling the size of the tables ($2 \times [2^{2(WordLength)} \times 32]$ bits).

Of the applications evaluated during this work, BLAST is the most mature application to have been ported to the Reconfigurable Computing Lab's *Spirit* cluster as part of the Reconfigurable Computing Cluster project. In fact, the specific BLAST hardware core has been peer-reviewed and published twice [96, 97] and a simplified implementation has been used as a demonstration kernel in [98] as well as [29]. Furthermore, a hand-tuned implementation exists which has successfully scaled the BLAST hardware core to the Spirit cluster. This implementation will also be used when comparing the Systematic Design Analysis flow's candidate configurations running on the Spirit cluster.

The main idea that emerged as part of the BLAST Spirit cluster implementation is that technology trends are making computation cheap and that managing bandwidth — bandwidth to main memory and bandwidth to secondary storage — is critical to

streaming applications. The design pattern of using a pre-computed look-up table to process large streams of data appears in a number of domains from bioinformatics to remote sensing Earth orbiting satellite applications. In the case of BLAST, a two-dimensional array of accelerator cores was conceived where every row worked collectively on processing a single query to reduce latency and multiple columns allowed for multiple queries (to increase throughput). The maximum width of the array is limited by the speed at which the database can be streamed into the array. The height is limited by the bandwidth (after caching) to each column's look-up table in main memory. As [97] shows, the height and width can be algorithmically determined based on the specific FPGA platform's characteristics and the user's preference for higher throughput or reduced latency.

6.2.1 Design

6.2.1.1 BLAST Hardware Core Design

The BLAST core is a hardware implementation of the `blast_nascan()` and `nt_word_finder()` functions in the NCBI software. This core comprises of five primary components: (1) a sequence data FIFO, (2) a scan FSM, (3) an ungapped extension and bus master FSM, (4) two Hit Index Tables (custom designed caches) and (5) four lookup queues. The general setup of the core is shown in Figure 6.11.

Each sequence of the database is streamed in through the hardware filesystem into the sequence data FIFO. The first bit of each offset element in the query lookup table is grouped into 32 bit words and written by the software running on the PowerPC-405 to the Hit Index Tables via the PLB. This table facilitates the hit check process locally as opposed to multiple reads from off-chip memory for every word in the subject database, which is expensive in terms of latency. A detailed study of the advantages of the Hit Index Table is listed in [96]. The scan FSM's operations are (1) pop four bytes of data from the subject FIFO (2) increment the subject offset counter (3) calculate addresses in the Hit Index Tables and (4) make hit checks every

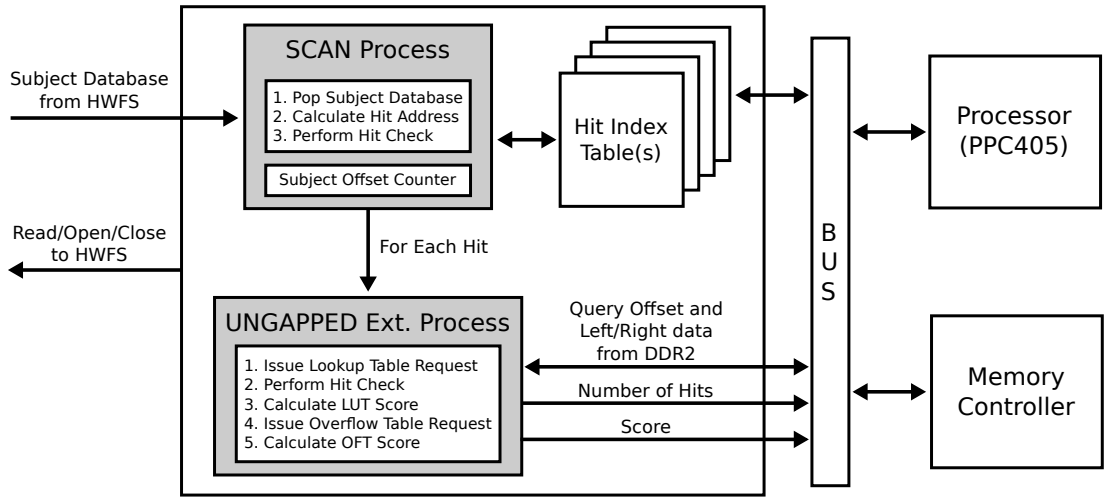


Figure 6.11: functional overview of the BLAST core

clock cycle. Four words are compressed into four bytes of subject data which have to be checked for hits in the query. The pipelined architecture of the scan FSM ensures all these steps are completed in one clock cycle.

In order to execute ungapped extension on every hit located during the scan process, a byte of data on the left and right of the identified word in the database stream and the word itself is stored into a lookup queue. Since there are four hits checked every clock cycle, four lookup queues are added to the core.

The ungapped extension and bus master FSM arbitrates across the lookup queues in a round robin technique. If a lookup queue is not empty, the state machine pops its data element and issues a read request across the bus to fetch the query offset and the left and right bytes of data from the query lookup table stored in main memory. If the data fetched is a pointer to the overflow table, sixteen elements (eight query offsets and eight corresponding left and right Datum) are fetched. Calculation of the ungapped extension score is measured to execute in one clock cycle.

As discussed in [96, 97], the database data is streamed into the BLAST core at 400 MB/s. Since four hits are checked every clock cycle, the consumption rate of the core matches the rate at which the data is written. These factors yielded a system

with a single stream of data to multiple cores, each loaded with different queries' information in their Hit Index Tables. This design completely focuses on increasing the throughput of the system.

6.2.1.2 Tree Topology

The BLAST implementation has already been ported to the Spirit cluster. This specific hand-tuned implementation exploits some of the early work of the Systematic Design Analysis flow, but relies heavily on the designer implementing custom glue-logic to connect the cores and the nodes of the cluster together. This implementation uses 25 nodes of the Spirit cluster, a 21 compute node plus four disk nodes (for Hardware Filesystem) was assembled, as is shown in Figure 6.12, which is connected in a tree topology with the head node connecting to four disk nodes and four intermediate nodes. Each intermediate node is then connected to four additional leaf nodes. The tree structure is the most appealing interconnect for the streaming application since data flows in one direction.

In order to evaluate the scalability of this system four types of nodes were created by hand: a head node, a disk node, intermediate BLAST nodes (parents in the tree) and leave BLAST nodes.

HEAD NODE The head node, shown in Figure 6.13, consists of common System-on-Chip (SoC) components (processor, memory controller, system bus, etc) for a fully running Linux 2.6 kernel on the 300 MHz PowerPC 405 processor as well as BLAST compute cores, the Hardware Filesystem (HWFS) core [24], Redundant Array of Independent Disks (RAID) controller core, and a broadcast hardware core. The head node's BLAST cores directly access the HWFS to retrieve the database needed for the computation. The HWFS delivers the database to the broadcast core which then transfers the database to both the head node's local BLAST cores and to the other BLAST nodes via the AIREN network.

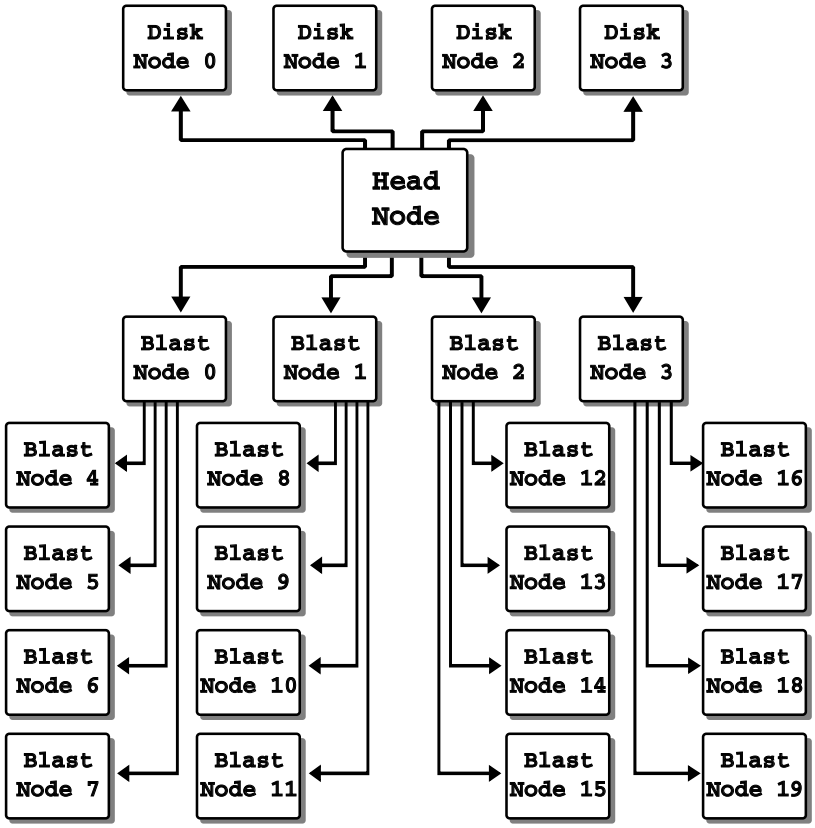


Figure 6.12: dataflow of tree topology

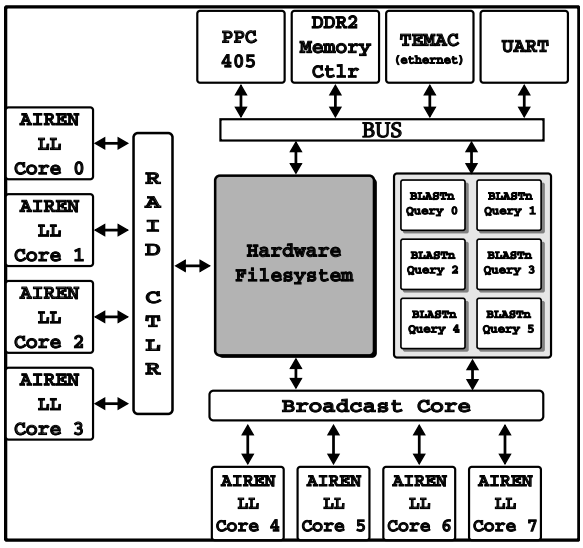


Figure 6.13: tree topology's head node

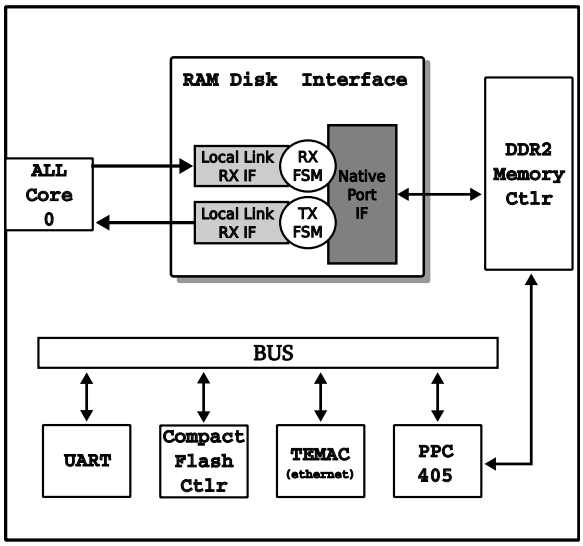


Figure 6.14: tree topology's disk node

Disk Node Figure 6.14 illustrates the disk node which contains a single bidirectional AIREN Data Link Layer (ALL) network interface to the head node's RAID controller, and an interface to a RAM Disk (512 MB per RAM Disk). There is no additional BLAST logic on the disk node due to resource limitations. The filesystem is loaded on each node's RAM Disk from CompactFlast when the disk node is powered on and resides there until the system is shutdown.

Intermediate BLAST Node There are two types of BLAST nodes in this topology. First is the intermediate BLAST node, which can be seen in Figure 6.15, and can be thought of as a parent node in the tree topology. The node includes a fully running Linux SoC along with one ALL core to receive the database from the head node, a broadcast core, four ALL cores to connect to leaf BLAST nodes (or additional intermediate nodes). The intermediate BLAST nodes can support up to eight BLAST hardware cores each.

Leaf BLAST Node The last type of node is the leaf BLAST node, Figure 6.16. The leaf node is based on the intermediate node with exception of the broadcast core and four ALL cores which were removed to free up resources for an additional four BLAST cores, for a total of twelve BLAST cores per node. The decision to include

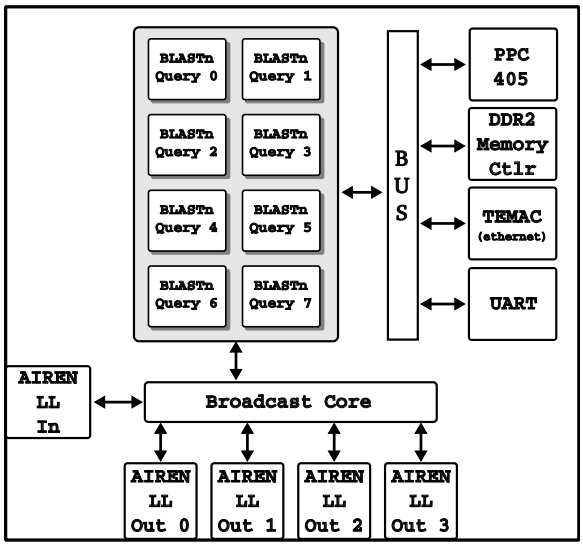


Figure 6.15: tree topology's BLAST intermediate node

a fourth node type was motivated by the goal to construct a system with as many BLAST hardware cores as possible in the resources allocated, with minimal changes to the BLAST core.

Linux is run on all of the nodes to provide MPI [99] support. MPI is used to distribute the BLAST application to each node with BLAST cores. Each node then loads the query (or queries) into its own BLAST core(s). The results presented in this paper include the time to load the queries, that is they are not considered to be preloaded. Since each BLAST node contains a Linux system, the queries can be loaded in parallel unlike a conventional co-processor accelerator model where each query would need to be loaded sequentially. The BLAST application also retrieves the results from each BLAST core which is the number of hits and number of alignments found to exceed the threshold score provided by the user.

With the tree topology, two configurations have been tested. Namely, five FPGA nodes — one head node with four BLAST leaf nodes, and 21 FPGAS — one head node, four BLAST intermediate nodes and 16 BLAST leaf nodes (in addition to the four disk nodes). In the five FPGA configuration 54 parallel BLAST hardware cores are performing queries in parallel. In the 21 FPGA configuration 230 parallel BLAST

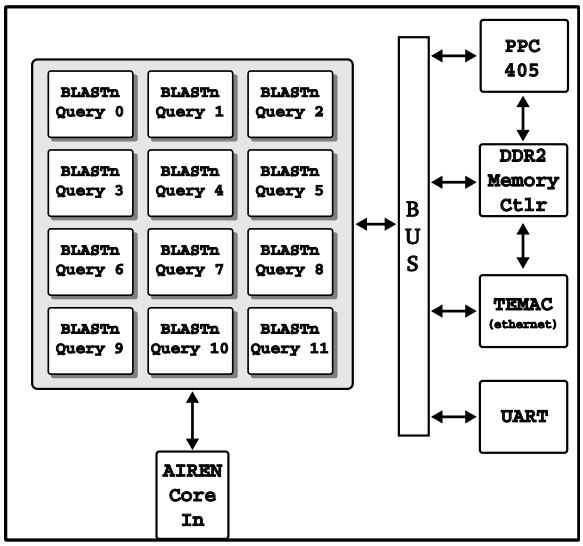


Figure 6.16: tree topology's BLAST leaf node

hardware cores are running in parallel, Figure 6.12. The results of the tree topology can be found in Section 6.2.3.2.

For the tree implementation three hardware designers spent approximately one month hand-tuning the design, creating the glue logic to connect the BLAST core, hardware file system, and the network. Modifications were made to all of the cores such that the design could be implemented on the Spirit cluster. While this undertaking appears to be significant, the Systematic Design Analysis flow was used in places to help reduce the implementation time. Specifically, performance monitors were inserted into the designs to better understand the performance and behavior of the system. In the most extreme case it was identified that a software implementation of the MPI_Barrier synchronization increased execution time by over an order of 100×. In another case the BLAST hardware core resource utilization of FIFOs was identified as being severely high, reducing the overall scalability of the hardware core on a single node. Finally, performance monitors also helped identify software coding problems that when corrected significantly improved the runtime of the BLAST application.

Of course, the tree implementation provided a framework for the Systematic De-

sign Analysis flow to be evaluated against. Most noticeable is the unnecessary inclusion of intermediate BLAST nodes and leaf BLAST nodes. A more generic design is ideal which would not limit the overall scalability of the system. While not reported, the hand-tuned design also relied heavily on the use of the AIREN network's positive and negative network channels, requiring twice the number of base designs to be created and maintained. Overall, the tree implementation consists of the following base systems: one head node, two disk nodes (positive and negative), two intermediate BLAST nodes (positive and negative), and two lead BLAST nodes (positive and negative). In total that is seven separate base systems that must be maintained by the designer. In contrast, the Systematic Design Analysis flow to be presented next was able to reduce this down to: one head node, one disk node, and one blast node. This dramatically reduces the build times for the designs and improves the designer's productivity by narrowing the focus of BLAST onto a single base system.

6.2.2 Implementation

The evaluation of the BLAST hardware core with the Systematic Design Analysis provides valuable feedback about not only the ability to quickly create different configurations and test them with very little intervention from the designer, but also to provide feedback about the current system. Furthermore, BLAST has been evaluated in both a single node capacity as well as on multiple nodes. Therefore it is possible to track the scalability of the application across the increasing number of resources.

Two projects already exist for the BLAST system. First is a single BLAST core system running on a single BLAST node. Second is the multiple node system configured to run on a fixed tree network topology. First the Systematic Design Analysis will be used to evaluate the single node implementation. Specifically, performance monitors will be added and alternative configurations will be explored. Then the tree topology system will be evaluated to determine the feasibility of the Systematic Design Analysis to generate a more generic torus topology implementation.

6.2.2.1 Project Assembly

The Project Assembly stage of the Systematic Design Analysis flow on the single BLAST core system is used to create the necessary infrastructure for the remainder of the flow to be run. In this case the `Generate Systems` tool is used to create the `RCS_TOOLS` subdirectory which contains the necessary HDL and synthesis scripts to be used in the Static HDL Profiling and Component Synthesis stages.

6.2.2.2 Static HDL Profiling

Following after the Project Assembly is the Static HDL Profiling stage which parses the system and identifies the components and subcomponents of the system. This is accomplished through the `System Parser` tool. The BLAST core is identified to contain the following subcomponents: `plb_slave_ipif`, `plb_master_ipif`, `user_logic`, `blast_scan_ungap`, two `fifo_32_32` components, one `lookup_queue_64` component, and two `blast_bram_dual_port`.

From these components it is identified that the bus is the PLB, there is both a bus mater and a bus slave, there are 20 software addressable registers, and two individual FSMs. The first FSM is found within the `blast_scan_ungap` component, which is the primary FSM in the core with 17 states. The second FSM is found in the `user_logic` component and controls the bus master transactions when hits are detected, consisting of 14 states. This information will be used by both the Performance Monitor Insertion and Candidate Set Generation stages.

6.2.2.3 Component Synthesis

The next stage consists of synthesizing the individual components to identify resource utilization. Each of the resulting synthesis reports are then parsed to identify additional information about the component that were not readily identifiable via traditional VHDL parsing. Specifically, in the event of using components created by the Xilinx CoreGen tool. With the use of the `Parse PCORE` tool the components identified as black boxes during `xst` synthesis are matched up against those listed

Table 6.6: original BLAST hardware resource utilization (V4FX60)

Resource Type	Occupied	Total Available	% Used
Number of Slice Flip Flops:	1427	50560	2.82%
Number of 4 input LUTs:	3192	50560	6.31%
Number of FIFO16/RAMB16s:	12	232	5.17%

in the hardware core's BBD file. Then, the matching CoreGen project file for the component can be parsed and the additional parameters of the component can be identified. Namely, the width and depth of the component. For the BLAST core the aforementioned FIFOs were further identified to be: two 32-bit by 512 deep FIFO, one 64-bit by 512 deep FIFO, and two 32-bit by 2048 element BRAMs. In total, the BLAST core consumes twelve BRAMs. Table 6.6 lists a brief summary of the BLAST core's resource utilization.

6.2.2.4 Performance Monitors

Several performance monitors were identified for inclusion within the BLAST core based on the Static HDL Profiling and Component Synthesis stages. These include monitoring the PLB slave bus interface, the PLB master bus interface, the two FSMs, and the three FIFOs. In addition, utilization monitors are included to understand the efficiency of the BLAST implementation. What is of interest with these monitors is to understand how the newly accelerated BLAST application behaves with the two most time consuming functions ported to the BLAST hardware core. Also of interest is how the system behaves as it is integrated with the hardware filesystem in place of previous designs which connect the hardware core to off-chip memory and the Network Filesystem for database retrieval.

However, it should be stated that the focus of this case study truly is on the scalability of the system across the cluster of FPGAs. Therefore, measuring the existing performance on a signal node to identify over-utilized FIFOs and under-performing FSM and generating candidate configurations with larger FIFOs, but with a smaller scalability capability is less of a concern than those candidate configurations

that can scale the system as a whole. Moreover, during the cores lengthy design process as part of [96, 97], the FIFO sizes have been tuned to a satisfactory degree. Instead, the performance monitors will be inserted in the tree and torus topology candidate configurations. It should also be noted that the other case studies include the performance monitors with more detail.

6.2.2.5 Candidate Configuration - Torus Topology

While the Tree Topology implementation yields an astounding 230 parallel BLAST hardware cores, the dedicated network configuration does not allow the Spirit cluster to be used for other applications with a more general network connection requirement. Therefore, the goal with the Systematic Design Analysis flow is to generate an implementation that can both scale at least to the same number of parallel nodes and cores (if not greater) while reverting to a more traditional 3-ary 4-cube AIREN infrastructure.

As part of the Tree Topology analysis, it was discovered that the speedup achieved for databases with relatively large sequence sizes outperformed databases with relatively small sequence sizes. Upon investigation it was discovered that the issue was with reading in the sequence length from a separate file from the database. This meant two files were being read from and transferred to the BLAST cores. By reformatting the databases to include the sequence length followed by the sequence only one file would need to be opened. Of course this meant modifying the BLAST hardware core, which normally would be avoided because it required the designer; however, basic performance monitor analysis of the utilization of the BLAST core was less than 1% under the current implementation. Furthermore, modification only added a single state to the BLAST core's FSM while eliminating an entire FIFO. From a scalability perspective this means that the core now only consumes 11 BRAMs which could lead to an additional four BLAST hardware cores being implemented per node in the torus system.

The Systematic Design Analysis flow replaced many of the custom glue-logic components from the Tree Topology. Specifically, the custom broadcast core that is found on the head node and the intermediate BLAST nodes was replaced with a full crossbar switch. Furthermore, the logic to perform barrier synchronizations was removed because the AIREN network and crossbar switch provided better support for back pressure and pausing the database transmission from the hardware filesystem. Finally, the sheer number and complexity of the base systems (seven in total for the Tree Topology) can be reduced down to three when a software controlled routing controller for the crossbar switch. The specific details of these changes will be reported within the remainder of this section.

Figure 6.17 provides a high-level block diagram of the torus topology implemented on the Spirit cluster. Using the Systematic Design Analysis flow a more general implementation of the BLAST system can be constructed with minimal intervention from the designer. The trade off of moving from a physically wired network to match the application (as was done for the Tree Topology) to the more general network will also be evaluated. It is also worth noting that the Spirit cluster's default network topology is a 4-ary 3-cube torus, except that all 64-nodes are connected instead of the 32 presented in this work.

In order to evaluate the scalability of this system three types of nodes were constructed: head node, disk node, and BLAST nodes. Linux runs on all of the nodes to provide MPI [99] support. MPI is used to distribute the BLAST application to each node with BLAST cores. Each BLAST node then loads the query (or queries) into its own BLAST cores. Since all of the BLAST nodes contain a Linux system, the queries can be loaded in parallel unlike a conventional co-processor accelerator model where each query would need to be loaded sequentially. The Systematic Design Analysis flow is used in the creation of these node types; however, a base template has been used due to the complex details of the application. Specifically, the hardware file

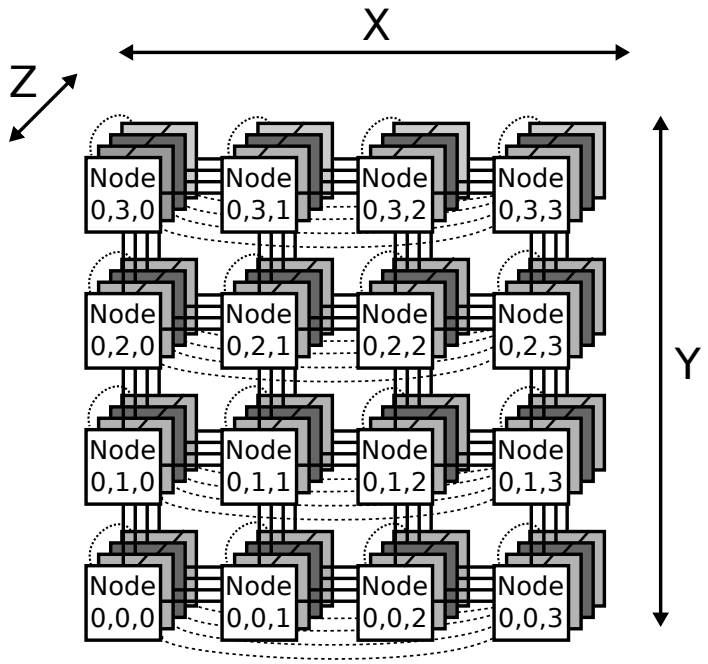


Figure 6.17: torus topology implemented on Spirit cluster

system's implementation on the head node and disk nodes extracted from the Tree Topology. The Systematic Design Analysis flow improved the intermediate glue-logic used to connect these components and to reduce the number of unnecessary node types.

Head Node The head node, shown in Figure 6.18, consists of common System-on-Chip (SoC) components (processor, memory controller, system bus, etc) for a fully running Linux 2.6 kernel on the 300 MHz PowerPC 405 processor as well as the Hardware Filesystem core, RAID controller core, and a broadcast hardware core. The head node is responsible for initiating the retrieval of the databases from the HWFS, as well as the initial broadcast to its children nodes in the network. The head node provides a central point for control and initial distribution; however, once the data distribution begins, the parallel connectivity of the AIREN network enables decentralized processing, dramatically reducing the sequential performance bottleneck of processor-centric systems.

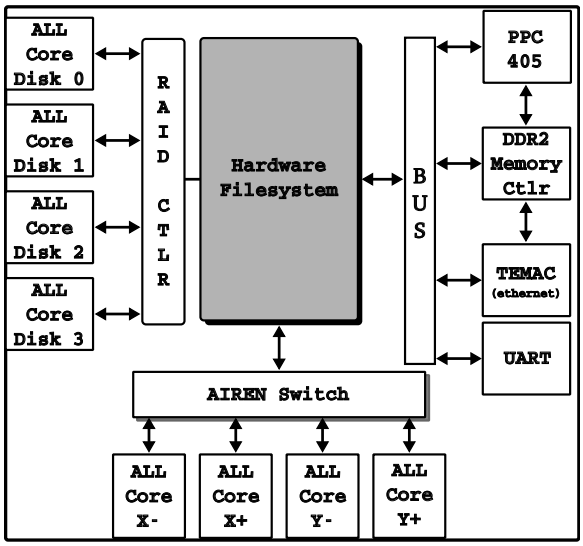


Figure 6.18: torus topology's head node

Disk Node Figure 6.19 illustrates the disk node which contains a single bidirectional AIREN Data Link Link network interface to the head node's RAID controller, and an interface to a RAM Disk (512 MB per RAM Disk). There is no additional BLAST logic on the disk node due to resource limitations. The filesystem is loaded on each node's RAM Disk from CompactFlast when the disk node is powered on and resides there until the system is shutdown. For this work, the RAM Disk is used in place of conventional disks due to unimplemented SATA disk controllers. While work is currently underway to provide such support, the focus of this report is on the complete integration of the HWFS, AIREN network and an application hardware core. Functionality is of critical concern as opposed to direct performance comparisons between general purpose clusters.

BLAST Node The BLAST node used in the torus topology, which can be seen in Figure 6.20, includes a fully running Linux SoC along with four ALL cores to send and receive the database from the head node and/or other BLAST nodes, the AIREN switch, and up to sixteen BLAST hardware cores. The AIREN switch is configured for each node during boot and can be changed by software to support testing the scalability of the system. This implementation is an improvement upon [100] based on

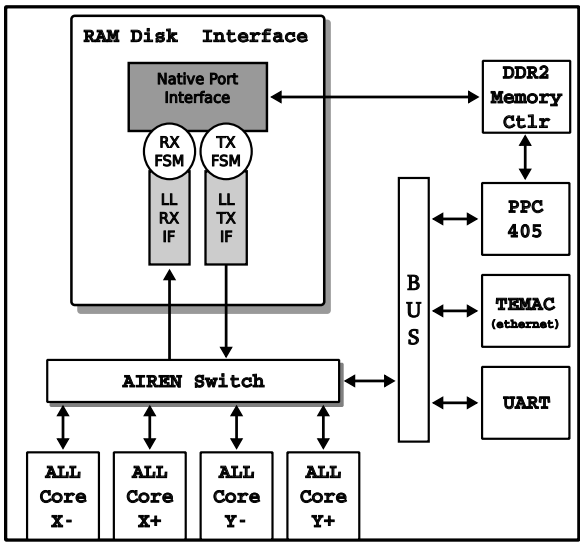


Figure 6.19: torus topology's disk node

several performance and utilization analysis. First, the utilization of BRAM limited the scalability of the number of cores on a single node. Second, a more efficient interface to the system bus reduced contention for look-ups to main memory. Finally, a simple modification to the storage of the databases in the Hardware Filesystem improved I/O bandwidth. These improvements were able to be quickly identified when monitoring the performance of the initial tree topology.

To aid in a designer's productivity, the entire system is developed around the application hardware core, in this case BLAST. Therefore, the designer first focuses on porting the algorithm to hardware. Interfaces to the processor, main memory, secondary storage, and the network can be abstracted away from the designer and are constructed for the application. This is one of the key benefits of the Systematic Design Analysis flow. For BLAST, the network interface which connects with the HWFS is of critical concern. If the network is unable to sustain the bandwidth the HWFS is capable of providing, then no matter how efficient of a BLAST compute core, it will remain underutilized. This concern is not for the designer of the application hardware core, instead it is for the system designer.

Figure 6.21 shows the data flow of the torus topology. The head node connects

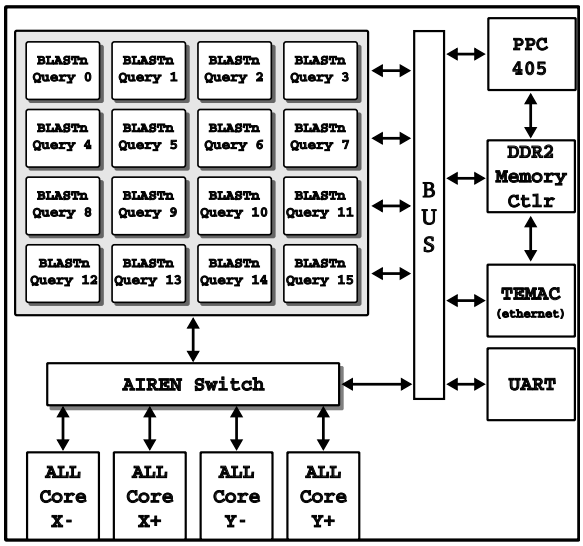


Figure 6.20: torus topology's BLAST node

to four BLAST nodes through the AIREN network. Each of the four BLAST nodes transfer the database to their children and to their local BLAST cores. The AIREN switch is configured for each node specifically to transfer the database so that each node only receives the database once. Only four channels of the AIREN network are used per node to minimize resource utilization. With four channels, every node can receive data within four hops from the head node. This could be reduced to three with the addition of an additional channel, but at the cost of two BLAST cores. The results from the tree topology [100] indicated that the number of hops between nodes played only a small role in the performance of the system due to the very low latency of the AIREN network.

6.2.3 Results and Analysis

The last step is to compare the two implementations to determine if the Systematic Design Analysis flow was able to provide comparable performance with a more generic infrastructure.

6.2.3.1 Experimental Setup

Table 6.7 represents the five databases tested in terms of their total size and the number of sequences [101]. While it is not feasible to include every database and

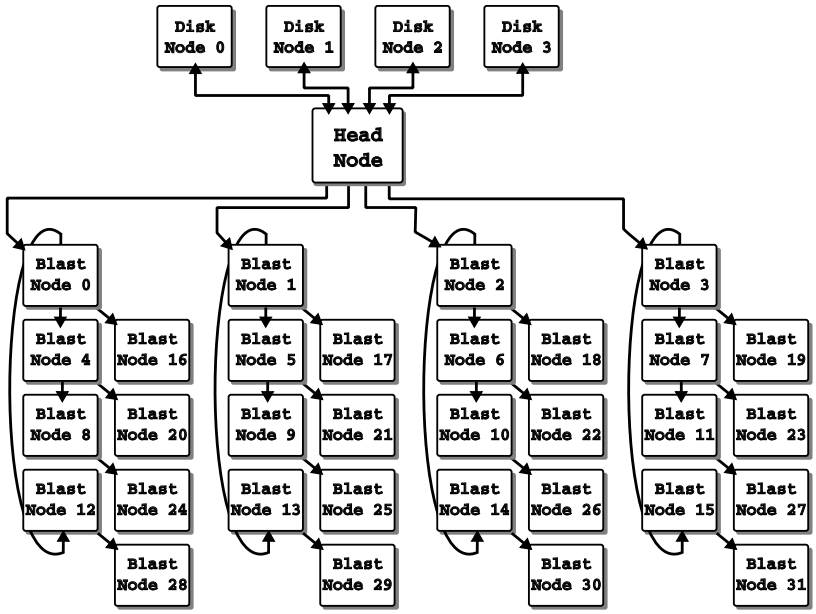


Figure 6.21: dataflow of torus topology configuration

query currently in existence in these experiments, these are chosen to both highlight the strengths of the BLAST hardware core (large sequences with relatively few hits) as well as the current weaknesses (small sequences with many hits). In configurations tested with more than a single query running in parallel, each system is running with the same query. As the authors have been unable to locate any currently accepted benchmarks, existing peer-reviewed publications [96, 97] must suffice. The impact of testing with the same query on all BLAST cores is that for each hit, all cores will issue their individual lookups to off-chip memory at the same time. Since there is only a single channel of off-chip memory on the Xilinx ML410 development board this will result in sequential access for all lookups. As the number of parallel cores increases per node, the contention for memory will increase with these experiments. Under more realistic loads where every core is performing different queries in parallel at any given time a hit may or may not occur on each core. This would result in less contention for the memory channel. To be as unbiased as possible we have chosen to use the same query to provide a worst case scenario for our system.

The original size of the databases refers to the database size when downloaded

Table 6.7: databases used in experiments

Database Name	Original Size (MB)	Formatted Size (MB)	Number of Sequences
env.nr	1,726	117	6,027,398
env.nt	2,782	174	167,450
month.nt	719	698	2,799,207
est_human.nt	5,045	1,093	30,565
human_genomic.nt	1,536	1992.63	30,565

Table 6.8: query (1) 248 (2) 4,292 and (3) 14,216 bytes number of hits

Database Name	Number of Hits Query 1	Number of Hits Query 2	Number of Hits Query 3
env_nr	226,576	5,262,389	22,916,040
env_nt	831,680	15,572,380	57,203,714
month	3,650,764	63,179,552	233,175,103
est_human.nt	4,374,004	107,987,009	371,148,957
human_genomic.nt	6,732,670	158,369,822	(n/a)

from [101]. The databases are then formatted which removes unnecessary data (such as protein information which BLASTn does not process). While the exact format follows NCBI, the formatting is done to support the BLASTn hardware core. Each database only needs to be formatted once. The time to format each database is not included in any test, including the software implementation. For the torus topology experiments Table 6.8 reports the number of hits that are expected for three query set sizes of 248 Bytes, 4,292 Bytes and 14,216 Bytes. The Tree implementation was only tested with queries 1 and 2, also human_genomic.nt is used in place of est_human.nt for the tree implementation due to size restrictions on the tree implementation's HWFS storage capacity.

6.2.3.2 Tree Topology Results

To begin, the tree topology is compared to a single, unmodified BLAST implementation in order to better understand how the system scales. In Figure 6.22(a) and Figure 6.22(b) the speedup of a single BLAST core is compared with a fully utilized single node, five node, and 21 node implementation with 10, 54, and 230 BLAST cores respectively, connected in a tree topology.

When scaling up to the five node system with 54 total BLAST cores the geometric mean speedup is 26.16× and 7.97× that of the single node, single core implementation for query size 248 and 4,292 respectively. Compared to the ideal speedup of 54×, the 5 core system is 48.4% and 14.8% efficient, depending on the query. Further scaling to 21 nodes with 230 BLAST cores results in a geometric mean speedup of 108.1× and 35.2× the single core implementation and is 47.0% and 15.3% efficient.

To understand why the scalability is limited to ≈50% for query 1 (248 bytes) and ≈15% for query 2 (4,292 bytes), consider the speedup of one fully utilized BLAST node (with ten BLAST cores) to one node with one BLAST core, shown in Table 6.9. Scaling to ten cores should ideally result in a 10× speedup. Instead the speedup ranges from 1.52–8.65×. This is due in part to the size of the database, number of sequences in the database, and the number of hits based on the query. Recall that the BLAST hardware core has been designed with an on-chip cache for identifying hits. In the event of a hit, the core issues a lookup to another table stored in off-chip memory. When there is only one BLAST core on the node, it has no contention to accessing off-chip memory. As the number of cores increases, so to does the contention. The result is that query 1 is 52.3% efficient and query 2 is 22.5% efficient.

Alternatively, the performance of the system can be considered in terms of scalability of nodes. That is, by increasing the number of nodes, how does the performance scale? Figure 6.22(c) depicts the speedup from a single fully utilized FPGA node to five nodes to 21 fully utilized FPGA nodes. With the tree network, data can be streamed from a head node to its children with a relatively few number of hops (one hop for five nodes and two hops for 21 nodes). With the AIREN network's low latency per hop and large bandwidth per channel, the geometric mean speedup for the five node system is 5.0× and for the 21 node system is 20.92×, or linear speedup. This is a favorable result for scientists. Increasing the number of nodes increases the number of queries that can be run in parallel without requiring the database to be

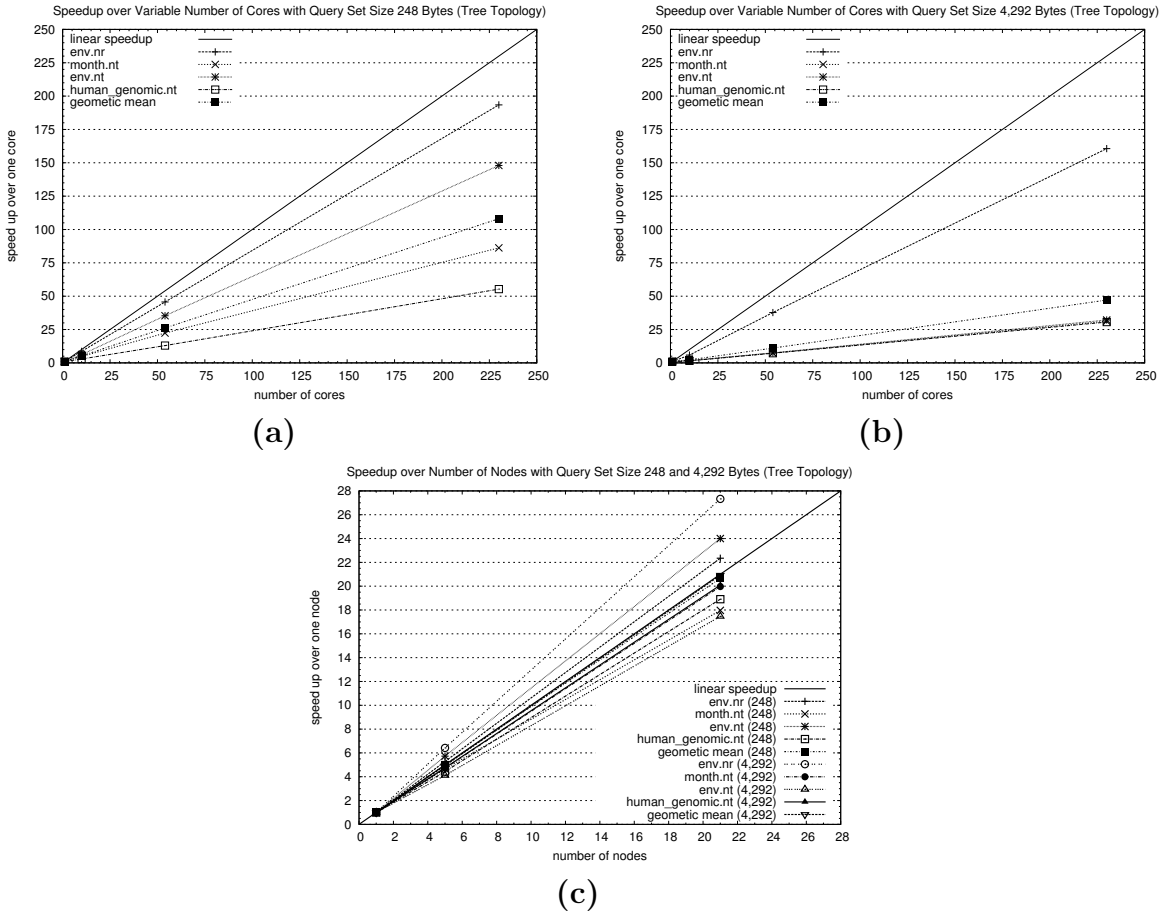


Figure 6.22: speedup of 1 node-1 query vs. 1 node-10 query, 5 node-54 query and 21 node-230 query on tree topology with query size of (a) 248 Bytes (b) 4,292 Bytes and (c) speedup of 1 node-10 query vs. 5 node-54 query and 21 node-230 query on tree topology with query size of 248 and 4,292 Bytes

re-broadcast through the system.

6.2.3.3 Torus Topology Results

The torus topology consists of one head node, four disk nodes, and 32 BLAST nodes. The head node connects to all four disk nodes and to four of the BLAST nodes. To investigate the scalability of the torus topology tests were run with one, four, eight, 16 and 32 nodes. Figure 6.23(a), Figure 6.23(b), and Figure 6.23(c) show the scalability results for 16, 64, 128, 256 and 512 hardware cores over one core. The performance does not scale linearly with the number of hardware cores due to the memory bottleneck for a single node. Furthermore, maximizing the number of

Table 6.9: speedup of 1 BLAST node-10 query (1-10) over 1 BLAST node-1 query (1-10) with tree topology BLAST node implementation with (1) 248 byte and (2) 4,292 byte query sets

Database	Query 1	Query 2
env.nr	8.65×	5.88×
month.nt	4.80×	1.56×
env.nt	6.17×	1.84×
human_genomic.nt	2.92×	1.52×
Geometric Mean	**5.23**×	**2.25**×

BLAST cores (16 per node) resulted in additional contention for main memory when performing lookups, which yielded less efficient performance scaling. However, this limitation is an artifact of how the design was being tested. Namely, that each core on the same node were running the same query. This resulted in the worst case memory bandwidth performance and greatest amount of contention for the single memory channel. Of the three queries evaluated, query 1 performed the best with ≈30% of the expected speedup achieved as the number of cores scaled to 512. Query 2 reaches ≈11% and query 3 performed at ≈10%.

However, when analyzing the overall system performance, as shown in Figure 6.23(d), linear speedup is still maintained when scaling up to 32 nodes. The torus topology only increases the number of hops the database needs to make from three for the tree topology to four. The additional hop increases the latency by just an extra 0.8μs for the last node. Furthermore, by reformatting the database the scalability of the system with respect to small and large sequence is more uniform.

The goal to scale an application hardware core across an *all*-FPGA cluster was realized in both the tree and torus topologies. A total of 512 BLAST hardware cores were implemented on just 32 FPGAs, with 16 cores per FPGA. Considering that the Xilinx ML410's Virtex4 FX60 FPGA is now three generations old and has between 5–25% of the resources of current FPGA, being able to scale to 16 cores is an impressive feat. However, it is important to stop and ask, *what is the ideal number of cores needed on each node?* It is easy to get caught up in putting as many hardware cores

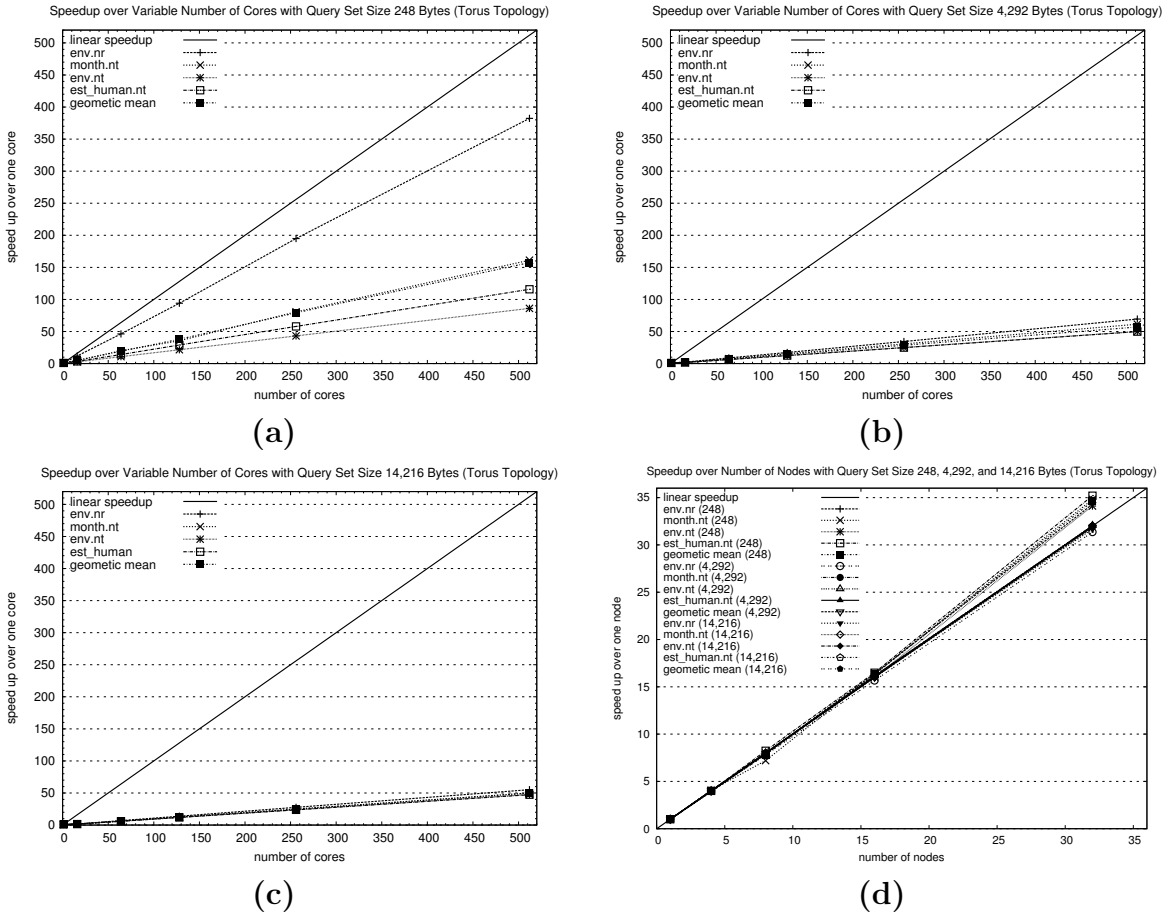


Figure 6.23: speedup of 1 node-1 query vs. 1 node-16 query, 4 node-64 query, 8 node-128 query, 16 node-256 query and 32 node-512 query on torus topology with query size of (a) 248 Bytes (b) 4,292 Bytes (c) 14,216 Bytes and (c) speedup of 1 node-16 query vs. 4 node-64 query, 8 node-128 query, 16 node-256 query and 32 node-512 query on torus topology with all queries

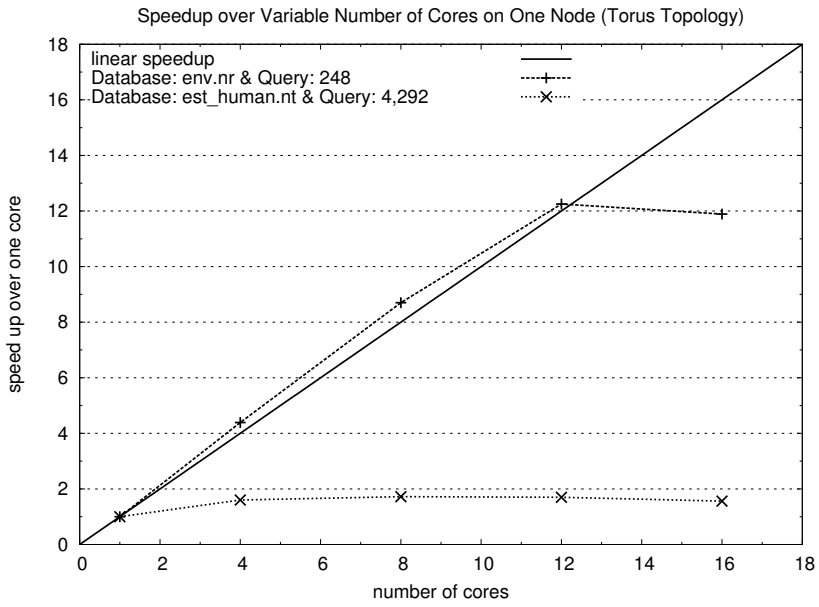


Figure 6.24: speedup of on a single node when incrementing the number of BLAST queries for env.nr with 248 byte query and est_human.nt with 4,292 query

on a single FPGA as possible. Our motivation to do so was based on initial results gathered in [96, 97] which pointed towards increased performance with an increasing number of FPGAs.

A quick experiment was run to further investigate the effects of scaling BLAST cores on a single node. Figure 6.24 shows the speedup of running the 248 byte query on `env.nr` and the 4,929 byte query on `est_human.nt`. The first query and database was chosen because it has the fewest number of hits which is best suited for the BLAST hardware core. The second query and database was chosen because it had a large number of hits. As the number of cores increases there is a local maxima between 8 and 12 cores. The results also shows the performance gap between the two query types. Work is currently underway that will close this gap, although no results are presented here.

Another experiment was run on the torus topology where each BLAST node is running with 8 BLAST cores, shown in Figure 6.25. This is to compare the scalability

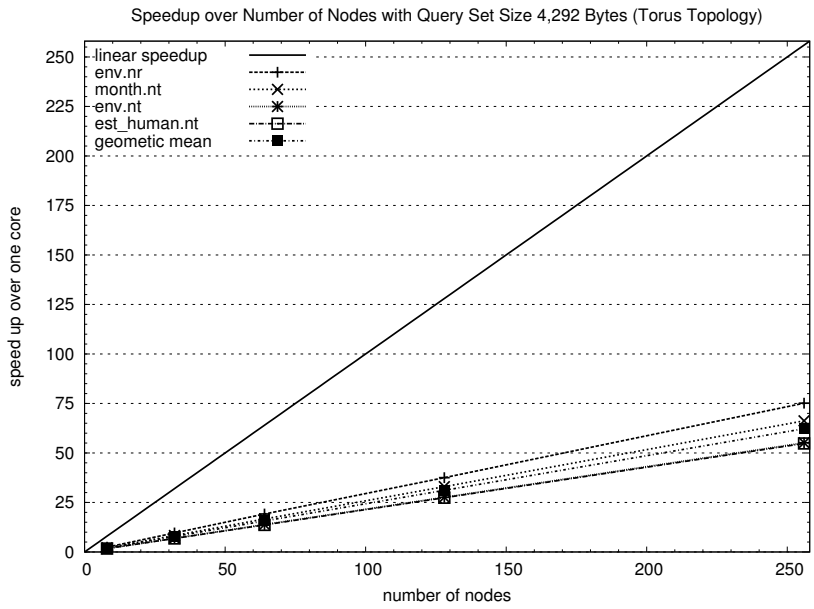


Figure 6.25: speedup of 1 node-1 query vs. 1 node-8 query, 4 node-32 query, 8 node-64 query, 16 node-128 query and 32 node-256 query on torus topology with query size of 4,292 Bytes

of cores when there is less memory contention (half in this experiment). The results show that the systems achieve approximately ≈25% of linear speedup as the number of cores scale to 256 cores. This is an improvement over the ≈11% achieved when there were 16 BLAST cores per node.

Finally, it is worth noting that since the queries can differ so greatly with the number of hits it is impossible to definitively state which system or which topology is best for all cases. The results accumulated here only show that the current implementation of the BLAST hardware core is better suited for large databases and queries with a small number of hits. The most encouraging result is that with the addition of the Hardware Filesystem to support directly transferring the database without any processor involvement and the AIREN network, scaling the BLAST cores across the *all*-FPGA cluster can be done with minimal overhead while maintaining linear speedup with respect to the number of nodes in the system when increasing the number of nodes.

6.2.4 Observations and Summary

To summarize, the BLAST case study provided many useful observations and significant feedback about the Systematic Design Analysis flow. Overall, the flow enables a designer to better understand how a system, such as BLAST, scales with different data sets, under different network topologies, and with increasing number of resources. Most interesting is the result that generated torus implementation was able to scale to more parallel BLAST cores (512 vs. 230) even though the design was optimized around a more generic network topology (torus vs. tree). The overall speedup achieved by the 32 node torus topology when compared to a fully utilized single node was 31.92×, as compared to the 21 tree topology which achieved 20.92× speedup. In addition to the performance gains, Table 6.10 highlights several additional observations of this case study.

6.3 Case Study: Smith/Waterman Algorithm

The Smith/Waterman algorithm is commonly used in protein and nucleotide sequence alignments [95]. Specifically, the algorithm compares two sequences for similarities and reports the optimal alignment to the user. With gapped extension additional analysis is performed to identify the cost to transform one segment of a sequence into another segment. This is accomplished through substitutions, insertions, and deletions. Therefore, one sequence can be converted into another and the *cost* to accomplish this will yield information regarding the similarity of the sequences. Ideally, the smaller the cost means the fewer differences or operations were required, which indicates a more closely related sequences. Figure 6.26 demonstrates the optimal alignment of two sequences, `CACCCAGC` and `CTACACAC`.

The particular implementation on the FPGA was developed as a proof of concept for implementing Smith/Waterman on a single node of the Reconfigurable Computing Cluster's Xilinx ML410 development board [102]. Specifically, the hardware implementation is trying to accelerate the `FLOCAL_ALIGN()` function that is part of the

Table 6.10: summary of BLAST case study

Stage	Details
Static HDL Profiling	correctly identified both PLB master and PLB slave interfaces found all 20 software addressable registers found all three FIFOs found both BRAMs found both FSMs (17) state and 14 state)
Component Synthesis	identified resource utilization of all components scalability limited by BRAM utilization (11 BRAMs/core)
Performance Monitors	eight monitors inserted identify input FIFO was never full in tree topology lookup and output FIFOs are never full BLAST core utilization less than 1% in tree topology elimination of software barrier yielded ≈100× speedup fully utilized system required removal of several monitors monitoring tree topology identified unoptimized software
Candidate Configurations	thirteen configurations evaluated scaled tree topology to 230 cores across 21 nodes scaled torus topology to 512 core across 32 nodes evaluated underutilized eight core system on 32 nodes
Performance Evaluation	tree topology obtained 20.92× speedup over 1 node torus topology obtained 31.92× speedup over 1 node eight core design obtained 62.27× speedup over 1 core
Other Observations	torus topology used fewer resources than tree topology scalability with torus topology greater than tree torus topology used 3 base systems (vs. 7 for tree) best performance from 8 cores per node (vs. 16) performance monitors identified unoptimized software improving software resulted in reduction in BRAM utilization

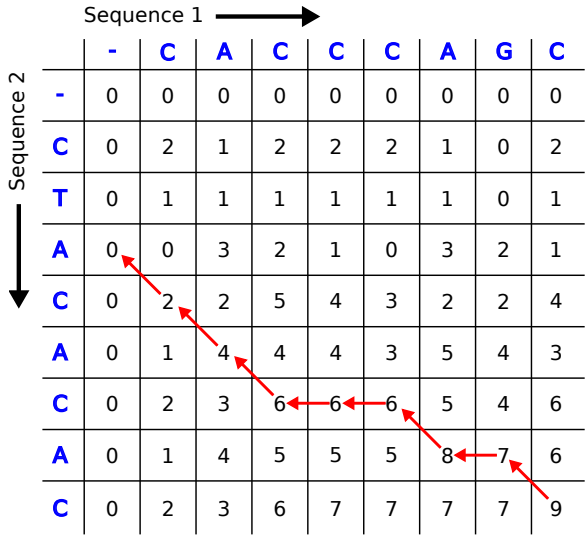


Figure 6.26: example of Smith/Waterman optimal alignment of two sequences

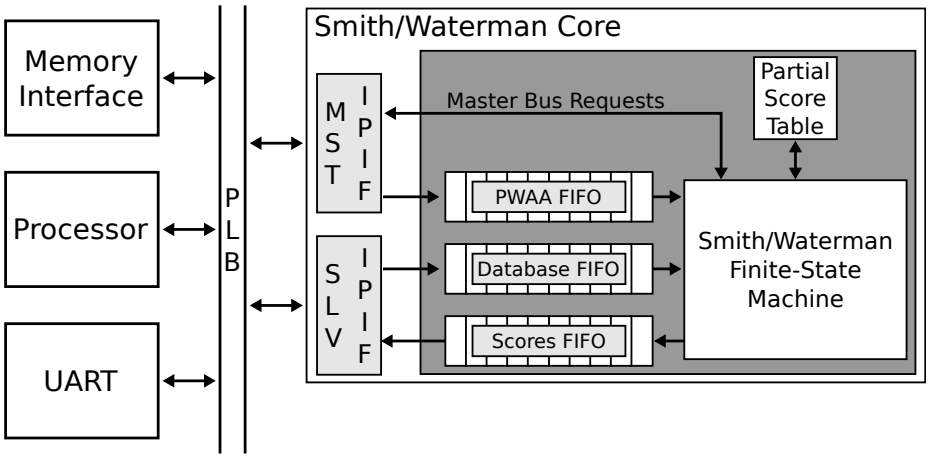


Figure 6.27: Smith/Waterman core top-level entity block diagram

SSEARCH program which is an implementation of the Smith/Waterman algorithm from the *FASTA35* code package [103]. In this hardware implementation the processor on the FPGA transfers state to the hardware core to perform the `FLOCAL_ALIGN()` which was identified as being the most computationally intensive function in *SSEARCH*. The hardware core designed is not optimally utilizing all resources in the system; therefore, it provides an excellent test bench for performance analysis and information feedback.

6.3.1 Design

This section briefly highlights the design running on the FPGA, a fully detailed description of the implementation can be found in [102]. Figure 6.27 illustrates the top-level entity block diagram for the Smith/Waterman hardware core. The hardware core operates by streaming in a database of sequences which are compared against a substitution matrix stored in off-chip memory. To compare against a query, the database is read by the hardware core and the substitution matrix is retrieved so that row-by-row the matrix can be compared with the database elements. During this process the partial score table is updated based on the database entry and the substitution matrix. Once the database has been consumed the optimal score is returned to software to be used to calculate the optimal alignment.

The core is comprised of several peripherals including interfaces to the bus, internal buffers, local storage, and a central FSM to perform the `FLOCAL_ALIGN()` operation in hardware. Specifically, there is one slave interface to the PLB which responds to requests from the processor to read and write to the Smith/Waterman hardware core. In this particular hardware implementation the database is written by the processor to the `database FIFO`. Also, the processor reads the final scores back from the `score FIFO` through the SLV IPIF. There is also one master interface (MST IPIF) which grants the hardware core access to the bus to perform bus transactions. The Smith/Waterman core uses the MST IPIF to retrieve the substitution matrix scores from off-chip memory. The Smith/Waterman uses the local storage to hold the partial score table that is updated during the calculation of each sequence.

In addition to the Smith/Waterman hardware core there are an assortment of hardware cores to support a fully functional Linux System-on-Chip. This includes the processor, memory interface to off-chip memory, fast Ethernet network interface, and UART for console I/O. The system boots Linux 2.6 through the Network Filesystem (NFS) and provides remote access via Remote Shell (RSH) from within the Reconfigurable Computing Systems (RCS) lab. The `FASTA35 SSEARCH` application has been cross-compiled for the PowerPC 405 and only the `FLOCAL_ALIGN()` function has been modified to transfer state from the application running in software to the Smith/Waterman hardware core.

To evaluate the design the existing testing infrastructure used during the original evaluation of the Smith/Waterman hardware core has been reused [102]. This includes the same query that was randomly assembled to form a sequence of 140 characters from the amino acid set. To this five databases have been evaluated ranging in size from 200 KB to 2 MB:

- MY_DB_LONG
- YEAST_REDUCED.AA

- MY_DB_EXPANDED
- MITO_REDUCED.AA
- MY_DB_LARGE

6.3.2 Implementation

The Smith/Waterman hardware core has been selected because unlike BLAST, where previous work has been done to evaluate the scalability of the system, this core has not been optimized for resource utilization nor has it been ported to other FPGA devices. The Systematic Design Analysis flow can increase the designer's productivity by evaluating the current status of the core and to generate a set of candidate configurations without requiring any modifications by the designer to the original core.

6.3.2.1 Project Assembly

In the Project Assembly stage the Systematic Design Analysis flow takes the existing project in the XPS directory structure and generates the subsystems that will be used throughout the remainder of flow. The first tool used is the `Generate Systems` tool. This tool is described in full details in Chapter 4. The primary functions are to run PlatGen and to construct the `RCS_TOOLS` subdirectory which contains the necessary HDL and synthesis scripts to be used in the Static HDL Profiling and Component Synthesis stages.

6.3.2.2 Static HDL Profiling

After Project Assembly is the Static HDL Profiling stage which parses the system and identifies the components and subcomponents of the system. This is accomplished through the `System Parser` tool, the details of which are described in Chapter 4. The Smith/Waterman core is identified to contain the following subcomponents: `plb_slave_ipif`, `plb_master_ipif`, `user_logic`, three FIFOs (`fifo_32x128`), and one BRAM (`sw_bram_rst`). From these components it is identified that the bus is the PLB, there is both a bus mater and a bus slave, there are 15 software addressable registers,

a 29 state FSM, and a four state FSM both within the `user_logic` component. This information will be used by both the Performance Monitor Insertion and Candidate Set Generation stages.

6.3.2.3 Component Synthesis

The next stage is Component Synthesis where the system is synthesized to identify the specific resources that are used along with providing the Systematic Design Analysis flow a total resource count for future scalability calculations. This is accomplished through the `Iterative Component Synthesis` tool which walks through and synthesizes a component from the bottom up. This tool is not perfectly accurate because the resources reported are based on pre-placed-and-routed circuits; however, it does provide a rough guide for the scalability of the system. The Smith/Waterman core uses 1719 slice flip-flops (FFs), 3485 4-input LUTs, four BRAMs, and one DSP48 slice. These numbers will be used in the Candidate Set Generation and Selection stage to determine the scalability of the system as well as to help understand the feasibility of other transforms, such as FIFO replacement.

6.3.2.4 Performance Monitor Insertion

From the Static HDL Profiling and Component Synthesis stages several performance monitors were identified for inclusion into the Smith/Waterman hardware core. At this time the insertion of the specific monitors is done manually; however, the modifications to the PCORE and to the whole system to support the performance monitoring infrastructure have been automated. More details regarding this process can be found in Chapter: 4. Figure 6.28 shows a high-level block diagram of the performance monitors in their respective locations to the Smith/Waterman hardware core. In addition to the monitors the Performance Monitor Hub and the CIF are added to the hardware core to connect to the Performance Monitoring infrastructure such that the data can be retrieved with minimal invasion to the hardware core.

The first two performance monitor inserted are the Processor Local Bus Slave and

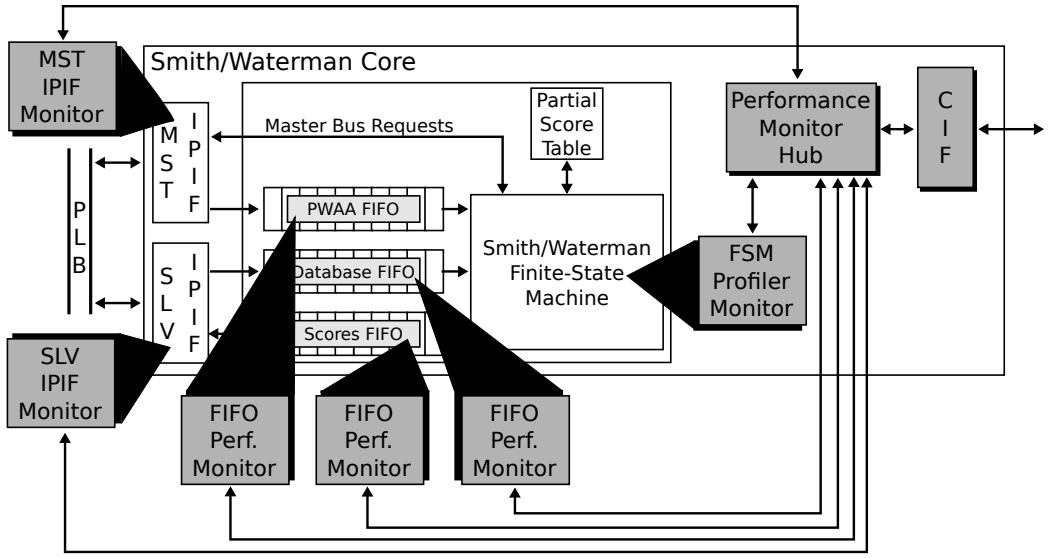


Figure 6.28: Smith/Waterman core's performance monitors

Mater Intellectual Property Interconnects, PLB SLV IPIF and PLB MST IPIF respectively. The PLB SLV IPIF monitor is used to count the number of reads and writes to the software addressable registers within the hardware core. In the Smith/Waterman Core there are 15 such software addressable registers. The PLB MST IPIF monitor is used to count the number of bus master transactions performed by the core along with the address range the requests are being made to. This is to determine if the volume of requests is sufficient and the destination is off-chip memory, if so the Generate Candidate Configurations stage will look to directly connect the hardware core to off-chip memory. By directly connecting to main memory the hardware core has lower latency and higher bandwidth access to main memory. Furthermore, the PLB currently puts limitations on the size of the transfer that can be performed across the bus, DMA to off-chip memory does not have that fundamental limitation.

The next performance monitors inserted are a set of FIFO Utilization monitors. The Smith/Waterman core employees three FIFOs, initially set to be 32-bits $\times$ 128 elements deep with one for the input database, one for the substitution matrix, and one for the output scores. The FIFO Utilization monitors counts the amount of time the FIFO is asserted to be full along with the total number of reads and writes to

the FIFO. In the event a particular FIFO is often full, the Candidate Configuration Generation stage can automatically replace the FIFO with one of a larger capacity. The reads and writes are used to determine if the data in the FIFO does in fact get completely consumed. Not all cores require this check; however, as will be shown in Section 6.3.3, this performance monitor can identify logic errors in the existing hardware core.

The sixth and final performance monitor is the FSM profiler. The Smith/Waterman core consists of 29 states which for brevity are excluded from this work, yet can be found discussed in [102]. The FSM profiler is used to identify the amount of time spent in each of the states. This information is presented back to the user as percentages of the overall execution time of the FSM such that the designer can better understand which states consume the most time.

6.3.2.5 Performance Monitor Evaluation

The performance monitor data is collected through the Performance Monitor Infrastructure presented in Chapter 4. The individual node under test is connected to a centralized head node which retrieves the performance data. This data is then output to a file for future analysis. This section details the results of the aforementioned performance monitors prior to any manipulations being made to the hardware core as part of the Candidate Configuration sets.

The first performance monitor identifies the number of writes to the software addressable registers in the Smith/Waterman hardware core, the results of which are listed in Table 6.11. In addition to the register name, number of reads and number of writes, Table 6.11 also presents these reads and writes when run in *Original* and *Modified* modes. Both modes refer to whether the software application running is in its original formation, meaning no changes have been made to the application, or if the application has been modified. The specific modification is shown in Figure 6.29 where the missing `only_once` guard has been added.

```
ssearch->aa1 = *aa1p;
if (only_once == 0) {
  ssearch->n1 = n1;
  ssearch->n0 = n0;
  ssearch->GG = GG;
  ssearch->HH = HH;
  ssearch->f_str_waa_s = PWAA_
BASE;
  ssearch->ssj = SS_BASE;
  /* AGS: Added Missing Guard HERE */
  only_once = 1;
}
```

Figure 6.29: modification made to original dropgsw2.c

Table 6.11: performance monitor results for PLB SLV IPIF

Register Name	**Original**		**Modified**	
	# Reads	**# Writes**	**# Reads**	**# Writes**
control_reg	0	186	0	186
core_status	29540	0	29540	0
aa1	0	2095745	0	2095745
n1	0	2095838	0	93
n0	0	2095838	0	93
GG	0	2095838	0	93
HH	0	2095838	0	93
f_str_waa_s	0	2095838	0	93
score	186	0	186	0
ssj	0	2095838	0	93
slv_reg10	0	0	0	0
slv_reg11	0	0	0	0
counter_idle	186	0	186	0
counter_work	186	0	186	0
counter_bram_reset	0	0	0	0

During the original evaluation the PLB SLV IPIF performance monitor identified several registers were being written to the same number of times. While this might be by design, a quick search over the source code found that the previously mentioned guard was in fact missing for the source code. After adding the guard into the code the application was then run again to verify correctness. The results of this modification are a significant reduction in the number of unnecessary writes to registers by the processor which reduces the application's runtime by a third. More details will be discussed in Section 6.3.3.

The second performance monitor evaluates the PLB Master interface (PLB MST IPIF). Since each database under evaluation may require a varying number of transfers the `MY_LONG_DB` was selected for evaluation. The PLB MST IPIF identified that only off-chip memory transactions were performed between the address range of `0x00000000` and `0x1FFFFFFF`. Furthermore, the transactions were read-only requests (meaning no data was written back out to off-chip memory). Finally, the number of transaction recorded for the database was 118,144. This is a significant number of transfers and warrants the evaluation of a DMA interface.

The next set of performance monitors evaluated the utilization for the three FIFOs in the Smith/Waterman core. Considering that each FIFO was by default configured as a 32-bit $\times$ 128 element deep FIFO, it is estimated that the FIFOs may be restrictively small for the application. Furthermore, when considering the Xilinx Virtex4 FX60 FPGA a FIFO of this size would be implemented in a BRAM; however, each BRAM on the FPGA supports up to 32-bit $\times$ 512 elements. This means that even though the designer specified to use only a 128 deep element, the resources used are the same for up to a 512 deep element. The performance monitor data shows that the database input FIFO does in fact reach a full state for 65,430,077 clock cycles (100 MHz) when evaluated with even the smallest database under test. Therefore, the FIFO is being over utilized and can be increased in capacity during the Candidate

Generation stage. The other two FIFOs did now show any indication of being full.

In addition, the FIFOs were checked for the number of read and writes to determine if all of the data written into the FIFOs was being consumed. Unfortunately, this performance monitor found another logic error in the application by discovering the database is not being read out in its entirety. More analysis will be presented in Section: 6.3.3.

The final performance monitor evaluated is the FSM profiler. The FSM profiler generates feedback in the form of a histogram to identify the percentage of time each state is active in the FSM. Figure 6.30 presents the breakdown of the time each states spends when running with the `MY_DB_LONG` database. From this it can be seen that the longest running state is the `BUILD` state occupying 27.67% of the execution time. The next four states, BUILD_SCORE, READ_SSJ, BUILD_SWITCH, POP_PWAA, each occupy ≈13.8%. Thirteen of the remaining states occupy less than 1% and have been group together in the `OTHERS` category. Overall, this profiling data should more quickly focus the designer's attention on the `BUILD` state to determine if there is a more efficient implementation of the specific state.

6.3.2.6 Candidate Set Generation, Selection, and Evaluation

From the Static HDL Profiling and the Performance Monitoring data the following sets of configuration were identified.

The first candidate is the software modification that was identified during the PLB SLV IPIF performance monitor evaluation with the addition of the `only_once` guard. The results of this configuration are prevalent in Table 6.11. This configuration requires no hardware modifications and can be included in with the remaining configuration candidates.

The second candidate configuration comes from the results from the PLB MST IPIF performance monitor. The number of bus master transactions to main memory indicates that it would be favorable to augment the Smith/Waterman core to include

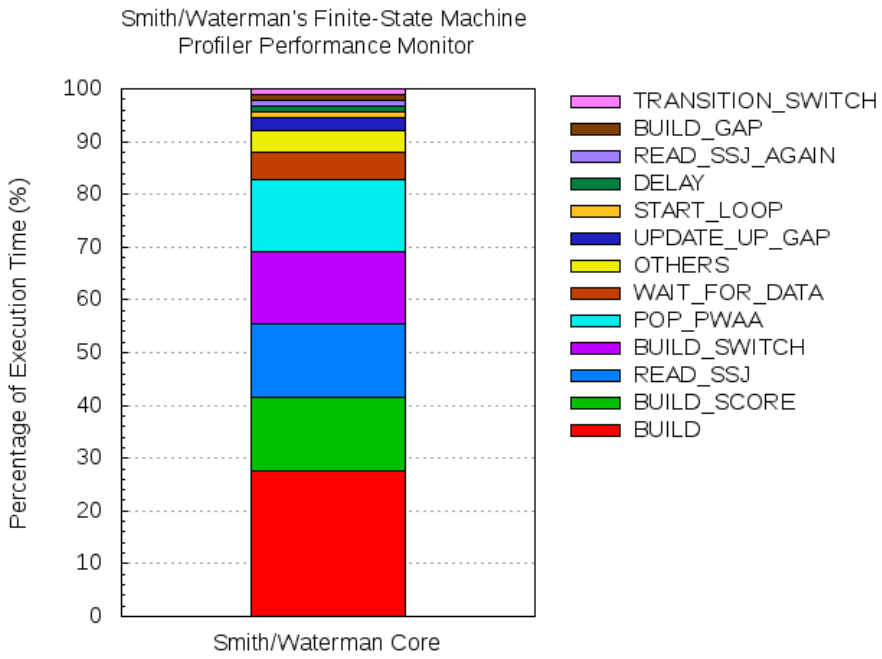


Figure 6.30: Smith/Waterman core's FSM profiler monitor results

a DMA port to off-chip memory. This is depicted in the high-level block diagram in Figure 6.31. The Smith/Waterman core is modified with the use of the `Bus Master to DMA` tool which parses the top-level entity and replaces the PLB Master IPIF components and replaces it with a DMA component. More details regarding this tool can be found in Chapter 4. For this core specifically, the DMA component translates bus master requests into native port interface (NPI) commands which are then directly sent to the off-chip memory controller.

Finally, the size of the database FIFO was identified to be restrictively small and through the use of the `FIFO Replacement Tool`, described in full in Chapter: 4. The FIFO component is identified during VHDL parsing and the corresponding Xilinx CoreGen component is augmented to support a deeper FIFO size. For this candidate configuration set the depths of 128, 512, 2048 and 4096 elements have been evaluated. The FIFO increase is part of both the bus master and DMA interface configurations.

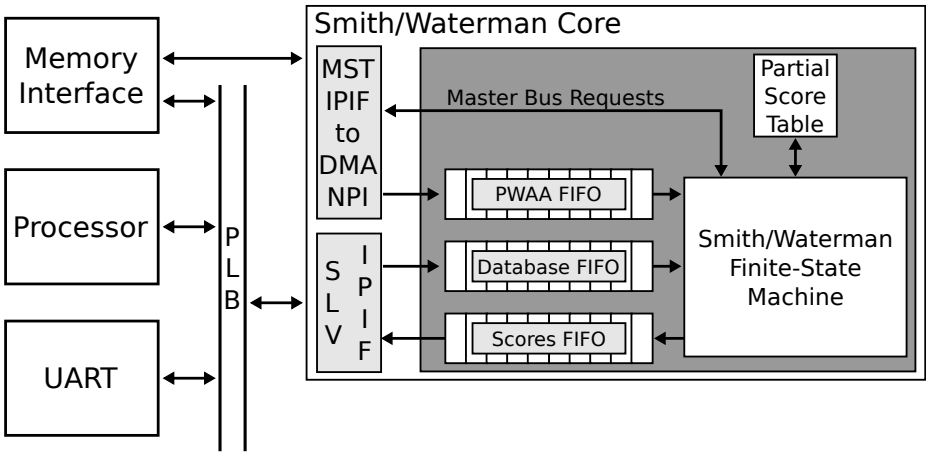


Figure 6.31: Smith/Waterman core's DMA interface

Table 6.12: databases used in evaluation of Smith/Waterman core

Database Name	**Database Size**
MY_DB_LONG	204 KB
YEAST_REDUCED.AA	384 KB
MITO_REDUCED.AA	384 KB
MY_DB_EXPANDED	792 KB
MY_DB_LARGE	2 MB

6.3.3 Results and Analysis

To evaluate the candidate configurations generated as part of the Systematic Design Analysis flow five databases have been selected ranging in size from 200 KB to 2 MB are shown in Table 6.12. The query used chosen is a random 140 amino acid character sequence, `my_small_query`. Conventionally, a more thorough analysis with more queries and database would be preferred — similar to the BLAST evaluation; however, for reasons that will be discussed shortly, this initial set proves to be sufficient. Furthermore, these are the same databases evaluated during the original evaluation of the Smith/Waterman core [102].

As part of the Systematic Design Analysis flow the original design's performance is used as a baseline for comparisons against the other candidate configurations. Also, since this implementation of the Smith/Waterman hardware core has already been evaluated it is necessary verify the performance and functionality match. Fig-

ure 6.32(a) presents the original, unmodified results that were collected as part of this work. The results closely match those of the previous implementation [102]. From this first figure it can be seen that the database size plays part of the role in the total execution time.

With a baseline available for comparison the candidates can be compared and evaluated for its overall performance improvement. However, before beginning with the candidate configurations that have modified the hardware core, the first configuration looks to apply a simple software patch that was identified as part of the Performance Monitoring Evaluation stage. Namely, that seven software addressable registers within the Smith/Waterman hardware core were unnecessarily being written to due to a missing software guard (`only_once`). With the addition of the guard, Figure 6.32(b) shows an $\approx$2.5$\times$ speedup. In fact, of all of the candidate configurations, this simple software patch provided the greatest performance gain. This software patch will be applied for the remainder of these experiments.

Prior to making any hardware modifications the Systematic Design Analysis also verifies that the inclusion of the performance monitors and necessary infrastructure does not degrade the performance of the running system. Figure 6.32(c) shows the total execution times of the same five databases with the performance monitors in place. When compared to Figure 6.32(b), the results demonstrate how minimally invasive the performance monitoring system is. The augmented Smith/Waterman hardware core is also still able to meet the 100 MHz timing requirement with the addition of the monitors.

In addition to the candidate configuration that was identified by the slave PLB IPIF performance monitor, the master PLB IPIF performance monitor identified a large number of transactions to off-chip memory. The second candidate configuration applies a bus master to direct memory access transform. The resource comparison from such an approach is shown in Table 6.13. The only noticeable difference is the

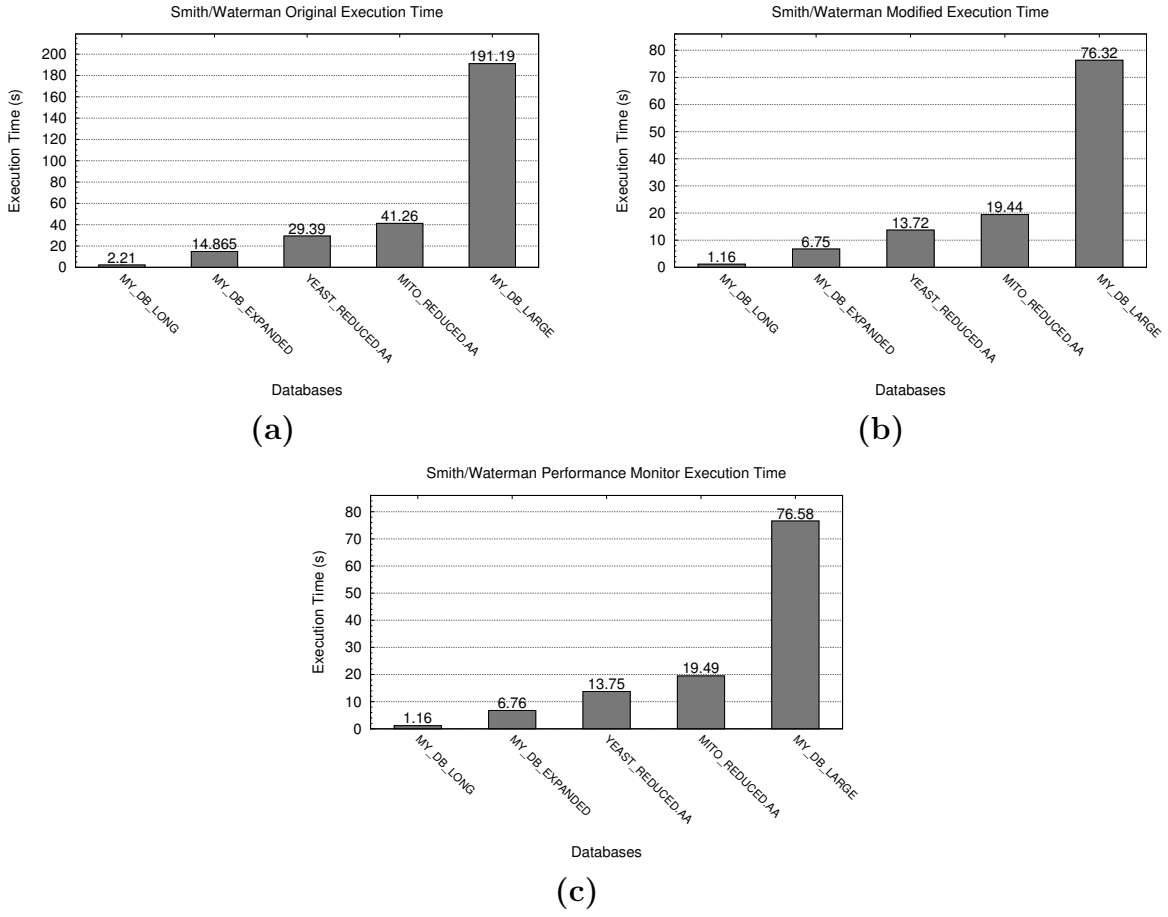


Figure 6.32: execution time of the (a) original Smith/Waterman hardware core (b) modified dropgsw2.c (c) performance monitor cores systems

Table 6.13: resource utilization of Smith/Waterman configuraitons (V4FX60)

Core Configuration	Slice FF (%)	4-LUTs (%)	BRAMs (%)
PLB MST IPIF	1719 (3.40%)	3485 (6.89%)	4 (1.72%)
Bus to DMA	2003 (3.96%)	3263 (6.45%)	8 (3.45%)

double in BRAMs which are necessary for spanning the 100 MHz-to-200 MHz barrier between the hardware core and the memory controller. The resource difference is also possible due to the performance monitor identifying the read-only operations being made by the hardware core.

In terms of performance the DMA implementation is able to obtain a slight increase in performance $\approx$1.22$\times$ increase in performance over the software modified implementation, reducing the largest database execution time from 76 seconds to just over 62 seconds, as seen in Figure 6.33. This improvement in performance comes at minimal resource utilization and requires no modification by the hardware designer. In fact, the results could motivate a designer to re-evaluate the memory interface to determine whether a more optimal solution exists. Certainly, there is some additional latency required to traverse both the Smith/Waterman core's PWAA FIFO and the DMA core's FIFO.

The next set of candidate configurations evaluates the manipulation of the input database's FIFO capacity. Initially the Smith/Waterman core uses a FIFO size of 128 deep elements. As mentioned already, this is under utilizing the BRAM by 4$\times$. The `FIFO Replacement` tool is able to adjust the FIFO size without requiring the designer to make any manipulations to the hardware core. Based on the utilization of the FIFOs, the input database FIFO is dramatically undersized. The first configuration attempts to set the FIFO size to 512 elements (to fully utilize the BRAM). An interesting discovery occurred when running the five databases through the `SSEARCH` application with the larger FIFO. The execution times actually increased, as seen in Figure 6.34(a). This is counter-intuitive as there should be no reason the system

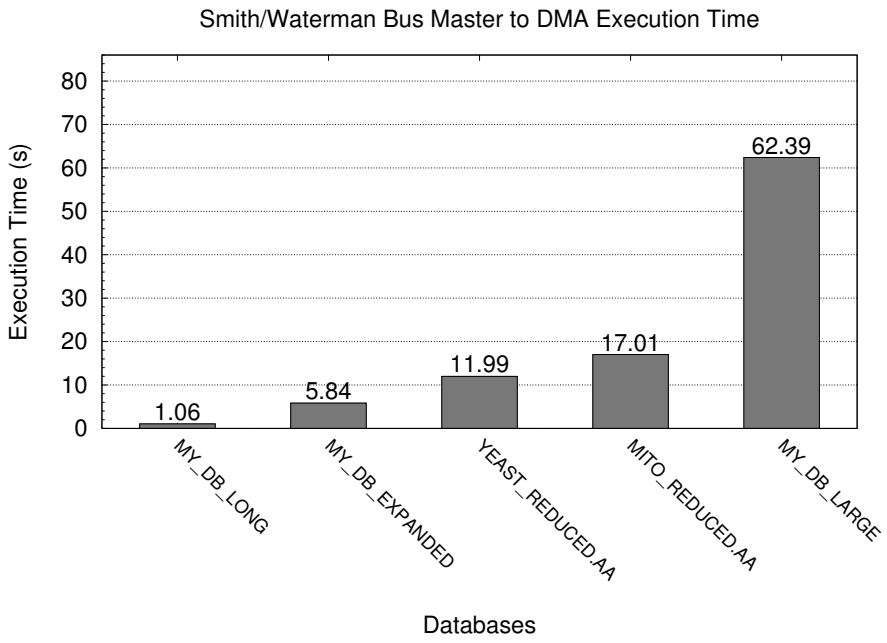


Figure 6.33: Smith/Waterman's DMA interface execution time

would slow down when having larger intermediate buffers.

To verify whether or not the FIFO size was the actual problem the FIFO sizes were increased through the `FIFO Replacement` tool to 2048 and 4096. Figure 6.34(b) and Figure 6.34(c) actually show the same trend as the 512 deep FIFO, with additional slow downs proportional to the size of the FIFO. This raised an alarm that the hardware core itself was not functioning correctly. To evaluate further, the FIFO utilization performance monitor was augmented to support counting the number of reads and write into and out of the database FIFO. Since the entire database should be consumed the number of reads and writes should match. The Systematic Design Analysis flow was able to support this level of debugging due to the modular design of the performance monitoring system. After running several tests on both the original hardware core and the various FIFO sizes it was determined that the database FIFO was not being read to its entirety. At this point debugging is required to identify where in the FSM the Smith/Waterman core is prematurely terminating. A quick analysis of the VHDL and the FSM identified that the `FL_RUN_SETUP` state could

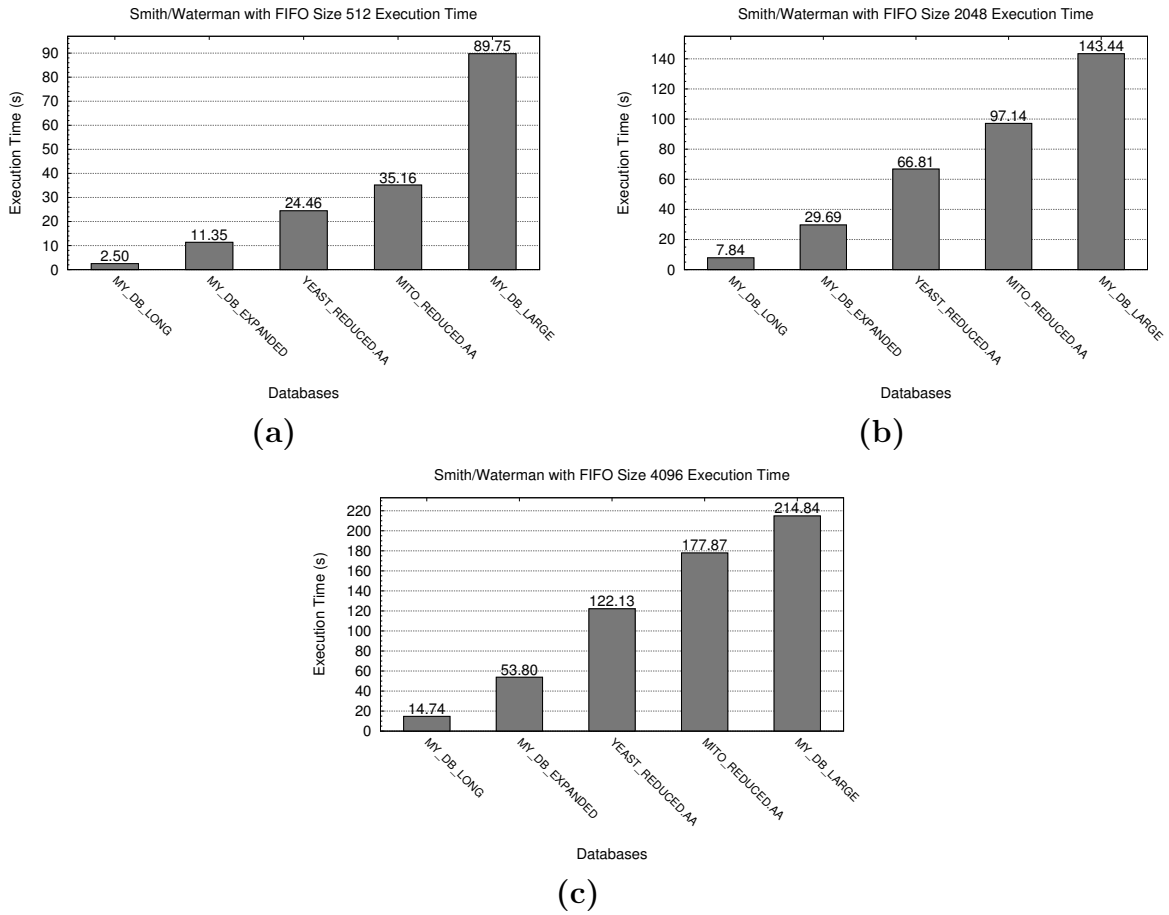


Figure 6.34: execution time of the Smith/Waterman hardware core modified to use (a) 512 (b) 2048 and (c) 4096 deep database input FIFOs

terminate early if the database FIFO is empty. This logic error does not take into account the processor is sending the database to the FIFO and it is possible that the hardware core consumes data out of the FIFO faster than the processor can write it. Instead, there should be an internal counter to identify when the exact number of elements has been consumed. Figure 6.35 shows the state with a comment pointing out the logic error in the VHDL.

6.3.4 Observations and Summary

After identifying the hardware error, the Systematic Design Analysis flow was manually stopped because scaling to different FPGA devices or to additional nodes in the cluster would also contain the design error. Since the other case studies have

```
--State 4: FL_RUN_SETUP
when FL_RUN_SETUP =>
  pwaa_next <= f_str_waa_
s + ((aa1p*n0(23 to 31))*x"4");
  if (aa1_int = x"00000000") then
    -- AGS: ERROR HERE - if fifo_db_
valid = '0'
    if (fifo_db_valid = '1') then
      aa1_int_next  <= fifo_db_dout;
      fifo_db_rd_en <= '1';
      ssj_addr_next <= ssj;
      fsm_ns        <= FL_FETCH_PWAA;
    elsif (aa1p /= x"00") then
      ssj_addr_next <= ssj;
      fsm_ns        <= FL_FETCH_PWAA;
    else
      fifo_score_wr_en   <= '1';
      n0_val_next        <= n0;
      ssj_addr_next      <= ssj;
      ssj_data_next      <= x"00000000";
      bram_write_en_next <= "1";
      -- AGS: ERROR HERE - could terminate early
      fsm_ns             <= FL_RESET;
    end if;
  else
    ssj_addr_next <= ssj;
    fsm_ns        <= FL_FETCH_PWAA;
  end if;
```

Figure 6.35: FSM error identified by performance monitors

already demonstrated the Systematic Design Analysis flow's capability to scale in these ways, it was determined that this case study should focus on the unexpected additional benefits of the flow. Namely, debugging the running system. The amount of time spent to identify where the error existed in the HDL took less than a half hour once it was realized the database FIFO was not being consumed in its entirety. On top of this, the Systematic Design Analysis flow also identified inefficiencies in the software design by identifying seven software addressable registers being over-written with the same value. To identify this problem required a quick search over the C function `FLOCAL_ALIGN()` in the `SSEARCH` application's `dropgsw2.c` file. The missing guard was quickly identified and added and the application was re-compiled. In fact, this patch did not require any hardware modifications and yielded the best individual performance of ≈2.5× speedup. Unfortunately, since the hardware core did not correctly function it is impossible for the Systematic Design Analysis flow to identify the best performing candidate because if the system were fully functional it is quite possible the larger FIFO sizes would yield a more efficient design. Overall the maximum speedup achieved was the combination of the software modification and the DMA interface which yielded a total speedup of ≈3.72× over the original design on a single node with only a modest 0.56% increase in slice flip-flop and 1.73% increase in BRAM utilization. Table 6.14 highlights several additional observations of this case study.

6.4 Case Study: Collatz Conjecture

The Collatz Conjecture states that given a natural number it is possible reduce the number to one by either dividing by two when even or multiplying by three and adding one when odd [104]. For even numbers the resulting calculation reduces the size in half; however, for odd numbers the new number produced is greater than the original number. Thus, it is not obvious that it will terminate to one. For example, given a small number such as $n = 3$, seven iterations are required to reduce n to one.

Table 6.14: summary of Smith/Waterman case study

Stage	Details
Static HDL Profiling	correctly identified both PLB slave and master interfaces found all 15 software addressable registers found all three FIFOs found both FSMs (29 states and four states)
Component Synthesis	identified resource utilization of all components component consumed modest amount of resumes however, scalability study limited by FSM error
Performance Monitors	six monitors inserted verified performance monitors did not impact execution time core with monitors meets timing requirement identify that database FIFO was over utilized easily augmented FIFO monitors to add read/write counters FSM profiler identified `BUILD` as longest running state
Candidate Configurations	nine configurations evaluated modified software implementation evaluated with larger FIFO sizes (512,2048,4096) DMA implementation with with larger FIFO sizes
Performance Evaluation	modified software implementation speedup: $\approx$2.50$\times$ DMA implementation with modified software speedup: $\approx$3.72$\times$ Database FIFOs not consumed due to FSM logic error
Other Observations	debugging capability of SDAflow FSM profiler provide feedback about state execution time DMA interface required no modifications by hardware design

This can be seen in Figure 6.36. Named after Lothar Collatz, the Collatz Conjecture is currently listed as one of the interesting, unsolved problems in number theory [105].

While its practical use has yet to be fully identified, the purpose of its inclusion as a case study within this work is to determine the number of steps for any given unsigned 32-bit integer to be reduced to one. Often, the intermediate values of the calculation are retained to construct a directed graph of the sequence of numbers to

```
n = 3 (odd)
n = 3*3 + 1 = 10 (even)
n = 10/2 = 5 (odd)
n = 5*3 + 1 = 16 (even)
n = 16/2 = 8 (even)
n = 8/2 = 4 (even)
n = 4/2 = 2 (even)
n = 2/2 = 1 *terminating case
```

Figure 6.36: Collatz Conjecture with $n = 3$

```
int collatz(int number, int steps) {
    // report error
    if (number < 1) return -1;
    // terminating base case
    else if (number == 1) return steps;
    // recursive case
    else {
        if ((number % 2) == 0)
            // EVEN: divide by 2 and repeat
            collatz(number/2, steps++);
        else
            // ODD: multiply by 3 add 1 and repeat
            collatz(number*3+1, steps++);
    }
}
```

Figure 6.37: Collatz Conjecture in C

one; however, this work excludes the requirement of such retention. This shifts the focus to a more compute bound problem than a memory bound problem, as storing the numerous results may occupy more time than performing the actual computation.

6.4.1 Design

The design is based on the simple C-code listed in Figure 6.37. In the event that the input number is less than one, the application rejects the input and returns an error. If the input number is one, the application returns the number of steps taken to reach one and terminates. This is to prevent an infinite loop of 1-4-2-1. To identify if the number is even, simple modulo arithmetic is used. Otherwise it is presumed to be odd. The new number is used in place of the original number along with the number of steps completed thus far and the calculation is repeated.

The recursive structure is easily replicated in a hardware description language as a finite state machine, performing the necessary calculations to reduce the initial input number to one. The FSM can be simply represented as shown in Figure 6.38. For the purposes of this design a number is provided by the processor via programmable I/O. Then the FSM calculates the number of steps to reduce the number to one. The FSM then interrupts the processor to indicate the calculation has been completed.

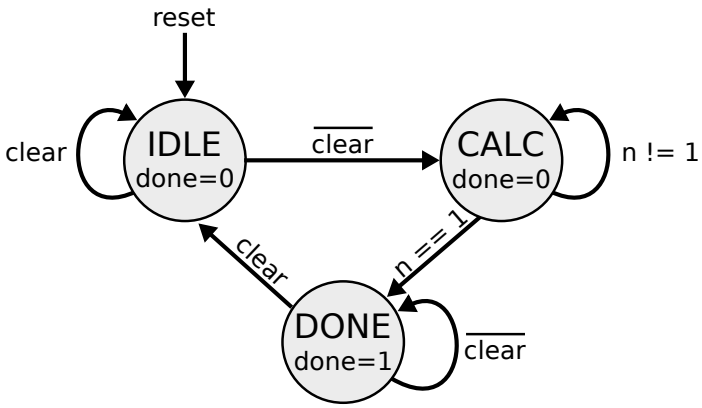


Figure 6.38: Collatz Conjecture FSM

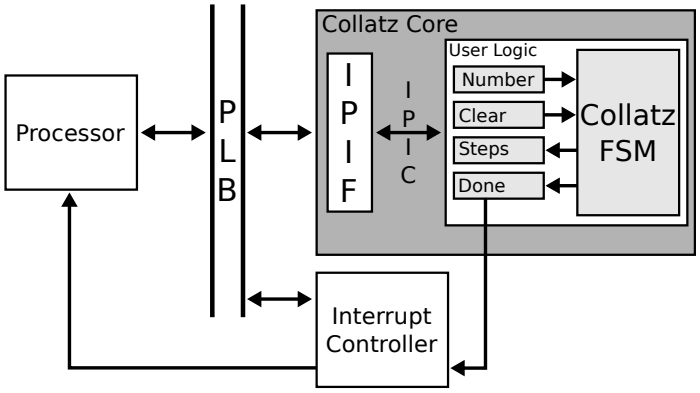


Figure 6.39: Collatz top-level entity block diagram with interface to the PLB through the slave IPIF

Finally, the processor retrieves the number of steps from the hardware core.

The custom compute hardware accelerator core for the Collatz Conjecture is `collatz kernel`. Figure 6.39 depicts the entity which is by default interfaced with the IPIC and encapsulated along with the IPIF in the top-level entity `collatz core`. The `collatz core` connects to the PLB as a simple slave, replying to 32-bit read and write operations to set the initial number input and retrieve the number of steps as the output. A software application running on the processor iterates through 1024 unsigned integers, sending each to the hardware core to be computed and retrieving the number of steps upon completion.

6.4.2 Implementation

The Collatz Conjecture hardware core is ideally suited for scalability as the core computation requires very few resources. Therefore, the Systematic Design Analysis flow can be applied to evaluate the scalability of the design. This subsection highlights the various tools used and configurations generated as part of the Systematic Design Analysis flow. In Section 6.4.3, the results are presented and analyzed.

6.4.2.1 Project Assembly

The first step in the Systematic Design Analysis flow is to take the existing project in the XPS directory structure and to generate the subsystems that will be used throughout the remainder of flow. The first tool used is the `Generate Systems` tool. This tool is described in full details in Chapter 4. The primary functions are to run PlatGen and to construct the `RCS_TOOLS` subdirectory which contains the necessary HDL and synthesis scripts to be used in the Static HDL Profiling and Component Synthesis stages.

6.4.2.2 Static HDL Profiling

The next stage is to parse the system and identify the components and subcomponents of the system. This is accomplished through the `System Parser` tool, the details of which are described in Chapter 4. The Collatz core is identified to contain the following subcomponents: `plb_slave_ipif`, `user_logic`, and `collatz_kernel`. From these components it is identified that the core connects as a slave to the PLB and there is one read-only software addressable register and one write-only software addressable register. A single interrupt has also been identified through the analysis of the MPD file. Finally, a three state FSM is identified within the `collatz_kernel`.

6.4.2.3 Component Synthesis

In the third stage the Collatz core is synthesized to identify the specific resources that are used along with providing the Systematic Design Analysis flow a total resource count for future scalability calculations. This is accomplished through the

Table 6.15: Collatz Core resource utilization (V4FX60)

Component	Slice FF	4-LUTs
Entire Core	233	405
User Logic	32	32
Collatz Kernel	99	354

`Iterative Component Synthesis` tool which walks through and synthesizes a component from the bottom up. While not perfectly accurate as the resources reported are based on pre-placed-and-routed circuits, it does provide a rough guide for the scalability of the system. From Table 6.15 it can be seen that the total size of the entire Collatz core is very small (233 Slice Flip-Flops and 405 4-input Lookup Tables). Therefore, the number of cores that can be replicated will be fairly large. The exact number will be calculated in the Candidate Set Generation and Selection stage.

6.4.2.4 Performance Monitor Insertion

From the static HDL profiling, performance monitors were identified and included within the hardware core. The Systematic Design Analysis flow identifies the following monitors. Presently, the instantiation of the specific monitors is done manually; however, as stated in Chapter 4 the modification to the hardware core are performed automatically. This enables the hardware core to be quickly adapted to include the performance monitors and work with the performance monitor infrastructure.

The first monitor inserted is the PLB slave IPIF which is used to count the number of reads and writes to the software addressable registers within the hardware core. In the Collatz Core there is one write-only register to hold the input number and one read-only register to hold the number of steps.

The second monitor inserted is the interrupt timer to identify the amount of time it takes for the processor to respond to the interrupt, which indicates the Collatz computation is complete. The timer is stopped when the processor retrieves the results from the step register, clearing the interrupt and setting the FSM back to its `IDLE` state.

The third monitor inserted is the utilization timer to calculate the amount of time the hardware core is spent performing useful work. This does not include the time for the processor to respond to the interrupt. For the Collatz Core this time should actually be the same as the number of steps for the number to reduce to one. For other cores this is not necessarily the case.

The fourth monitor inserted is the FSM profiler. The Collatz Core consists of three states: `IDLE`, `CALC`, and `DONE`. The FSM profiler is used to identify the amount of time spent in each of these states. For the purposes of this work the time spent in `CALC` should equal the number of steps and the `DONE` state should equal the time for the processor to respond to the interrupt.

Finally, a histogram performance monitor is inserted for demonstration purposes to capture the number of times each step value is repeated within the range of input numbers from 2 to 1024. This specific monitor demonstrates that additional functionality can be quickly added to an existing hardware core that would otherwise require the designer to manually create and insert it. While no feedback data is provided about the performance of the core, it does neatly present the Collatz results for the small range of evaluated numbers.

6.4.2.5 Performance Monitor Evaluation

The performance monitor data is collected with the assistance of the Performance Monitor Infrastructure which is fully described in Chapter 4. The head node retrieves the runtime data upon completion of the experiment and the results are stored in a file to be parsed during the next stage in the Systematic Design Analysis flow.

The first performance monitor captures the number of reads and writes to the software addressable registers in the Collatz Core. Since only one write-only and one read-only register were found, the monitor consists of two counters. The test application performed the Collatz calculation on the numbers from 2–1023, inclusive, which is a total of 1022 individual read and write operations. In total, 1022 writes

were monitored and 1022 reads were monitored, as expected.

The second performance monitor captures the time for the processor to respond to each individual interrupt. The monitor is a timer which is enabled when the interrupt is asserted and disabled when the processor performs the read request from the steps register. This time was then averaged over the 1022 runs. The average result is 1119.67 clock cycles (100 MHz), meaning that the processor takes $\approx$11.12 μs to respond to an interrupt, often longer than the time for the Collatz core to perform the entire computation.

The third performance monitor captures the time the Collatz core is being utilized (not in the `IDLE` or `DONE` states). Since for each input number the Collatz computation can differ in the number of steps to reach one, the monitor was averaged over the number of runs. The total time utilized is 61,307 clock cycles (100 MHz) or 613.07 μs with an average utilization time of 59.99 μs.

The fourth performance monitor profiles the FSM in the Collatz core and identified that for the 1022 input values the total percentage of time spent in the `IDLE` state was 2.07%, `CALC` was 4.98%, and `DONE` was 92.95%. The FSM profiler quickly identified the `DONE` state needing improvement.

6.4.2.6 Candidate Set Generation, Selection, and Evaluation

From the performance monitors and the static hardware profiling the following set of candidates were identified, generated, and evaluated.

The first candidate identified scales the number of unique Collatz cores connected to the PLB on a single node. The specific tool used was the `PLB Replicate PCORE` tool, which is detailed in Chapter 4. Upon initial analysis performed automatically as part of the PLB Replicate PCORE tool and run by the static hardware profiler the Collatz core should be able to be replicated at most 118 times on the Xilinx ML410 development board's V4FX60 FPGA. This does not take into account the limitation set by the Xilinx implementation of the PLB which limits the number of bus

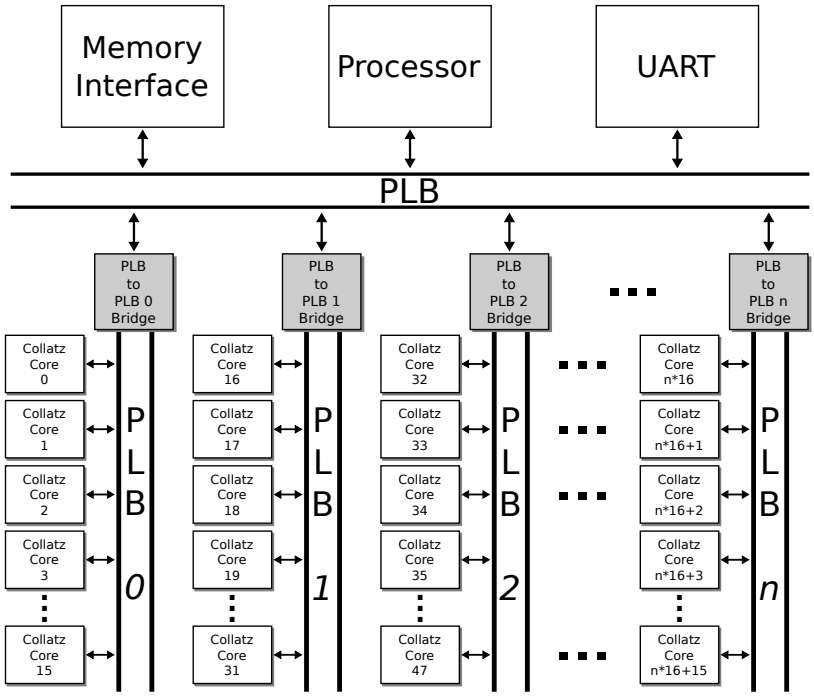


Figure 6.40: Collatz scaled across PLBs and bridges

connections to 16 slaves. When factoring the additional resources for the necessary bridges and buses to scale the design beyond 16 cores, the maximum number of cores drops to 102. The purpose of the PLB Replicate PCORE tool is to generate candidates of sizes up to the maximum number calculated, here set to 102. Figure 6.40 illustrates the scalable nature of the Collatz core across multiple buses and bridges. Each system is automatically generated and evaluated separately. For the purposes of this work (and to reduce the number of candidates) the following number of cores were evaluated: 8, 16, 32, 64, 96, and 102. The results of these candidate configurations are reported in Section 6.4.3.

The second candidate identified the PLB for replacement with a full crossbar switch. This is accomplished through the use of the `PLB to Crossbar Switch` tool, which is detailed in Chapter 4. During the static hardware profiling stage the PLB is identified as connecting the hardware core through a slave Bus interface. The PLB is then replaced with a crossbar switch which has a higher overall throughput than a

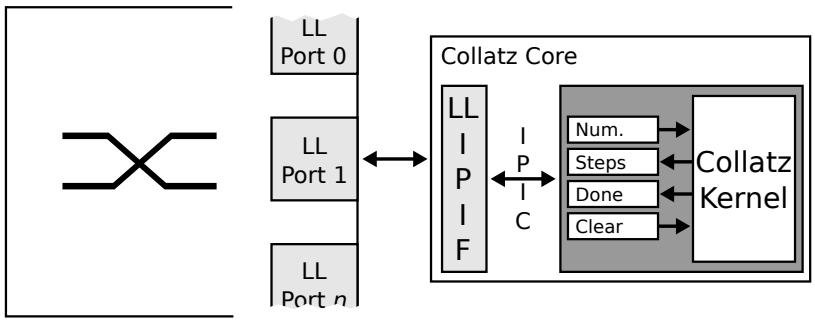


Figure 6.41: Collatz core with interface to the crossbar switch

traditional bus interconnect. The Collatz core is modified by removing the PLB IPIF and replaced with a custom developed LocalLink Intellectual Property Interface (LL IPIF) as is shown in Figure 6.41.

The third candidate extends the second candidate by replicating the number of Collatz cores connected to the crossbar switch. Using synthesis results of the single core implementation it was estimated that a maximum of 50 Collatz cores could be connected through the crossbar switch. Similar to the PLB system replication, the `Crossbar Switch Replication PCORE` tool is used; however, there is no limitation on the number of cores the switch can connect. Figure 6.42 shows the configurable nature of the crossbar switch connecting multiple Collatz cores. This number is approximately half the number of the bus implementation; however, the purpose of the Systematic Design Analysis flow is to identify configurations that may trade performance in place of scalability. For the purposes of this configuration set the number of cores evaluated is 8, 16, 32, and 50. The results of these candidate configurations are reported in Section 6.4.3.

The fourth candidate evaluates the scalability of the design to a larger FPGA, specifically the V5FX130T found on the Xilinx ML510 development board. Here the nearly double in capacity provides an interesting test bed for continued scalability and tries to address the question of how designers can productively scale a design to a larger chip. The `Migrate to ML510` tool is used to take an existing system and move

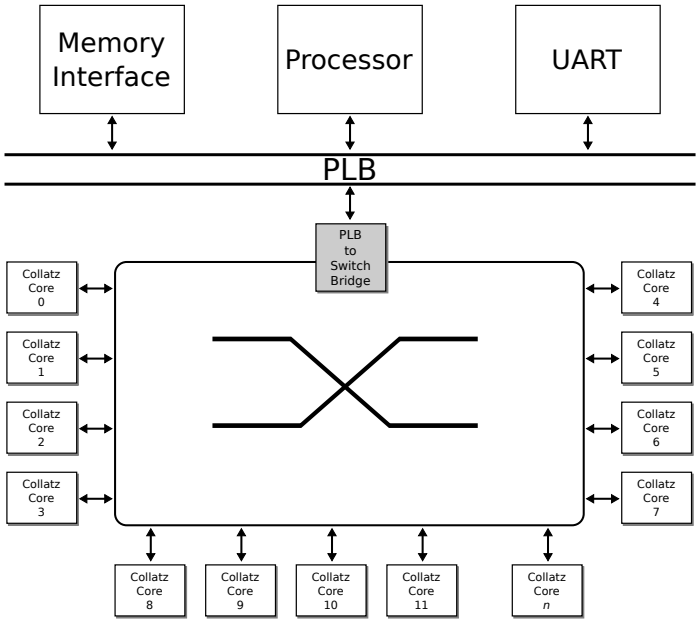


Figure 6.42: Collatz core scaled across crossbar switch

to the larger system. In this specific implementation only the custom hardware core is migrated since the remainder of the infrastructure (processors, memory interfaces, UARTs, etc) differ between generations.

The fifth candidate evaluates the scaling the fourth candidate configuration up to the larger number of hardware cores since the Xilinx ML510's V5FX130T FPGA has an 2× the number of slice and lookup table resources. Using the `PLB Replicate PCORE` tool the design is replicated to evaluate up to 170 hardware cores in increasing intervals: 8, 16, 32, 64, 128, 170.

The sixth and final candidate evaluates the scalability of the design to another FPGA, specifically the V5LX110 found on the Xilinx Virtex 5 XUP development board. Unlike the ML510's FX130T device, this V5LX110 does not include any integrated processors such as the PowerPC 440. Instead, the design must be migrated to work with a soft processor. Specifically, this work will use the Xilinx MicroBlaze[84]. With slightly less than double the resources over the Virtex 4 FX60 this candidate configuration explores the scalability over different processors. The `Migrate to XUPV5` tool is used to take an existing system and move to the XUP system.

6.4.3 Results and Analysis

The final stage in the Systematic Design Analysis flow is the evaluation of each of the candidate configurations generated in Section 6.4.2. For the Collatz Conjecture core the candidates span a single Xilinx ML410 or Xilinx ML510 development board.

The first candidate configuration set scales the number of Collatz cores running in parallel across the PLB. The limitation of the PLB to support at most 16 connect cores requires that the primary bus in the system be connected with multiple bridges to secondary buses. Each bridge acts as a slave on the primary bus and a master on its own respective secondary bus. This is best depicted in Figure 6.40. From the Systematic Design Analysis a maximum of 102 Collatz cores are estimated to be synthesized for the Xilinx ML410 development board. To support 102 cores the system must also include seven secondary buses and bridges. While 102 cores is considered the upper bound the Systematic Design Analysis flow considers a range of the number of Collatz cores to be 1, 8, 16, 32, 96, and 102. The `Resource Utilization Evaluation` tool is used to plot the resource utilization of the Collatz cores, PLBs, and PLB to PLB bridges individually as well as the system as a whole.

Figure 6.43(a) shows the linear scalability of the Collatz core as the number of cores in the system scales to 102 cores. At its peak the Collatz cores consume over 81% of the 4-input Lookup Tables (LUTs) and 47% of the Slice flip-flops of the available resources with 102 cores. The resources required by the PLBs is presented in Figure 6.43(b). The PLB requires a smaller amount of resources (139 FFs and 344 LUTs per bus) since each Collatz core is only a slave and the PLB to PLB bridge is the only master on each of the secondary PLBs. The bridges do consume more resources as they must span both buses, as can be seen in Figure 6.43(c). Finally, the total amount of resources needed by the system is shown in Figure 6.43(d).

In addition to resource utilization the candidate configurations were also evaluated using a standalone C application designed to operate with an increasing number of

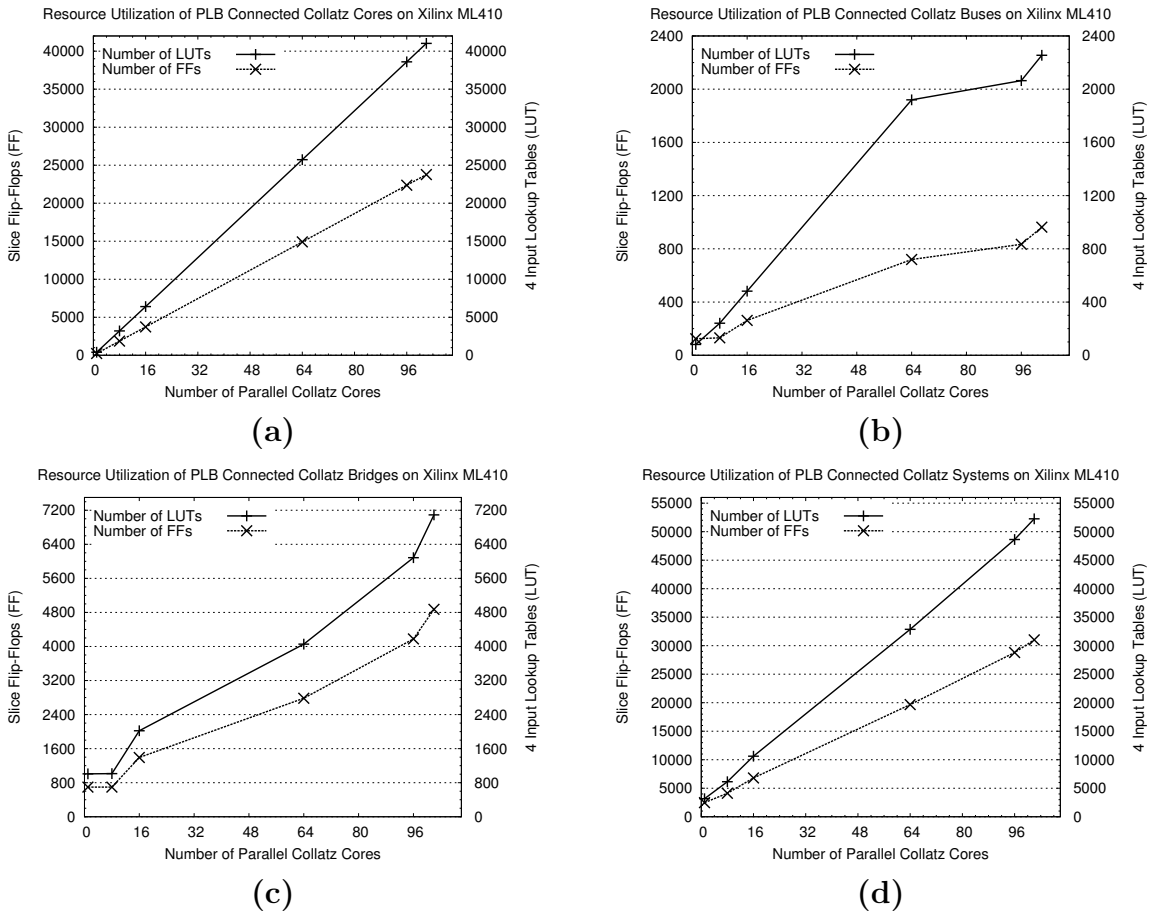


Figure 6.43: resource utilization of PLB base system's (a) Collatz cores (b) PLBs (c) bridges (d) entire system on the Xilinx ML410 development board

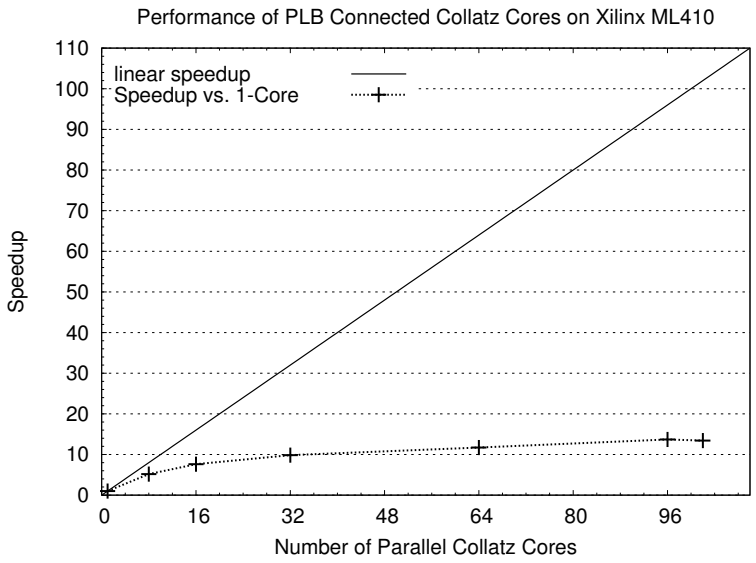


Figure 6.44: speedup over a one Collatz core implementation in the PLB base system on the Xilinx ML410 development board

parallel Collatz cores. The C application is not fully optimized to provide DMA transaction; therefore, programmable I/O is used by the processor to read and write to each of the Collatz core's software addressable registers. As was identified by the Interrupt Timer performance monitor, a large portion of the time is spent servicing the interrupts. As a result the overall speedup, shown Figure 6.44, shows a leveling off at 96 parallel Collatz cores and even though the system with 102 Collatz cores can fit in the available resources, the performance degrades to the point of rejecting that candidate configuration. Furthermore, the results shows that the speedup does not match linear speedup which strongly indicates that the original Collatz hardware core designed for the PLB on the Xilinx ML410 is not optimal.

Finally, the `PLB Replicate PCORE` tool can be adjusted to try varying configurations of the number of Collatz cores per bus. This allows a design trade off between a fully utilized bus to be compared against two half utilized buses. For demonstration purposes two systems were constructed and compared. The first system is comprised of one secondary bus with sixteen cores. The second system is comprised of two secondary buses each with eight cores. Both systems therefore have the same number

Table 6.16: PLB bus and bridge resource utilization comparsion

System	Collatz Cores/Bus	Bus(es) Slice FF	Bus(es) 4-LUTs	Bridges(s) Slice FF	Bridges(s) 4-LUTs
1 Bus	16	139	344	696	1014
2 Buses	8	262	482	1392	2020

of parallel running Collatz Cores. From a resource utilization perspective the single bus system uses approximately half the number of flip-flops and lookup tables due to the additional bridge and bus, as can be seen in Table 6.16. Clearly, the system with a single bus uses fewer resources. The largest contributor to the resource utilization is the additional bridge to connect the second bus. Besides resources it is important to compare performance as well. Using the same test infrastructure the two systems both executed the application in 1.683 ms. The result of this additional configuration confirms that for the Collatz core fully utilizing the bus in terms of the number of connections does not saturate the bus or diminish performance.

The second and third configuration sets take the PLB implementation and replaces it with a full crossbar switch. This configuration requires no modification by the designer and can be generated to scale up to 50 parallel Collatz cores. The difference in the PLB and crossbar switch configurations limits the number of Collatz cores; however, it could be worth the trade off if the performance of the crossbar switch implementation out performs the performance of the PLB implementation.

Ultimately, the system was unable to scale to 50 parallel cores due to timing issues with the crossbar switch implementation. Therefore, the results presented consider designs scaling from a single core up through sixteen parallel Collatz cores. Figure 6.45(a) shows the resource utilization of the Collatz core with the new LL IPIF in place of the PLB IPIF. The switch requires fewer flip-flops (183 vs. 233); however, due to the connectivity to the crossbar switch additional lookup tables are required (513 vs. 400). In Figure 6.45(b) the resource utilization of just the crossbar switch is presented, showing that the number of ports on the switch, also known as the radix

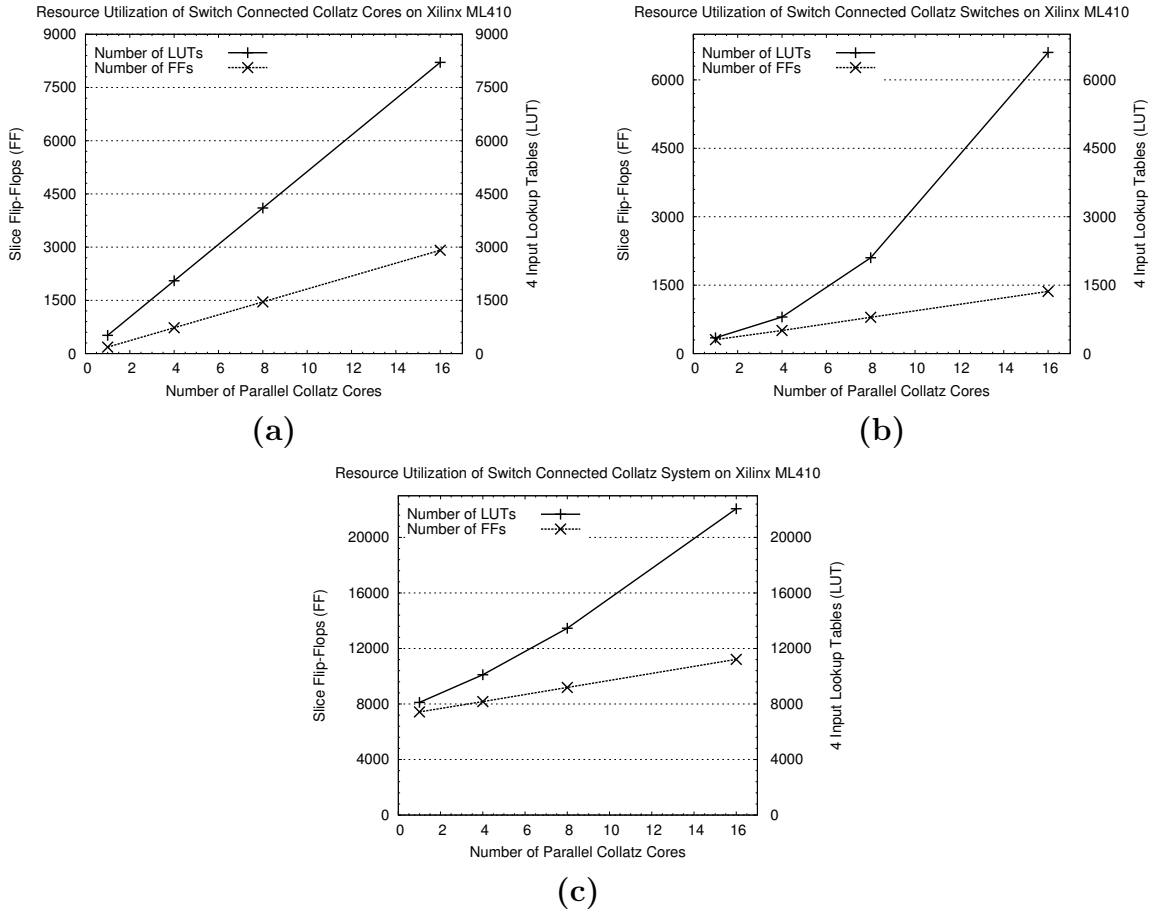


Figure 6.45: resource utilization of crossbar switch's (a) Collatz cores (b) crossbar switches and (c) entire system on the Xilinx ML410 development board

of the switch, greatly indicates the resource consumption. The trade off between a bus system requiring additional bridges and a crossbar switch cannot be argued in this case due to the significant resource requirements for each additional port. Future designs which have a higher degree of parallel communication may warrant such a trade off. Finally, Figure 6.45(c) shows the resource utilization of the entire system.

In comparing the performance of the PLB system to the crossbar switch system, a single Collatz core executes in 13.29 ms, which is ≈1 ms longer than the PLB system. This is primarily due to the additional latency though the crossbar switch interconnects which in this current form are implemented as FIFOs. Each FIFO has an additional 6 clock cycle (100 MHz) latency which is what accounts for the overall

slower performance. Therefore, the crossbar switch candidate configuration of the Collatz core can be rejected due to its comparison with the PLB based system.

The fourth and fifth candidate configuration set considers the core's scalability on a larger device, namely the Xilinx ML510 development board's V5FX130T FPGA. Considering the PLB replicated design for the ML410, a comparable ML510 design was constructed using the `Migrate to ML510` tool. With $\approx 2\times$ the available resources, an estimated 200 parallel Collatz cores are attempted to be implemented in this candidate configuration set. Specifically, the fourth configuration consists of the scalability to a single core on the large Virtex 5 FPGA. The fifth configuration set consists of 8, 16, 32, 64, 128, and 200 cores. Upon initial analysis; however, the 200 core implementation was unable to meet timing and there has been dropped from the set. Instead 170 Collatz cores has been identified as the upper bound to meet timing and therefore is included in place of the 200 cores.

As was shown for the ML410 development board the following figures illustrate the individual resource utilization of the Collatz cores, buses, bridges and the entire system as the cores are scaled from one core to 170 parallel cores. Figure 6.46(a) shows the linear scalability of the Collatz core as the number of cores in the system scales to 170 cores. The Collatz core implemented on the Virtex 5 FPGA uses the same number of slice flip-flops; however, with the use of the 6-input lookup tables the number of LUTs used drops to 321 (compared to 400 for the Virtex 4 implementation). This savings required no modifications to the original hardware core, it is a result of the silicon improvement with 6-input lookup tables. The resources required by the PLBs is presented in Figure 6.46(b). As with the ML410 the bridges continue to consume more resources as they must span to buses, as can be seen in Figure 6.46(c). Finally, the total amount of resources needed by the system is shown in Figure 6.46(d).

Again, in addition to resource utilization the candidate configurations were also evaluated using a standalone C application designed to operate with an increasing

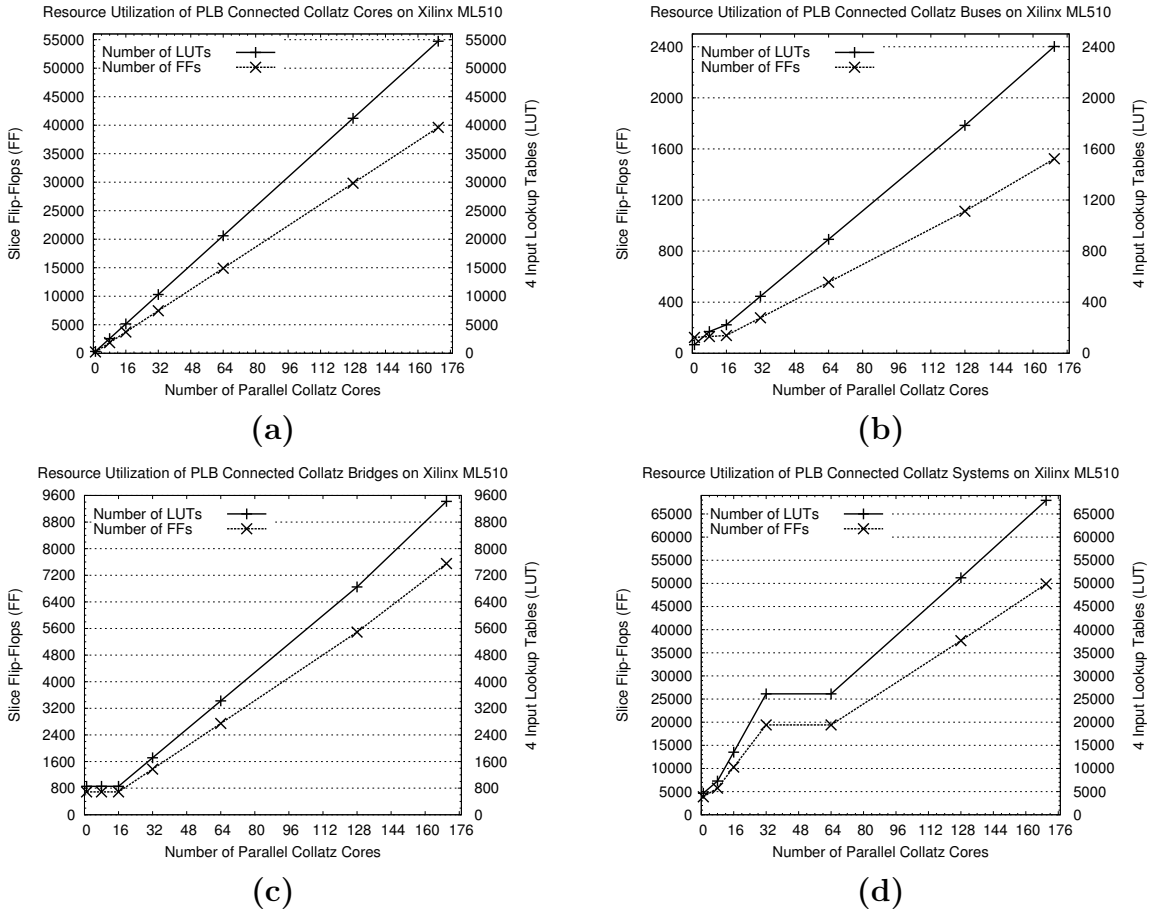


Figure 6.46: resource utilization of PLB base system's (a) Collatz cores (b) PLBs (c) bridges (d) entire system on the Xilinx ML510 development board

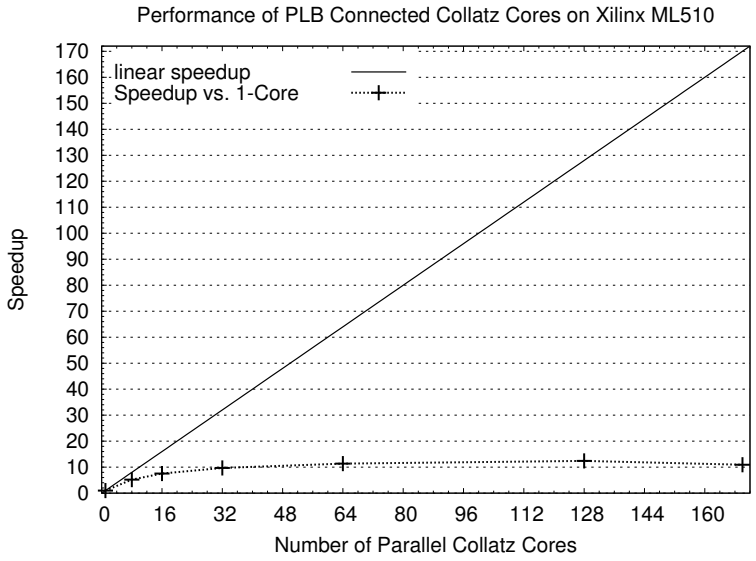


Figure 6.47: speedup of PLB base system on the Xilinx ML510 development board

number of parallel Collatz cores. The same standalone C application is used for the ML510 design as was used for the ML410 design. The ML510 design's PPC440 processor operates at 400 MHz compared to ML410's PPC405 processor operating at 300 MHz. This leads to a reduction in the time to respond to an interrupt as the number of interrupts increase. As can be seen in Figure 6.47 the performance scales to 128 parallel Collatz cores and then performance drops to 170 cores. The 128 Collatz system running on the Xilinx ML510 offers the best performance of the ML510 PLB based systems.

The sixth and final candidate configuration set considers the core's scalability with a different processor type, namely the Xilinx MicroBlaze on the XUP development board's V5FX130T FPGA. Considering the PLB replicated design for the ML410, a comparable XUP design was constructed using the `Migrate to XUP` tool. With slightly less than $\approx 2\times$ the available resources, an estimated 176 parallel Collatz cores are attempted to be implemented in this candidate configuration set. Specifically, the set consists of 1, 8, 16, 32, 64, 128, and 176 cores. Upon initial analysis; however, the 176 core implementation did not fit in the design and there has been dropped from

the set. Instead 150 Collatz cores has been identified as the upper bound to meet timing and therefore is included in place of the 176 cores.

As was shown for both the ML410 and ML510 development board the following figures illustrate the individual resource utilization of the Collatz cores, buses, bridges and the entire system as the cores are scaled from one core to 150 parallel cores. Figure 6.48(a) shows the linear scalability of the Collatz core as the number of cores in the system scales to 150 cores. The Collatz core implemented on the Virtex 5 FPGA for the XUPV5 and ML510 use the same number of slice flip-flops and LUTs. The resources required by the PLBs is presented in Figure 6.48(b). As with the other designs the bridges continue to consume more resources as they must span to buses, as can be seen in Figure 6.48(c). Finally, the total amount of resources needed by the system is shown in Figure 6.48(d).

6.4.4 Observations and Summary

The Collatz Conjecture application case study has clearly demonstrated the capabilities of the Systematic Design Analysis. This specific case study has been used to highlight the scalability and migration features which can help a designer quickly develop a core, integrate it into a system, scale it to utilize the available resources, and even migrate the design to other FPGA devices.

Overall, the Systematic Design Analysis flow offered 24 candidate configurations without requiring any modifications to the original Collatz hardware core. Furthermore, the flow used the performance monitoring data from a single core to identify some of the key bottlenecks in the current implementation. Specifically, looking at the performance monitoring data the time to respond to the interrupt ($\approx$11.12 μs) is identified as consuming $\approx$93% of the total execution time for the Collatz core. Figure 6.49 illustrates this result which was generated from the FSM profiler performance monitor. The performance monitors also offered additional functionality by creating a histogram of the steps needed by each number to reduce to one. The monitor was

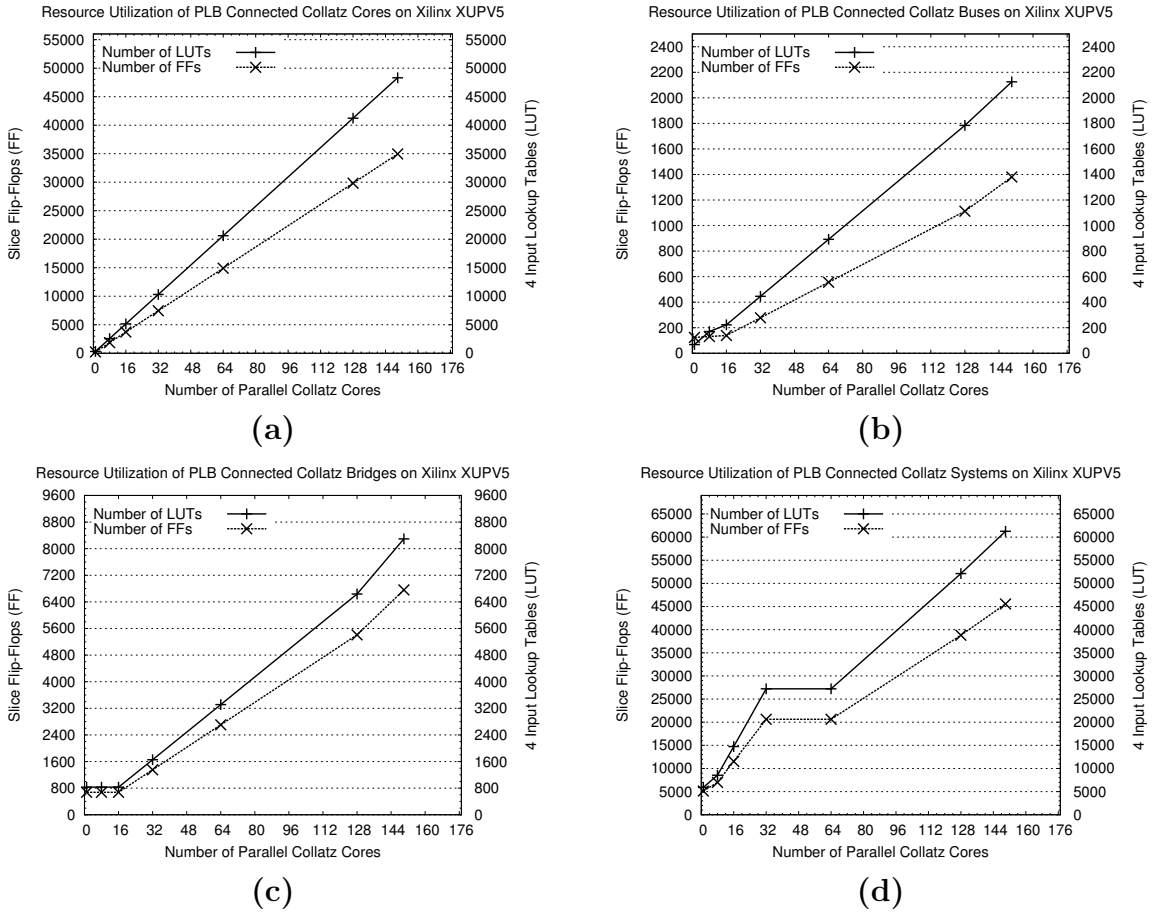


Figure 6.48: resource utilization of PLB base system's (a) Collatz cores (b) PLBs (c) bridges (d) entire system on the Xilinx XUPV5 development board

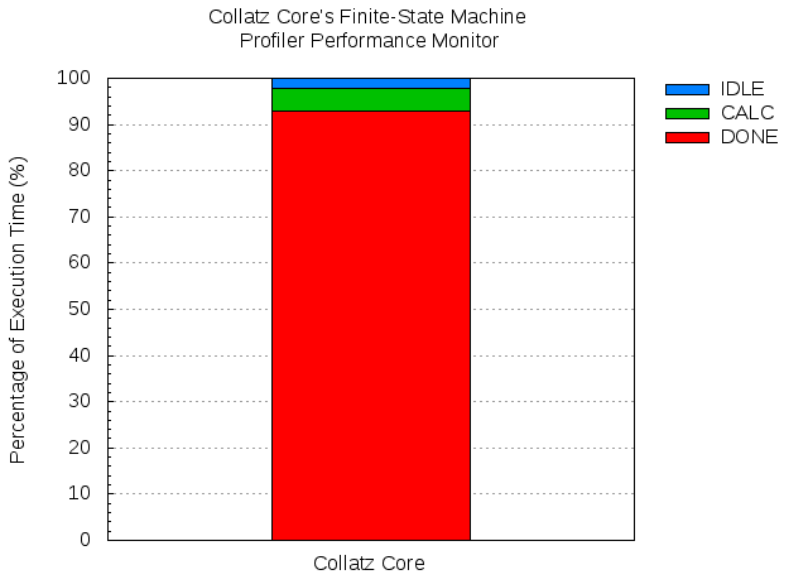


Figure 6.49: Collatz core's FSM profiler monitor results

added with minimal effort and collect along with the other performance monitoring data, requiring no effort by the processor running on the FPGA. Figure 6.50 depicts this histogram data. Finally, after analyzing all 24 configurations, the 128 Collatz system on the ML510 performs the best (lowest total execution time) and is selected as the best candidate of all of the aforementioned candidate configuration sets with an overall speedup of 16.74× over a single core on the ML410. In addition to the performance gains, Table 6.17 highlights several additional observations of this case study.

6.5 Systematic Design Analysis Flow Evaluation

In addition to the case studies which cover specific applications, it is necessary to also evaluate the functionality of the stages of the Systematic Design Analysis flow individually. While the case studies look at how the flow and supporting tools function as a cohesive unit, this section validates that each stage also functions correctly. Therefore, the following set of tables (Tables 6.18 to 6.27) are presented. To summarize these tables, and to provide a lead into the full evaluation of the Systematic Design Analysis based on the case studies, each stage is able to perform the specified

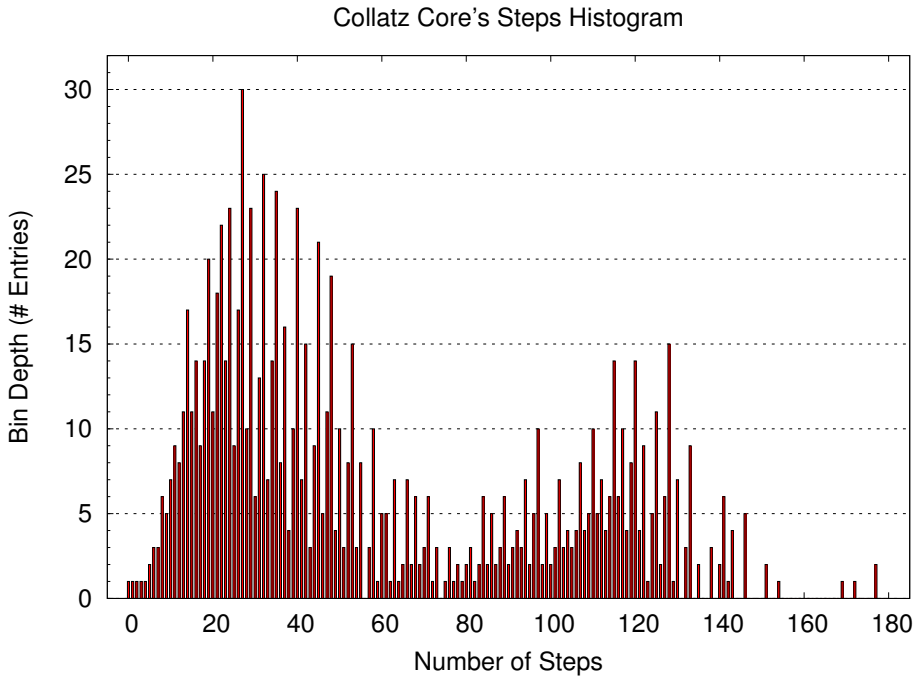


Figure 6.50: Collatz core's histogram of steps

Table 6.17: summary of Collatz Conjecture case study

Stage	Details
Static HDL Profiling	correctly identified PLB slave interface found all two software addressable registers found one FSM with three states
Component Synthesis	identified resource utilization of all components initial scalability limited by PLB (16 cores/bus maximum)
Performance Monitors	five monitors inserted processor responding to interrupt identified as bottleneck
Candidate Configurations	24 configurations evaluated migrated PLB design to crossbar switch migrated PLB design to ML510 migrated PLB design to XUPV5 scaled design to multiple buses and bridges on ML410 scaled design to multiple buses and bridges on ML510 scaled design to multiple buses and bridges on XUPV5 scaled design to multiple ports on crossbar switch on ML410
Performance Evaluation	achieved 13.70× speedup on 96 cores on ML410 PLB system achieved 16.74× speedup on 128 cores on ML510 PLB system ML410 crossbar switch high FIFO latency 92.95% of execution waiting for processor to read results
Other Observations	monitors can help identify software bottlenecks migrating between processor system possible without designer supplemental data collection post hardware design

functions. Initially, these functions were to aid with the development and to validate the stages behaved correctly. It is also included here to re-iterate the function of each stage. In addition, experiences regarding the specific stage will also be discussed.

Table 6.18 shows the functional performance of the Project Assembly stage. All of the core functionality is working and does so for both Virtex 4 and Virtex 5 base systems. An observation of this stage is that as the Xilinx tools continue to be developed, this tool may require additional modifications. Although, through the Xilinx 11.x tool chain, the Generate Systems tool has been shown to function correctly. Table 6.19 shows the functional performance of the Component Synthesis stage. Of all of the stages, this is probably the easiest stage for an existing hardware designer to comprehend since it runs XST across all of the components. What is unique about this stage, is how it is integrated with the Generate Systems tool so that the bulk of the work a designer would typically have to do in order to synthesize the low-level components of a system is automated. In Table 6.20 the functional performance of the Single Node Performance Evaluation stage is listed. This stage presents a significant hurdle for the Systematic Design Analysis flow. Each of the applications evaluated have been tested in a slightly different way. For example, Collatz uses standalone C with interrupts and a scaled address map to set and retrieve its test data; whereas, BLAST uses a fully automated Python testing infrastructure to do everything from booting the system to storing the results in custom Python data structures. As a result, no singular application testing framework could be created. As a result, this stage relies heavily on the designer to provide a working test application and to identify how to evaluate its performance.

Table 6.21 shows the functional performance of the Static HDL Profiling stage. This is one of the most useful stages of the Systematic Design Analysis flow because it aggregates all of the component's parsing data to identify different interfaces and specific components used in the system. This stage also combines synthesis results

Table 6.18: Project Assembly stage

Working?	Function
√	input source HDL for the design
√	input design constraints file
√	parse design constraints file
√	identify FPGA board types in system
√	user specified synthesis parameters
√	create project for component synthesis stage
√	create top-level project for synthesis tool
√	create sub-projects for each sub-component
√	create project for static HDL profiling stage

Table 6.19: Component Synthesis stage

Working?	Function
√	synthesize project created in project assembly stage
√	synthesize sub-projects
√	parse synthesis reports for all projects
√	generate resource utilization data structures
√	pass data structures to static HDL profiling stage
√	pass data structures to insert performance monitor stage

Table 6.20: Single Node Performance Evaluation stage

Working?	Function
√	run design with test application
√	store results in data structure
√	pass data structure to monitor single node performance stage
√	pass data structure to performance analysis stage

Table 6.21: Static HDL Profiling stage

Working?	Function
√	parse HDL for port map signals
√	parse HDL for internal signals
√	parse HDL for internal components
√	parse HDL for finite-state machines
√	evaluate signals for interfaces
√	evaluate resource utilization from synthesized project
√	pass data structures to insert performance monitors stage

Table 6.22: Insertion of Performance Monitors stage

Working?	Function
√	parse data structures
√	recommend performance monitors for insertion
√	insert performance monitoring infrastructure

with parsing information to better understand black boxes, FSMs, flip-flip/register utilization, among other basic elements. In the end, this information is passed along to the next stages.

The Insertion of Performance Monitors stage is the first stage to analyze the results to recommend performance monitors to the designer and also to insert the performance monitoring infrastructure. Table 6.22 shows the functional performance of this stage. To address the question as to why this stage does not also insert the monitors is because the designer may still choose to insert custom monitors. However, based on the four case studies, it could be possible to insert these monitors automatically, with a future tool. Once inserted the monitoring is simply a matter of re-running the original tests and then using the supplied tools to collect the performance monitor data. Table 6.23 shows the functional performance of this state. The performance monitoring infrastructure is minimally invasive and does not require any attention from the design under test.

Table 6.24 shows the functional performance of the most important state of the Systematic Design Analysis flow, Candidate Configuration Generation. Several candidate configurations have been generated and selected for evaluation as part of the

Table 6.23: Monitor Single Node Performance stage

Working?	Function
√	run design with test application
√	store results in data structure
√	retrieve monitor cores results
√	parse monitor cores results
√	pass results to next stage
√	verify results match single node performance evaluation

Table 6.24: Candidate Set Generation and Selection stage

Working?	Function
√	parse static HDL performance data structure
√	parse performance monitor results
√	determine possible memory configurations
√	determine possible network configurations
√	determine possible on-chip interconnect configurations
√	generate synthesis projects for each candidate configuration

four case studies. While the process is still based on recommendations from the tool flow, it does also strongly indicate these candidates could be more automatically generated. However, since the evaluation and comparisons is manual, this stage also being manually implemented does not detract from the capabilities of the flow. Table 6.25 shows the functional performance of the Cluster Synthesis stage, which just shows the candidate configurations generated valid designs that could be synthesized. The last two tables, Table 6.26 and Table 6.27, shows the functional performance of the Cluster Evaluation and Performance Analysis stages. While these stages are not automated, the functionality is validated with the four case studies.

6.6 Functional Analysis

This work sets out to address the key question, *can the knowledge of an experienced hardware designer be codified into a set of principles, guidelines, and tools* such that it

Table 6.25: Cluster Synthesis stage

Working?	Function
√	synthesize candidate for cluster of resources
√	pass bitstream to cluster performance evaluation stage

Table 6.26: Cluster Performance Evaluation stage

Working?	Function
√	parse input test vectors and results files
√	distribute configurations to each of the nodes in the test
√	execute the input test vectors
√	record the results
√	compare results to verify functional system
√	pass performance results to next stage

Table 6.27: Performance Analysis stage

Working?	Function
√	parse results from single node performance evaluation
√	parse results from cluster performance evaluation
√	compare performance results and generate results file

can be used to make designers more productive? In Chapter 4 the Systematic Design Analysis flow and supporting tools are presented which includes the use of static hardware profiling, timing and resource profiling along with runtime performance monitoring to create a model of performance and a set of tools for spatial scaling. Then, using the testing infrastructure listed in Chapter 5 four case studies have been used to evaluate the approach. More specifically each application is evaluated by addressing the following set of questions:

1. Can static hardware profiling data be collected for each application?
2. Can the profiling information be used to identify interfaces such as memory, the network, and other cores?
3. Can the profiling information be used to determine potential bottlenecks associated with scaling the design?
4. Can we use the profiling information to generate a list of signals or cores to monitor in a running system.
5. Does adding performance monitors aid in the designer's ability to scale the system?
6. Can we retrieve the performance monitor's information without any overhead

to the system?

7. Once the data in collected can it provide useful feedback to the designer or some tool flow that can be used to re-configure the design?
8. Can we gather useful information from existing designs to help predict performance scalability for future designs?
9. Can this information be presented to the designer such that future design efforts are more productive?
10. As the amount of resources grow, can the number of configuration choices options be minimized?

Finally, the testing methodology will be to make a qualitative judgment for each of these questions with each of the applications. This will enables a calculated summary of statistics for all of the questions which is then plotted using a heat map. If the results show an overwhelmingly positive response to these questions, then we feel we have strongly affirmed the thesis question. On the other hand, if the results are overwhelmingly negatively, then we have strongly refuted the approach to answering the thesis question.

Figure 6.51 presents the results of the aforementioned qualitative analysis over the four case studies evaluated using the Systematic Design Analysis flow. Overall, the results indicate highly favorable results, strongly affirming the thesis question. In fact, out of the forty questions, only six questions were answered negatively. This results in an 85% positive review of the Systematic Design Analysis flow. A brief breakdown of the review for each question are listed in the following four subsections.

6.6.1 Matrix-Matrix Multiplication

The Systematic Design Analysis flow has been successfully demonstrated for matrix-matrix multiplication. In response to the ten questions listed to evaluate the flow, all but Question 10 are positive. The tools and flow has not been optimized to minimize the number of configurations. The Candidate Configuration Recommendation tool is

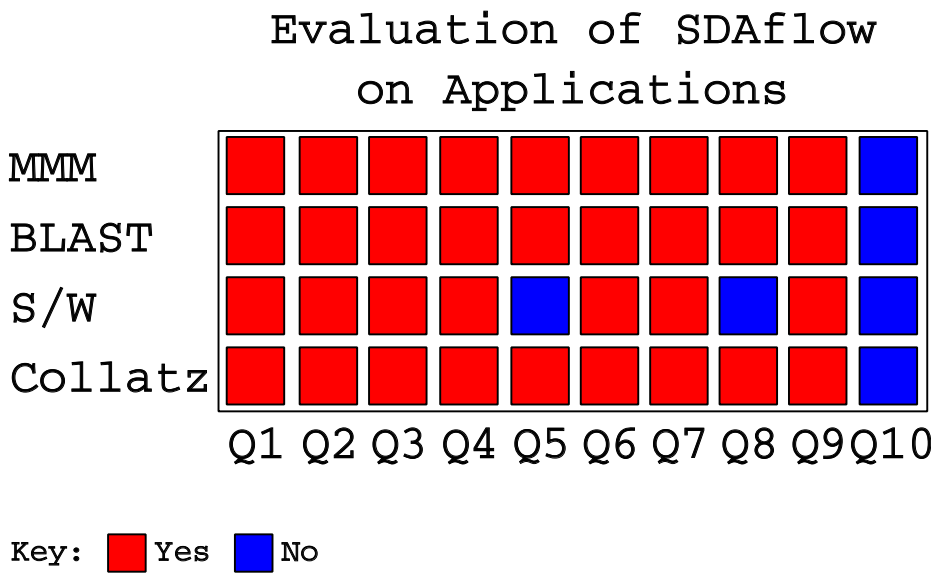


Figure 6.51: heat plot of the systematic design analysis flow's qualitative evaluation

able to quickly assemble a large number of candidates, to give the designer the greatest change of obtaining the highest performance. Overall, the performance obtained over the original base configuration is dramatic, a ≈40,000× speedup.

6.6.2 Basic Local Alignment Search Tool

The results for BLAST were also very positive, again answering in the affirmative for all but Question 10. Both BLAST and MMM are mature applications so it was anticipated that not only would both applications work well with the Systematic Design Analysis flow, but that they would also scale well. However, even though the cores were mature, in the BLAST evaluation the Systematic Design Analysis flow was able to reduce the number of base configuration while increasing the number of parallel cores. Furthermore, the performance obtained over the already well optimized single node implementation was 31.92× for 32 nodes.

6.6.3 Smith/Waterman Algorithm

The other bio application, Smith/Waterman, did not perform as well as BLAST. A large part of the problem with this core was in its immaturity. Several bugs were identified by the Systematic Design Analysis flow that made a scaled evaluation

impossible. That being said, a pleasantly unexpected result of evaluating Smith/Waterman is understanding how large of a role the Systematic Design Analysis flow and especially the Performance Monitoring Infrastructure can play during the debugging stage of a design. Overall, several questions (Question 5, 8 and 10) were answered negatively, the remaining seven questions were answered positively for this application which when combined with the newly identified debugging capabilities indicates the Systematic Design Analysis flow is useful for improving a designer's productivity.

6.6.4 Collatz Conjecture

Finally, the Collatz demonstrates the capability to quickly assemble systems with tens to nearly two hundred parallel instances of a core on a single device. Furthermore, the same design can be quickly migrated to other devices modeling a real world situation a designer would face when new devices or resources become available. This has the potential to save the designer significant redesign time and the ability of the Systematic Design Analysis flow to evaluate the complex scalability when dealing with constraints like the PLB slave limitations, indicate the flow can positively benefit the designer's productivity. All in all, the Collatz case study answered positively for all but Question 10.

CHAPTER 7: CONCLUSION

With the shift from single processor frequency scaling to resource scaling, careful attention needs to be placed on not only how to design for such systems, but how to maintain a designer's productivity as the amount of resources increases. A large part of this investigation is centered around the development of a Systematic Design Analysis flow (SDAflow) to aid a designer in the assembly of scaled systems. Towards this goal, a large assortment of tools have been created to automate routine tasks that a designer is normally responsible for performing. Furthermore, these tools can be combined to evaluate the current state of a system, both in terms of resource utilization and runtime performance. From this evaluation SDAflow can be used to generate candidate configurations to provide the designer a set of modified systems that aim to scale the design to the specified available resources or to meet a specific performance metric.

As part of SDAflow a designer can take advantage of the 26 different tools and graphical user interfaces that have been created for the purposes of system generation, iterative component synthesis and analysis, static HDL profiling, performance monitoring insertion and evaluation, and candidate configuration generation and evaluation. These tools can significantly increase a designer's productivity by not only performing routine tasks such as replication of a core across a system, but also by implementing the knowledge of an experienced hardware designer to identify and recommend configurations to a designer and then implementing them.

In addition to the tools, several custom hardware cores and interfaces have been developed to enable a greater range of flexibility in the system. In total, 24 different cores and interfaces have been included as part of this evaluation (this is in addition

to the hardware cores and designs that were part of the original applications that were evaluated in the four case studies). The goal of these cores and interfaces is to provide a higher level of abstraction from the designer to critical resources such as on-chip interconnects, off-chip networks, memory subsystems and the processor. With SDAflow, these cores can be quickly assembled and configured to meet the performance metrics set by a designer without requiring the designer to manually augment the system. Furthermore, to support these hardware cores several device drivers, control applications, and scripts have been written, enabling a designer to quickly modify existing software designs with minimal effort to take advantage of the augmented hardware infrastructure.

The overall evaluation of SDAflow has been performed on four applications as case studies to identify the benefits and to also understand the weaknesses of the flow. These applications range in maturity and complexity to evaluate the SDAflow. The four applications are: matrix-matrix multiplication, a hardware implementation of the scan and ungapped extension portions of the Basic Local Alignment Search Tool, a hardware implementation of the `flocal_align()` in the Smith/Waterman algorithm, and a hardware implementation of the Collatz conjecture's to evaluate the number of steps needed for a number to be reduced to one.

Throughout these case studies a total of 38 performance monitors have been inserted and used in the runtime performance evaluation of these applications. In addition to the performance monitors static HDL profiling of all four systems accounted for the analysis of over one hundred HDL files and the identification of several different interfaces, subcomponents, finite state machines, and registers. With the addition of component synthesis information, which was applied to these same component and subcomponents, 53 configurations were generated and evaluated. While several configurations provided satisfactory speedups in the ranges of 1× to 9× several implementations also resulted in multiple orders of magnitude performance gains. The

best performance gain came from the implementation of matrix-matrix multiplication on sixteen nodes of the Spirit cluster with an ≈40,000× speedup.

Also, the use of SDAflow provided several interesting observations. In several cases it was identified that the peak performance was limited not by the number of parallel instances, but by memory or network bandwidths. The SDAflow was able to identify these peaks and provide alternative configurations with different interfaces to memory and the network without requiring the designer to perform manual modifications to the hardware core. In other cases it was identified that through the use of SDAflow, designs could be quickly migrated to not only different FPGA devices (e.g. Virtex 4 to Virtex 5), but also could also be migrated between different processor types (hard processor to software processor). Moreover, profiling data and runtime performance data could easily be collected to give the designer a more narrow view of where a design's bottlenecks were and in some cases, was able to even identify logic errors in both software and hardware.

In short, the use of SDAflow has been shown to significantly improve the design and implementation of hardware designs that were optimized for a single FPGA device to allow for scalability and increased performance with minimal designer intervention. This dramatically improves a designer's productivity, alleviating several of the routine tasks and recommending several alternative configurations that may have otherwise not been considered. The use of SDAflow and the accompanying tools enables a designer to focus on improving the specific hardware core performance as opposed to the integration with the rest of the system. Moreover, as the design's lifetime spans several devices, SDAflow can help minimize re-design, further improving a designer's productivity.

REFERENCES

[1] P. Ross, "Why CPU frequency stalled," *IEEE Spectrum*, vol. 45, no. 4, p. 72, April 2008.

[2] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: A view from berkeley," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183, December 2006, http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html.

[3] Intel Corporation, "http://www.intel.com," last accessed July 6, 2011.

[4] Advanced Micro Devices, Inc., "http://www.amd.com," last accessed July 6, 2011.

[5] Xilinx, Inc., "http://www.xilinx.com," last accessed July 6, 2011.

[6] Altera Corporation, "http://www.altera.com," last accessed July 6, 2011.

[7] A. G. Schmidt and R. Sass, "Characterizing effective memory bandwidth of designs with concurrent high-performance computing cores," in *International Conference on Field-Programmable Logic and Applications*. IEEE Computer Society, August 2007, pp. 601–604.

[8] P. Alfke, "Evolution, revolution and convolution recent progress in field-programmable logic," in *Workshop on Electronics for LHC Experiments*, September 2001, pp. 25–31.

[9] G. E. Moore, "Cramming more components onto integrated circuits," in *Readings in Computer Architecture*, 2000, pp. 56–59.

[10] Xilinx, Inc., "ISE in-depth tutorial (ug695) v11.2," June 2009.

[11] Xilinx, Inc., "ChipScope Pro integrated logic analyzer data sheet(ds299) v1.03.a," June 2009.

[12] USB Implementers Forum (USB-IF), "Usb 2.0 specification," January 2010, http://www.usb.org/developers/docs.

[13] USB Implementers Forum (USB-IF), "Usb 3.0 specification," January 2010, http://www.usb.org/developers/docs.

[14] K. Grimsrud and H. Smith, *Serial ATA Storage Architecture and Applications: Designing High-Performance, Cost-Effective I/O Solutions*. Intel Press, 2003.

[15] 1394 Trade Association, "1394 ta specifications," January 2010, http://www.1394ta.org/developers/Specifications.html.

[16] W. T. Futral, *InfiniBand Architecture: Development and Deployment–A Strategic Guide to Server I/O Solutions*. Intel Press, 2001.

[17] Xilinx, Inc., "Command line tools user guide (ug628) v11.4," December 2009.

[18] Xilinx, Inc., "XST user guide (ug627) v11.3," December 2009.

[19] R. Sass, W. V. Kritikos, A. G. Schmidt, S. Beeravolu, P. Beeraka, K. Datta, D. Andrews, R. S. Miller, and J. Daniel Stanzione, "Reconfigurable computing cluster (RCC) project: Investigating the feasibility of FPGA-based petascale computing," in *Symposium on Field-Programmable Custom Computing Machines*. IEEE Computer Society, April 2007, pp. 127–140.

[20] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, K. Yelick, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Keckler, D. Klein, P. Kogge, R. S. Williams, and K. Yelick, "Exascale computing study: Technology challenges in achieving exascale systems," DARPA Information Processing Techniques Office (IPTO) sponsored study, Tech. Rep. TR-2008-13, 2008, http://www.exascale.org/iesp/IESP:Documents.

[21] K. Underwood and K. S. Hemmert, "Closing the gap: CPU and FPGA trends in sustainable floating-point BLAS performance," in *Symposium on Field-Programmable Custom Computing Machines*. IEEE Computer Society, April 2004, pp. 219–228.

[22] Xilinx, Inc., "ML410 embedded development platform user guide," September 2008.

[23] A. G. Schmidt, W. V. Kritikos, R. R. Sharma, and R. Sass, "AIREN: A novel integration of on-chip and off-chip FPGA networks," in *Symposium on Field-Programmable Custom Computing Machines*. IEEE Computer Society, April 2009, pp. 271–274.

[24] A. Mendon, A. G. Schmidt, and R. Sass, "A hardware filesystem implementation with multi-disk support," *International Journal of Reconfigurable Computing*, September 2009.

[25] S. Gao, A. G. Schmidt, and R. Sass, "Hardware implementation of MPI_Barrier on an FPGA cluster," in *International Conference on Field-Programmable Logic and Applications*. IEEE Computer Society, September 2009, pp. 12–17.

[26] S. Gao, A. G. Schmidt, and R. Sass, "Impact of reconfigurable hardware on accelerating MPI_Reduce," in *International Conference on Field-Programmable Technology*. IEEE Computer Society, December 2010, pp. 29–36.

[27] B. Huang, A. G. Schmidt, A. A. Mendon, and R. Sass, "Investigating resilient high performance reconfigurable computing with minimally-invasive system monitoring," in *International Workshop on High-Performance Reconfigurable Computing Technology and Applications*. IEEE Computer Society, November 2010, pp. 1–8.

[28] A. G. Schmidt, B. Huang, and R. Sass, "Checkpoint/restart and beyond: Resilient high performance computing with FPGAs," in *Symposium on Field-Programmable Custom Computing Machines*. IEEE Computer Society, May 2011, pp. 162–169.

[29] W. V. Kritikos, Y. Rajasekhar, A. G. Schmidt, and R. Sass, "A radix tree router for scalable FPGA networks," in *International Conference on Field-Programmable Logic and Applications*. IEEE Computer Society, September 2011.

[30] Xilinx, Inc., "LocalLink interface specification (sp006)," July 2011.

[31] W. J. Dally and B. Towles, "Route packets, not wires: on-chip inteconnection networks," in *Design Automation Conference*. ACM, 2001.

[32] Xilinx, Inc., "LogiCORE IP aurora v3.0 users guide 61," July 2008.

[33] D. A. Buell, J. M. Arnold, and W. J. Kleinfelder, *Splash 2: FPGAs in a Custom Computing Machine*. Wiley-IEEE Computer Society Press, 1996.

[34] J. Vuillemin, P. Bertin, D. Roncin, M. Sh, H. Touati, and P. Boucard, "Programmable active memories: Reconfigurable systems come of age," *IEEE Transactions on VLSI Systems*, vol. 4, pp. 56–69, 1996.

[35] M. Gokhale, W. Holmes, A. Kopser, S. Lucas, R. Minnich, and D. Sweely, "Building and using a highly parallel programmable logic array," *Computer*, vol. 24, no. 1, pp. 81–89, Jan 1991.

[36] K. D. Underwood, R. Sass, and W. B. Ligon, "A reconfigurable extension to the network interface of beowulf clusters," in *IEEE Conference on Cluster Computing*, October 2001, pp. 212–221.

[37] K. D. Underwood, W. B. Ligon, and R. Sass, "Analysis of a prototype intelligent network interface," *Concurrency and Computation: Practice and Experience*, pp. 751–777, 2003.

[38] Cray, Inc., "Cray (Octigabay) XD-1 supercomputer," August 2008, http://cray.com.

[39] Silicon Graphics Internation Corporation, "SGI reconfigurable application specific computing," August 2008, http://www.sgi.com.

[40] DRC Computer Corporation, "DRC reconfigurable processor units (RPU)," August 2008, http://drccomputer.com.

[41] XtremeData, Inc., "XtremeData, inc. DX1000 FPGA coprocessor module," August 2008, http://www.xtremedatainc.com.

[42] D. Burke, J. Wawrzynek, K. Asanovic, A. Krasnov, A. Schultz, G. Gibeling, and P.-Y. Droz, "RAMP blue: Implementation of a manycore 1008 processor system," in *Proceedings of the Reconfigurable Systems Summer Institute*, 2008.

[43] R. Baxter, S. Booth, M. Bull, G. Cawood, J. Perry, M. Parsons, A. Simpson, A. Trew, A. McCormick, G. Smart, R. Smart, A. Cantle, R. Chamberlain, and G. Genest, "Maxwell-a 64 FPGA supercomputer," in *Second NASA/ESA Conference on Adaptive Hardware and Systems*, 2007, pp. 287–294.

[44] C. Pedraza, E. Castillo, J. Castillo, C. Camarero, J. Bosque, J. Martinez, and R. Menendez, "Cluster architecture based on low cost reconfigurable hardware," in *International Conference on Field-Programmable Logic and Applications*. IEEE Computer Society, 2008, pp. 595–598.

[45] M. Saldana and P. Chow, "TMD-MPI: An MPI implementation for multiple processors across multiple FPGAs," in *International Conference on Field-Programmable Logic and Applications*. IEEE Computer Society, 2006, pp. 1–6.

[46] A. P. Michael Showerman, Jeremy Enos, "QP: A heterogeneous multi-accelerator cluster," in *International Conference on High-Performance Cluster Computing*, 2010.

[47] P. P. Kuen Hung Tsoi, Anson Tse and W. Luk, "Programming framework for clusters with heterogeneous accelerators," in *International Workshop on Highly-Efficient Accelerators and Reconfigurable Technologies*, 2010.

[48] NSF Center for High Performance Reconfigurable Computing (CHREC), "NOVO-G: Adaptively custom research supercomputer," April 2005, http://www.xilinx.com/support/documentation/sw_manuals/edk92i_ppc405_isaext_guide.pdf.

[49] RAMP, "Research accelerator for multiple processors," August 2008, http://ramp.eecs.berkeley.edu.

[50] J. Wawrzynek, "Adventures with a reconfigurable research platform," *International Conference on Field-Programmable Logic and Applications*, Aug. 2007, comment made during keynote. http://ce.et.tudelft.nl/FPL/wawrzynekFPL2007.pdf.

[51] A. Pant, H. Jafri, and V. Kindratenko, "Phoenix: A runtime environment for high performance computing on chip multiprocessors," *Euromicro International Conference on Parallel, Distributed and Network-based Processing*, pp. 119 –126, 2009.

[52] N. S. Woo and J. Kim, "An efficient method of partitioning circuits for multiple-FPGA implementation." in *Design Automation Conference*, 1993, pp. 202–207.

[53] K. Roy-Neogi and C. Sechen, "Multiple FPGA partitioning with performance optimization," in *International Symposium on Field-Programmable Gate Arrays*. ACM, 1995, pp. 146–152.

[54] W.-J. Fang and A. C. H. Wu, "Integrating HDL synthesis and partitioning for multi-FPGA designs," in *IEEE Des. Test*, vol. 15, no. 2, 1998, pp. 65–72.

[55] W.-J. Fang and A. C.-H. Wu, "Multiway FPGA partitioning by fully exploiting design hierarchy," in *ACM Trans. Des. Autom. Electron. Syst.*, vol. 5, no. 1, 2000, pp. 34–50.

[56] W.-J. Fang and A.-H. Wu, "A hierarchical functional structuring and partitioning approach for multiple-FPGA implementations," in *IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers*, nov. 1996, pp. 638 –643.

[57] Y.-S. Kwon and C.-M. Kyung, "Atomi: An algorithm for circuit partitioning into multiple FPGAs using time-multiplexed, off-chip, multicasting interconnection architecture," in *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 13, no. 7, jul. 2005, pp. 861 – 864.

[58] S. wei Ong, N. Kerkiz, B. Srijanto, R. Tan, M. Langston, D. Newport, and D. Bouldinq, "Automatic mapping of multiple applications to multiple adaptive computing systems," in *Symposium on Field-Programmable Custom Computing Machines*. IEEE Computer Society, 2001, pp. 218–227.

[59] M. Porrmann, J. Hagemeyer, C. Pohl, J. Romoth, and M. Strugholtz, "RAPTOR - a scalable platform for rapid prototyping and FPGA-based cluster computing," in *International Conference on Parallel Computing, Symposium on Parallel Computing with FPGAs*, September 2009, pp. 1–4.

[60] C. Reardon, A. George, G. Stitt, and H. Lam, "An automated scheduling and partitioning algorithm for scalable RC systems," in *International Conference on Engineering of Reconfigurable Systems and Algorithms*, 2010.

[61] K. H. Tsoi and W. Luk, "Axel: a heterogeneous cluster with FPGAs and GPUs," in *International Symposium on Field-Programmable Gate Arrays*. ACM, 2010, pp. 115–124.

[62] Xilinx, Inc., *Xilinx CORE Generator System*, July 2011, http://www.xilinx.com/tools/coregen.htm.

[63] Xilinx, Inc., *Embedded System Tools Reference Manual EDK 10.1*, June 2010.

[64] Altera Corporation, *System-on-Programmable-Chip (SOPC) Builder User Guide (UG-01096-1.0)*, December 2010.

[65] Xilinx, Inc., "Chipscope pro and the serial I/O toolkit," http://www.xilinx.com/tools/cspro.htm.

[66] Altera Corporation, "Design debugging using the SignalTap II embedded logic analyzer," http://www.altera.com/literature/hb/qts/qts_qii53009.pdf.

[67] M. Wirthlin, B. Nelson, B. Hutchings, P. Athanas, and S. Bohner, "FPGA design productivity existing limitations and root causes," in *FPGA Tool Flow Studies Workshop*, June 2008.

[68] M. Wirthlin, B. Nelson, B. Hutchings, P. Athanas, and S. Bohner, "FPGA design productivity potential solutions and roadmap," in *FPGA Tool Flow Studies Workshop*, June 2008.

[69] C. Pohl, C. Paiz, and M. Porrmann, "vMAGIC-automatic code generation for VHDL," *International Journal of Reconfigurable Computing*, pp. 1–10, 2009.

[70] D. Koch, "Architectures, methods, and tools for distributed run-time reconfigurable FPGA-based systems," Ph.D. dissertation, University Erlangen-Nuremberg, 2009.

[71] M. Schulz, B. S. White, S. A. McKee, H.-H. S. Lee, and J. Jeitner, "Owl: next generation system monitoring," in *Conference on Computing Frontiers*. ACM, 2005, pp. 116–124.

[72] S. Koehler, J. Curreri, and A. D. George, "Performance analysis challenges and framework for high-performance reconfigurable computing," *Parallel Comput.*, vol. 34, no. 4-5, pp. 217–230, 2008.

[73] J. Lancaster, J. Buhler, and R. Chamberlain, "Efficient runtime performance monitoring of FPGA-based applications," in *International System-on-Chip Conference*, September 2009, pp. 23–28.

[74] J. M. Lancaster and R. D. Chamberlain, "Crossing timezones in the TimeTrial performance monitor," in *Proc. of 2010 Symposium on Application Accelerators in High Performance Computing*, 2010.

[75] A. Pellegrini, K. Constantinides, D. Zhang, S. Sudhakar, V. Bertacco, and T. Austin, "CrashTest: A fast high-fidelity FPGA-based resiliency analysis framework," in *International Conference on Computer Design*, Oct. 2008, pp. 363 –370.

[76] Xilinx, Inc., *Platform Specification Format Reference Manual (UG642) v11.2*, June 2009.

[77] Python Software Foundation, *Pickle - Python object serialization*, July 2011, http://docs.python.org/library/pickle.html.

[78] Free Software Foundation, *GNU gprof*, July 2011, http://www.gnu.org/software/binutils.

[79] Xilinx, Inc., *XPS Central DMA Controller (DS579) v2.01b*, June 2009.

[80] Y. Rajasekhar, W. V. Kritikos, A. G. Schmidt, and R. Sass, "Teaching FPGA system design via a remote laboratory facility," in *International Conference on Field-Programmable Logic and Applications*. IEEE Computer Society, September 2008, pp. 687–690.

[81] Reconfigurable Computing Systems Lab, "http://www.rcs.uncc.edu," University of North Carolina at Charlotte, last accessed July 6, 2011.

[82] Xilinx, Inc., "ML510 embedded development platform user guide," September 2009.

[83] Xilinx, Inc., "XUPV5 xilinx university program virtex-5 development platform user guide," September 2009.

[84] Xilinx, Inc., *MicroBlaze Soft Processor Core*, June 2011, http://www.xilinx.com/tools/microblaze.htm.

[85] Netlib Repository, "BLAS (basic linear algebra subprograms," July 2011, http://www.netlib.org/blas.

[86] J. Liang, R. Tessier, and O. Mencer, "Floating point unit generation and evaluation for FPGAs," in *Symposium on Field-Programmable Custom Computing Machines*. IEEE Computer Society, April 2003, pp. 185–194.

[87] S. Chen, R. Venkatesan, and P. Gillard, "Implementation of vector floating-point processing unit on FPGAs for high performance computing," *Canadian Conference on Electrical and Computer Engineering*, pp. 881–886, May 2008.

[88] W. B. Ligon, S. McMillan, G. Monn, K. Schoonover, F. Stivers, and K. D. Underwood, "A re-evaluation of the practicality of floating-point operations on FPGAs," in *Symposium on Field-Programmable Custom Computing Machines*. IEEE Computer Society, 1998, p. 206.

[89] M. J. Beauchamp, S. Hauck, K. D. Underwood, and K. S. Hemmert, "Embedded floating-point units in FPGAs," in *International Symposium on Field-Programmable Gate Arrays*. ACM, 2006, pp. 12–20.

[90] K. S. Hemmert and K. D. Underwood, "An analysis of the double-precision floating-point fft on FPGAs," in *Symposium on Field-Programmable Custom Computing Machines*. IEEE Computer Society, 2005, pp. 171–180.

[91] X. Wang, S. Braganza, and M. Leeser, "Advanced components in the variable precision floating-point library," in *Symposium on Field-Programmable Custom Computing Machines*, vol. 0. IEEE Computer Society, 2006, pp. 249–258.

[92] L. E. Cannon, "A cellular computer to implement the kalman filter algorithm," Ph.D. dissertation, Montana State University, 1969.

[93] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool," *Journal of Molecular Biology*, vol. 215, pp. 403–410, October 1990.

[94] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino-acid sequence of two proteins," *Journal of Molecular Biology*, vol. 48, pp. 443–453, 1972.

[95] T. Smith and M. S. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology*, vol. 147, pp. 195–197, 1981.

[96] S. Datta, P. Beeraka, and R. Sass, "RC-BLASTn: Implementation and evaluation of the BLASTn scan function," in *Symposium on Field-Programmable Custom Computing Machines*. IEEE Computer Society, May 2009, pp. 88–96.

[97] S. Datta and R. Sass, "Scalability studies of the BLASTn scan and ungapped extension functions," in *International Conference on Reconfigurable Computing and FPGAs*. IEEE Computer Society, December 2009.

[98] A. G. Schmidt, S. Datta, A. A. Mendon, and R. Sass, "Investigation into scaling I/O bound streaming applications productively with an all-FPGA cluster," *International Journal on Parallel Computing*, April 2011, [*Under Review*].

[99] MPI-Forum, "Message passing interface forum," January 2009, http://www.mpi-forum.org.

[100] A. G. Schmidt, S. Datta, A. A. Mendon, and R. Sass, "Productively scaling I/O bound streaming applications with a cluster of FPGAs," in *Symposium on Application Accelerators in High-Performance Computing*, July 2010.

[101] NCBI User Services, "Genbank overview," August 2005, http://www.ncbi.nlm.nih.gov/Genbank.

[102] S. Ganesh, "Implementation of the smith-waterman algorithm on FPGAs," Master's thesis, University of North Carolina at Charlotte, 2009.

[103] W. R. Pearson, "FASTA sequence comparison at the university of virginia," July 2011, http://fasta.bioch.virginia.edu/fasta_www2.

[104] L. Collatz, "Einschlieungssatz fr die charakteristischen zahlen von matrize," in *Mathematische Zeitschrift*, 1942, pp. 221–226.

[105] R. K. Guy, *Unsolved problems in Number Theory*. New York, New York: Springer Science+Business Media, Inc., 2004.