Matching Genetic Sequences in Distributed Adaptive Computing Systems

William J. Worek

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

Dr. Mark T. Jones, Chair
Dr. Peter M. Athanas
Dr. James M. Baker

July 29, 2002
Bradley Department of Electrical and Computer Engineering
Blacksburg, Virginia

Keywords: configurable computing, FPGA, adaptive computing, genetic sequences, pattern matching

Copyright © 2002, William J. Worek

Matching Genetic Sequences in Distributed Adaptive Computing Systems

William J. Worek

Abstract

Distributed adaptive computing systems (ACS) allow developers to design applications using multiple programmable devices. The ACS API, an API created for distributed adaptive computing, gives developers the ability to design scalable ACS systems in a cluster networking environment for large applications. One such application, found in the field of bioinformatics, is the DNA sequence alignment problem. This thesis presents a runtime reconfigurable FPGA implementation of the Smith-Waterman similarity comparison algorithm. Additionally, this thesis presents tools designed for the ACS API that assist developers creating applications in a heterogeneous distributed adaptive computing environment.

To Everyone,

so I don't leave out anyone.

Acknowledgements

While working on my thesis, a number of people have provided an enormous amount of support. First, I would like to thank my committee, Dr. Athanas, Dr. Baker, and, especially, my advisor Dr. Jones. A debt of gratitude is owed to all of the people that had a hand in developing the ACS API. A special thanks goes to Kiran Puttegowda and James Hendry. Without their support, none of this would have been possible. I would also like to thank all of the diligent workers in the CCM Lab for providing me with entertainment as well as guidance. I would like to acknowledge DARPA and ISI for supporting the development of this project. A many thanks go to all of the people that have helped me get to this stage in my life. And finally, I would like to acknowledge the support of all of my friends and family for all of their help.

William J. Worek

July 2002

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Motivation

Since the introduction of the Human Genome Initiative [1], genetic sequences are being discovered at an ever increasing rate. It is a common problem in the field of bioinformatics to compare a gene sequence to a database of sequences, or databases of sequences against another database. Gene repositories, such as GenBank [2] or Swiss-Prot [3], used in these comparisons have grown to be quite large and require a great deal of computation to compare against. Faster methods of comparing gene sequences are required in order to keep up with the size of the databases.

Developing large distributed applications on adaptive computing systems requires knowledge of hardware design languages, FPGA board APIs, and a method for communicating between computers. Becoming proficient in all of these areas is a difficult task. Furthermore, there are few resources for debugging and developing applications in a distributed adaptive computing environment.

1.2 Thesis Statement

This thesis explores the use of the Adaptive Computing System API (ACS API) for developing a distributed gene matching application. This thesis contributes the following.

- It demonstrates and analyzes the use of the ACS API in developing distributed systems.
- It suggests methods for designing distributed adaptive computing systems.
- It analyzes the results of using a runtime reconfigurable approach to comparing genetic sequences on FPGAs.
- It demonstrates and analyzes the use of heterogenous nodes in an adaptive computing system.

A runtime reconfigurable gene matching implementation of the Smith-Waterman [4] algorithm was developed as part of this research. To assist in developing distributed adaptive computing systems, a textual and graphical interface was created for the ACS API.

1.3 Thesis Organization

This thesis is organized in the following manner. Chapter 2 presents the background for the work presented in this thesis. The background discusses related research and concepts that contributed to this thesis. Chapter 3 describes the ACS API. It explains the development environment that was used in designing the distributed systems. Chapter 4 describes the user interface designed to assist in developing applications. Chapter 5 presents the hardware implementation of the genetic sequence alignment algorithm. Chapter 6 describes the

distributed adaptive computing systems developed to utilize the hardware described in Chapter 5. Chapter 7 presents an analysis of the results from the systems developed. Chapter 8 summarizes the research and proposes avenues for future work.

Chapter 2

Background

This section will discuss the background information used in this thesis. The background is divided into four sections. The first section discusses adaptive computing devices and examines the resources available on two boards used. The second section explains the Smith-Waterman [4] pattern matching algorithm. The third section gives an overview of the genetic code matching problem, as well as examining alternative algorithms and commercial implementations. The final section examines applications that make use of pattern matching.

2.1 Adaptive Computing Devices

Adaptive Computing Systems (ACS) are comprised of reprogrammable hardware devices. These hardware devices contain programmable logic devices (PLDs) such as commercially available Xilinx Field Programmable Gate Arrays (FPGAs). This section gives an overview of PLDs. This section also gives a description of the SLAAC-1V and the Osiris, the two Xilinx-based platforms used in this thesis.

2.1.1 Overview

Digital logic devices are typically divided into two broad categories, Application Specific Integrated Circuits (ASICs) and general purpose processors (GPPs). GPPs, such as CPUs in a personal computer, are very flexible devices that are characterized by a high degree of programmability at the sacrifice of performance. The GPPs limited performance is due to the overhead of decoding and executing instructions. ASICs are integrated circuits designed for a specific task. In contrast to GPPs, a very high level of performance and limited flexibility defines an ASIC. ASICs may have a limited amount of flexibility that is built into the circuit during the design. Any flexibility typically results in a sacrifice in performance [5].

Programmable Logic Devices (PLDs) allow a high degree of flexibility without sacrificing very much in performance. PLDs are approximately three times slower than ASICs, but can be significantly faster than GPPs [5]. A typical PLD, such as a Xilinx FPGA, is comprised of three parts, Input/Output Blocks (IOBs), Configurable Logic Blocks (CLBs), and a programmable interconnection network. IOBs are interfaces for connections external to the chip. A large array of CLBs implements the desired logic. The programmable interconnection controls signal routing between CLBs [6].

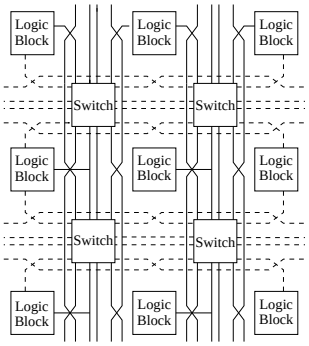

Figure 2.1: Basic FPGA Components

The CLB consists of function generation, internal routing, and memory. To generate func-

tions, the CLB uses an array of lookup tables (LUTs). The internal routing allows signals to be connected to the memory or continue straight out of the CLB. D-flip-flops (DFF) provide fast memory for storing intermediate values. The ability to store values local to the CLB allow a high degree of pipelining in the circuit. The CLBs and interconnections are programmed using anti-fuses or memory. Anti-fuses are one-time programmable cells that create permanent connections when stimulated with a current. Modern PLDs use memory units such as Static RAMs (SRAMs) to program the device. Figure 2.1 shows logic blocks (LB) in relation to programmable switch matrices (PSM). A configuration defines the hardware which the PLD implements by specifying the values in all of the LUTs and SRAMs used in the CLBs and programmable interconnections [5].

2.1.2 SLAAC-1V

The Information Sciences Institute (ISI) East, a part of the University of Southern California (USC), developed the SLAAC-1V under the Systems Level Application of Adaptive Computing (SLAAC) project [7]. The Defense Advanced Research Project Agency (DARPA) sponsored the SLAAC project, which combined the resources of several different organizations including the Configurable Computing Machines Laboratory of Virginia Tech.

The SLAAC-1V [8] is a FPGA platform capable of being connected to a host computer via a 32-bit 33 MHz PCI connection. Three Virtex 1000 FPGAs reside on the SLAAC-1V board. The Virtex XCV1000 FPGAs contain the equivalent of 1 million system gates and 27,648 Logic Cells. Each part hosts 64x96 CLBs and 128K bytes of Block RAM. Each CLB in the Virtex chip is composed of two slices. A slice is the most primitive logic block contained in an FPGA, made up of 2 DFF, 2 4-input LUTs, and assorted logic for internal routing [6]. The three FPGAs are named X0, X1, and X2. The X0 FPGA contains the interface to the CPU, known as the Interface FPGA (IF). For convenience, X0 and IF are referred to as

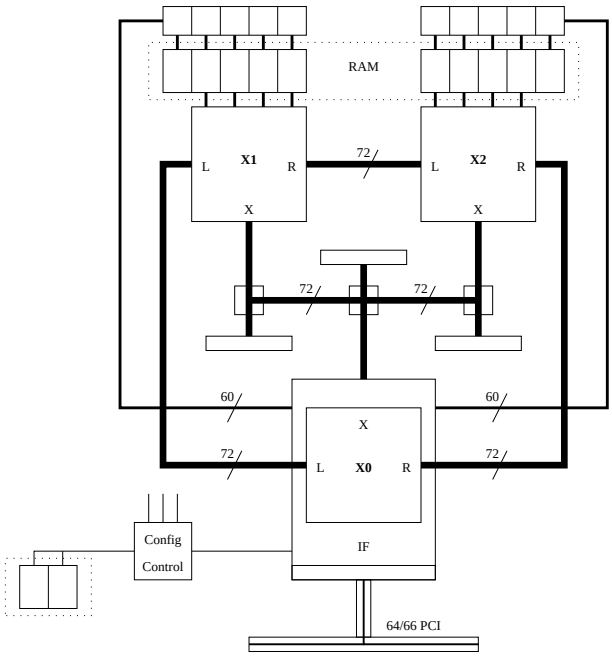

Figure 2.2: SLAAC-1V Board Architecture

separate units within the SLAAC-1V.

The SLAAC-1V also contains memory on the board. The user FPGAs X1 and X2 can each access four zero bus turnaround (ZBT) SRAMs. The X0 FPGA only has two ZBT SRAMs available to it. All the ZBT SRAMs have 36 bit words and are 256K addressable. ZBT RAMs allow a read or write during every clock cycle, allowing 100 percent utilization of the bus. The user can specify whether the IF will preempt the FPGAs to allow the host access to the memories. During preemption, the clocks on the FPGAs are halted while the memories are being used. The user can also specify that the IF use non-preemptive memory access. In this case, the user must ensure that the host and the FPGAs do not access the memories at the same time [8].

The SLAAC-1V has two sets of FIFOs. FIFO A0 is an input to the board and has a

complement output FIFO named B0. Likewise, FIFO A1 is used for input and FIFO B1 is used as an output. The A0/B0 FIFO is 64 bits wide, but has no buffering. This type of FIFO is referred to as a mailbox. The A1/B1 FIFO also has words which are 64 bits wide. In contrast to the mailbox FIFOs, the second type of FIFOs can buffer up to 256 words [8].

A crossbar exists between X0, X1, and X2. The crossbar is configured by the X0 element and allows data to be broadcast between the processing elements. In addition to the crossbar, there is a 72-bit connection that creates a ring. The ring connection, using RIGHT and LEFT descriptors, is useful for streaming data through all of the processing elements. Figure 2.2 shows the architecture of the SLAAC-1V board.

2.1.3 Osiris

The Osiris board [9] is the latest generation from the SLAAC family of FPGA boards. Like its predecessors, the Osiris board was developed at ISI-East under the SLAAC project sponsored by DARPA. The Osiris board has many improvements in hardware over the SLAAC-1V. The design of the drivers and support software has also changed.

The Osiris board contains two Virtex II FPGAs. One FPGA, the IF, is used as an interface between the FPGA board and the host computer. The interface FPGA is a Virtex II 1000 (XC2V1000) part. The second FPGA, the XP, is a user programmable device available for implementing hardware designs. The user FPGA is a Virtex II 6000 (XC2V6000) part [9].

The Virtex II 1000 device contains the equivalent of one million system gates. A total of 720K of block RAM bits reside on this device. In contrast, the Virtex II 6000 is a much larger device, containing the equivalent of six million system gates. A total of 2,592K block RAM bits are contained in this device. The number of configurable/complex logic blocks (CLBs) increase from 40x32 in the 1000 part to 96x88 in the 6000 part [10].

The Osiris board contains two types of memory available to the XP and two types of memory accessible to the IF. The memory attached to the IF is typically used for holding configurations or partial reconfigurations. Four megabytes of flash memory is available to the IF along with six megabytes of asynchronous SRAM [9].

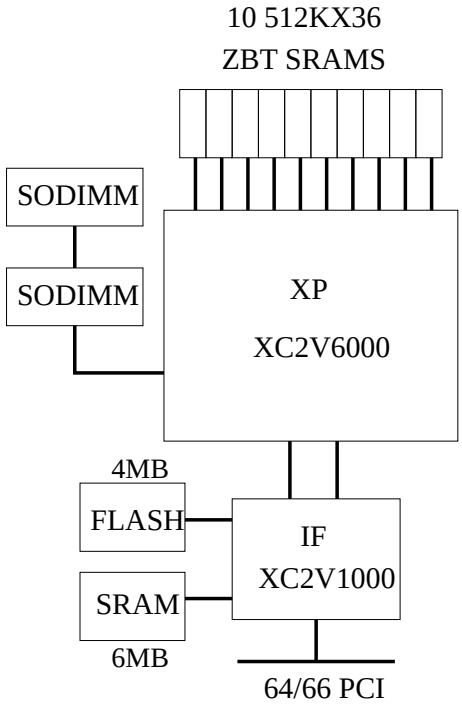

Figure 2.3: Osiris Board Architecture

Two types of memory are also available to the XP. The XP has access to 10 SRAM memories. Each SRAM is a 512Kx36 ZBT capable of running at 200 MHz. The XP can also access two Micron 256MB SDRAMs, for a total of 512MB of PC133 memory. A memory ring component, part of the VHDL library, controls the memory available to the XP. The two types of memory are available to the XP and the host. The host can access the memory using preemption and non-preemption. During preemption, the XP processing will halt while the host accesses the memory. Preemption ensures a seamless visualization by the XP. If non-preemption is preferred, the XP must not access the same memory resource as the host at

the same time. The XP can check a memory busy signal to ensure exclusive access to the memory resource [9].

The Osiris board interfaces to the host CPU via a 64-bit 66 MHz PCI connection. The Osiris Board streams data through an input FIFO named FIFO-0 and an output FIFO named FIFO-0. While the two FIFOs have the same name, they are separate FIFOs. Each FIFO is 64 bits wide and 256 words deep. The architecture for the Osiris board is shown in Figure 2.3.

2.2 Smith-Waterman Pattern Matching Algorithm

Pattern matching problems appear in many different disciplines. Algorithms designed to solve the pattern matching problem have evolved over time. This section discusses the Smith-Waterman algorithm [4]. In this algorithm, a pattern P is to be matched against a text T. The pattern P consists of m letters, in other words $P = p_1 p_2 \dots p_m$ such that for each i between 1 and m, $p_i \in S$ where S is the alphabet. T consists of n letters of the same alphabet S or, more formally, $T = t_1 t_2 \dots t_n$ with each $t_i \in S$. For ease of explanation, the pattern P will always be shorter or equal in length to the text T ($m \leq n$).

Smith and Waterman devised an algorithm for matching similar patterns [4]. The Smith-Waterman (SW) algorithm compares a pattern P to text T and calculates the penalty required to change P into T. Due to the fact that P and T may not match exactly, the penalty will take into account the number of insertions, deletions, and substitutions needed to convert the strings to match each other. This penalty is referred to as the edit distance.

Let s be the penalty for substituting characters if a mismatch is found. Let g be the penalty for a gap. We will assume a gap penalty for an insertion is the same as a gap penalty

for a deletion. For instance, if P= CAT and T = CGG, then the penalty would be two substitutions; one for substituting A for G and one for T for G. If P = CAT and T = CAAT then the penalty would be one deletion or gap for removing the second A in sequence T.

$$H_{ij} = max \begin{cases} 0 \\ H_{i-1,j-1} + s(t_i, p_j) \\ max\{H_{i-k,j} - W_k\} \\ max\{H_{i,j-l} - W_l\} \end{cases} \tag{2.1}$$

where $1 \leq i \leq n$ and $1 \leq j \leq m$ and,

H	the SW matrix,
$s(t_i, p_j)$	similarity function between characters, and
W_k	penalty for a deletion of length k.

The SW algorithm solves a similar matching problem using a dynamic programming approach. The complete algorithm is shown in Equation 2.1 [4]. The algorithm uses the solutions from smaller problems to create the larger solution and in the process, creates a matrix of edit distances. The value in cell $H_{i,j}$ represents the degree of similarity between the sequences up to t_i and p_j. A simplification to the SW algorithm was presented by Lipton and Lopresti [11]. In the modified SW algorithm, each element of the matrix uses three other elements to compute its value. Table 2.1 shows the matrix for computing element d.

$$d = min \begin{cases} \begin{cases} a : T_i = P_j \\ a + s : T_i \neq P_j \end{cases} \\ b + g \\ c + g \end{cases} \tag{2.2}$$

			T_i
			$\vdots$
		a	b
P_j	...	c	d

Table 2.1: Internal Matrix of SW Algorithm

$s = 5$			C	
$g = 8$			$\vdots$	$a = 7$
T_i ='C'		7	12	$b = 12$
P_j ='A'	A	... 10	d	$c = 10$

$$d = min\{a + s, b + g, c + g\} = 12$$

Table 2.2: Example Matrix Element Calculation

Equation 2.2 shows the modified SW computation required for each element in the edit distance matrix [11]. In this equation, g is the gap penalty, the cost of an insertion or a deletion, and s is the penalty for a substitution, a mismatch between characters. Table 2.2 shows an example computation of a matrix element. Because the characters T_i and P_j result in a substitution, the $a + s$ penalty is used to compute d. The SW Algorithm computes all values in the matrix in order to generate the global edit distance in the bottom right cell. For the n x m matrix, the complexity to compute all values is O($n * m$). Because data dependencies exist only on top and to the left of each cell, diagonal values in the SW matrix can be computed concurrently. The parallelized SW Algorithm computes up to n operations at each step. However, due to the data dependency, the algorithm cannot achieve perfect speedup. Thus, the algorithm computes the matrix in $n + m$ steps assuming n available processing elements.

2.3 Gene Matching

The gene-matching problem is actually a set of problems with slight variations. When analyzing a gene-matching algorithm it is important to note whether it is solving global or local alignment, using affine or linear scoring, and if the algorithm compares DNA or proteins [12].

When comparing genetic codes, scientists typically are comparing either deoxyribonucleic acid (DNA) or proteins. DNA is a chemical inside of cells that is made up of four possible nucleotide bases, Adenine, Thymine, Guanine, and Cytosine. These nucleotides are abbreviated A, T, G, and C. The four bases combine in pairs to create rungs in a twisted ladder called a double helix. Adenine always combines with Thymine and Guanine always combines with Cytosine. The bases form words that define the creation of amino acids, the building blocks for proteins. There are 20 different amino acids which chain together to form proteins. Proteins perform a wide variety of activities inside of the body. Missing or altered proteins are the cause of many diseases [1].

Scientists sometimes wish to match entire genes against each other, requiring a global alignment algorithm. However, sometimes, scientists wish only to find a match in a small section of the gene. When only a small section of the gene is required to match, a local alignment algorithm is used [12].

A variation in the way matches are scored provides the last variation in gene-matching algorithms. All of the algorithms have penalties for a gap, an insertion, or a substitution. However, some algorithms provide a larger penalty for an initial gap penalty and smaller penalties for gap extensions. This type of variable scoring uses affine gap penalties. The alternative to affine gap penalties is linear gap penalties, where each insertion or deletion incurs the same cost as the previous one [12].

In addition to the Smith Waterman algorithm, there are a number of other algorithms used to solve the gene-matching problem. Both FASTA [13] and the Basic Local Alignment Search Tool (BLAST) [14] use heuristics to reduce the complexity of the algorithm. Both FASTA and BLAST compare small segments of the query to the database. The assumption in both algorithms is that good matches have a large number of substrings with exact matches. FASTA first computes a position-specific comparison of the genes to generate sets of matches on the same diagonal of the SW matrix. Using the diagonal with the most matches, the SW Matrix is computed for a small band surrounding the chosen diagonal. BLAST operates using a similar technique. However, instead of performing a gapless alignment with a specific gene in the database, the algorithm compares the query sequence against a set of genes with common structure, function or evolutionary origin. BLAST associates a higher scores when the set of genes have common entries. BLAST then takes the highest matches and computes a small section of the SW matrix. These heuristic approaches can miss matches and produce false positives. Both FASTA and BLAST improve the speed of the SW algorithm at the cost of sensitivity in comparing genes. BLAST becomes less precise in finding matches when a family of genes has less in common.

Several companies have produced commercial solutions to the gene matching problem. Time Logic [15] and Compugen [16] have produced products that exploit the use of FPGAs to accelerate their solutions. Both speed up their inner loops of their algorithms by using FPGA boards. The Paracel [17] Gene Matcher2 [18] product uses ASICs and software designed for a Linux distributed system. Implementations of BLAST and FASTA are available for download over the Internet.

2.4 Pattern Matching Applications

Matching patterns is a common function that is useful in a variety of different fields. This section will give an overview of a few of these fields. The pattern matching algorithm discussed in this paper is defined by a large database, a high query load, and noisy data, requiring similarity comparisons.

Network Intrusion Detection (NID) systems are created to use two types of detections. An anomaly detector generates a profile of normal network traffic to be used as the database and queries this database with the current network traffic. A misuse detector queries the network traffic against a preexisting database of attacks, known as signatures. Most intrusion detection systems use perfect match algorithms to compare data. Snort [19], a common packet sniffing program for Linux, uses the perfect matching Boyer-Moore algorithm [20] to compare data.

Fingerprint matching is another important pattern matching application. The first step in fingerprint matching is to identify areas of interest, known as minutiae. The minutiae describe a collection of anomalies within a fingerprint, such as ridge endings or bifurcation [21]. Because fingerprints can be stretched or smeared, most fingerprint applications desire a similarity score rather than a binary match or mismatch. An FPGA implementation of the fingerprint problem was created on the SPLASH-2 [22], an early Xilinx enabled board. The SPLASH-2 implementation has been shown to be over 1,500 times faster than a sequential solution on a SPARC workstation.

Voice recognition and face detection applications also use similar matching algorithms. It has been shown that the face detection problem is a two dimensional representation of a voice recognition problem and is simple to translate between the two applications. The Hidden Markov Model (HMM) is a popular model used in voice [23] and face recognition. The HMM

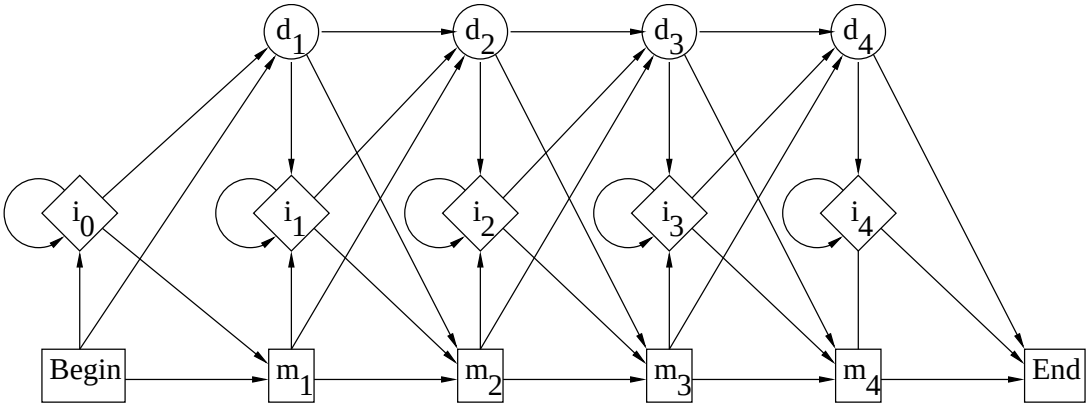

Figure 2.4: Example of a Linear Hidden Markov Model

uses state transitions to represent a sequence alignment. There are three types of states in the HMM, match (m) states, insertion (i) states, and deletion (d) states. Lines between states represent probabilities in transitioning between those states. The HMM model uses position-specific scoring, rather than pairwise scoring as in the Smith-Waterman, BLAST, and FASTA models. The HMM is compared or aligned to a query sequence to determine if the sequence belongs to the same family as the HMM. Figure 2.4 shows a simple HMM model. A parallelized HMM application was implemented to match genes by Hughey [24].

Chapter 3

ACS API Description

The ACS API is an object-oriented approach to controlling distributed adaptive computing systems. The design of the ACS API is scalable and allows users to write single node applications or develop applications using many nodes. The ACS API is a cross-platform interface that currently supports both Windows and Linux. By developing the code in an open-source environment, the ACS API is able to evolve and support more FPGA boards as well as allow users to increase its functionality. A set of default behaviors are defined by the ACS API, which makes the API a simple way to control complex systems. The user need only write one host program to control the entire system regardless of the number of nodes in the system. This chapter provides an overview of the ACS API presented in [26] and [27]. The three sections presented in this chapter divide the ACS API into broad categories of functionality for controlling the distributed ACS environment. The system management section describes the methods for controlling the overall system. The second section discusses how the API controls individual boards. The final section describes the methods for transmitting data to and from the boards in the system.

3.1 System Management

The *host program* is simple program written by the user that controls the adaptive computing system. The first stage of the *host program* is to instantiate a *system* object. The *system* is composed of an array of *nodes* and an array of *channels*. *Nodes* are objects that represent FPGA boards in the system. *Channels* are logical connections between *nodes'* FIFO ports.

The user creates an eXtensible Markup Language (XML) [28] file to define the system. Listing 3.1 shows an example ACS API configuration file. The *commtype* field defines the interprocessor communication type, such as the Message Passing Interface (MPI)[29]. Within the *boards* element, any number of *nodes* may be defined and configured with the *PE* field, such as the Osiris board shown in the example. The array of *channels* is constructed in the *channels* field. Each *channel* element defines the endpoints using the *src* and *dest* attributes. The user's XML configuration file is a simple, reusable method for describing the ACS system.

Listing 3.1: Sample XML Configuration File

```
1  <?xml version='1.0' encoding='UTF-8'?>
2   <!DOCTYPE config SYSTEM "/project/acs_api/src/config/acs_conf.dtd">
3  <config>
4  <header>
5     <commtype> MPI </commtype>
6     <version> 0.03 </version>
7     <author> William J. Worek </author>
8     <desc> This is a sample config file </desc>
9  </header>
10 <boards>
11    <osiris board_id="node0" location="local"  ctrl_proc_dir ="/project/acs_api/bin" >
```

```
12      <PE pe_num="0" prog_file="xp.bit" />
13    </osiris>
14      ⋮
15  </boards>
16  <channels>
17    <channel>
18      <src> <host port="HOST_OUT"/> </src>
19      <dest> <node board_id="node0" port="OSIRIS_FIFO_A0" /> </dest>
20    </channel>
21      ⋮
22  </config>
```

The host program uses the *ACS_XMLConfig* class to parse the XML configuration file and create the *system* object. Listing 3.2 contains the beginning code segment for an example ACS host program. After instantiating an *ACS_XMLConfig* object, the XML configuration file name is set. The XML file is parsed during the *getSystem* function to generate an array of nodes and an array of channels. The arrays are then passed to the *ACS_Initialize* function to instantiate the *system* object. *ACS_Initialize* also starts the *control processes* on each host processor in the system.

Listing 3.2: Example ACS Host Program

```
1  #include "acs.h"
2  #include "acs_system.h"
3  #include "acs_xmlconfig.h"
4  void main(int argc, char ** argv)
5  {
6    ACS_SYSTEM * system;
```

```
7   int  node_id, node_num;
8   ACS_XMLConfig config;
9   config.setFilename(argv[1]);
10  config.getSystem(&system);
11  config.getNodeCount(&num_sites);
12  ⋮
```

3.2 Board Management

The ACS API provides several functions for managing individual boards. Boards may either be local on the host machine or remote on a networked machine. The ACS API uses an object-oriented design to hide the location of the nodes in the system. Figure 3.1 shows the class hierarchy implemented in the API. Local nodes implement functions by calling the underlying board-specific API. Remote nodes package the parameters of the function and pass them across the network.

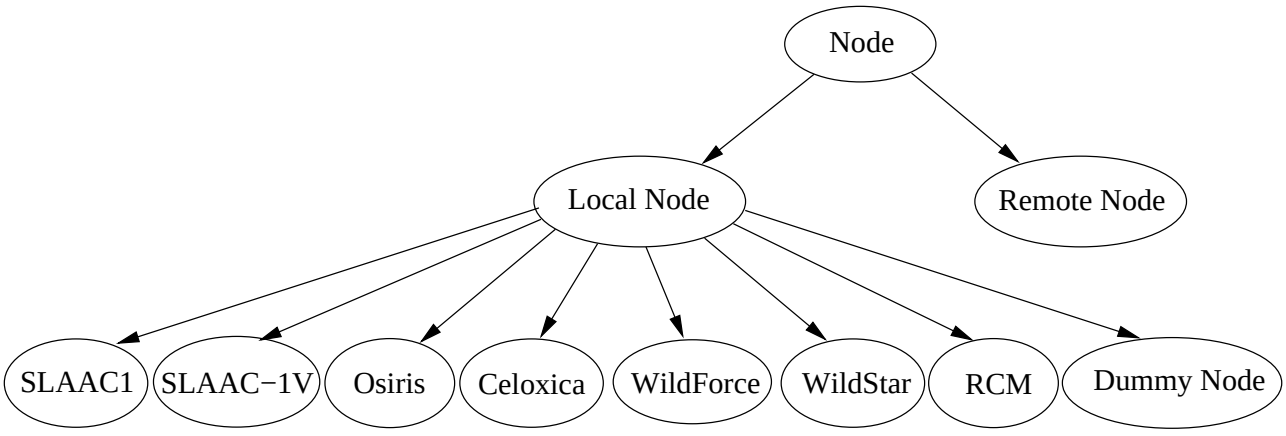

Figure 3.1: Hierarchy of Node Classes

Listing 3.3 shows a code segment from an example host program that demonstrates the use of board management functions. FPGA boards have user clocks that are used to control the speed of the implemented hardware. The *ACS_Clock_Set* and *ACS_Clock_Get* are used to control and query the clocks. The *ACS_Run* and *ACS_Stop* functions are used to turn the clocks on and off. The FPGA has reset signals that are controlled by the *ACS_Reset* function. Finally, there are methods, not shown in the example, for loading hardware configurations onto the FPGA, such as the *ACS_ConfigureFromFile* function. The ACS API reduces information sent to the nodes by caching FPGA configurations on the board.

Listing 3.3: Example ACS Host Program (cont. from Listing 3.2)

```
13    ACS_CLOCK clock;
14    for (node_num = 0; node_num < num_sites; node_num++) {
15        config.getNodeNumber(node_num,&node_id);
16        clock.frequency = 50.0;
17        clock.countdown = 0;
18        ACS_Clock_Set(&clock, node_id, system, &status);
19        ACS_Clock_Get(&cur_clock, node_id, system, &status);
20        ACS_Reset(node_id, system, pe_mask, 1, &status);
21        ACS_Run(node_id, system, &status);
22        ACS_Reset(node_id, system, pe_mask, 0, &status);
23    }
```

3.3 Data Management

This section describes the functions used for passing data and control information between the host program and the nodes. Specifically, the methods for streaming data using FIFOs,

accessing onboard memory, and manipulating registers are described. The ACS API, as mentioned in Section 3.2, makes all board functions uniform despite the location of the node in the system.

Listing 3.4 shows the final code segment of the example host program. This segment of code shows examples of the methods for communicating with the FPGA board. The *ACS_Dequeue* and *ACS_Enqueue* functions stream data through the specified port on the host node. The *control process* passes the data through the ACS *channels* to the board according to the structure defined in the *system* object. Registers can be used for a variety of functions, including sending control information to the FPGA, and providing status information such as the FIFO empty and full registers. The *ACS_Reg_Read* and *ACS_Reg_Write* functions take parameters for the node, address, and size of the register. Onboard memory can be used for providing initialization data or retrieving accumulated data post processing. The *ACS_Write* and *ACS_Read* functions require parameters specifying the node, processing element, and memory bank.

Listing 3.4: Example ACS Host Program (cont. from Listing 3.3)

```
24  unsigned char      out_buffer [8];
25  unsigned char      in_buffer [8];
26  ACS_Dequeue(in_buffer, 8*sendsize, 0, system, &status);
27  ACS_Enqueue(out_buffer, 8*sendsize, 1, system, &status);
28  ACS_REGISTER *reg;
29  reg.size = 64;
30  ACS_Reg_Read(system, node_id, 0x17, reg);
31  ACS_Reg_Write(system, node_id, 0x18, reg);
32  ACS_ADDRESS *addr;
33  addr.pe = 1;
34  addr.mem = 4;
```

```
35      addr. offset  = 0;
36      ACS_Read(out_buffer, 1024, node_id, address, system, &status);
37      addr.pe = 2;
38      ACS_Write(in_buffer, 1024, node_id, address, system, &status);
39      ACS_System_Destroy(system, &status)
40      ACS_Finalize();
41  }
```

Chapter 4

ACS API User Interface

The ACS API was designed to provide a simple, portable, and scalable interface for developing applications on adaptive computing systems. This section describes two interfaces developed as part of this thesis for streamlining the process of creating ACS applications. The ACS API provides a textual interface that allows the user to run scripts and run ACS commands at a command prompt. The second tool is a graphical user interface (GUI), which is the simplest method for interacting with the system.

4.1 Textual Interface

The ACS API textual interface is a useful tool for debugging and constructing applications. The textual interface was developed using the tool command language (TCL) scripting language for creating the textual interface. TCL was developed at the University of California, Berkeley in the 1980s [30]. The scripting language was initially used to run tools for designing integrated circuits.

TCL, like other scripting languages, are useful for "gluing" applications. Scripting languages provide a higher level of programming compared to system programming languages. System programming languages are typically faster and have stronger typing. It is possible to create entire applications simply by creating a script. Applications may also be debugged by running the program command by command.

The ACS textual debugger provides an interface with all the ACS API functions available to the user. The user can combine several function calls into a script. Scripts may be written to perform an entire application and are run by executing them as a command line parameter. Commands can also be run by starting an ACS debugging shell and typing in each command one by one.

While the ACS textual debugger is capable of running applications, it is recommended for debugging purposes only. TCL, like most scripting languages, is an interpreted language. As such, it is unlikely that a script run program will exhibit the same performance as a conventional program created using a system programming language. But, because TCL is interpreted, debugging applications is simpler since scripts do not need to be recompiled every time a change is made.

The textual debugger uses the *tclreadline* package. *Tclreadline* provides the user with a number of convenience functions, such as tab completion and command history. Tab completion is a useful feature that makes learning how to use the ACS API easier because the user no longer needs to remember the full name of the command.

4.2 Graphical Interface

Packaged with the ACS API is a graphical user interface intended for debugging applications. The graphical interface uses Qt [31], a cross-platform C++ GUI development kit. The ACS graphical interface is a simple and effective tool for testing and debugging the functionality of a hardware design without having to create a specific program.

The graphical interface to the ACS accepts an XML configuration file as an optional command line input. Alternatively, the user may configure the system by opening an XML configuration file from the file menu. Once the system is configured, a set of tabs appears in the main window. Each tab represents a different set of functions for accessing the FPGA board. The user may change between boards by selecting the appropriate node from the view menu.

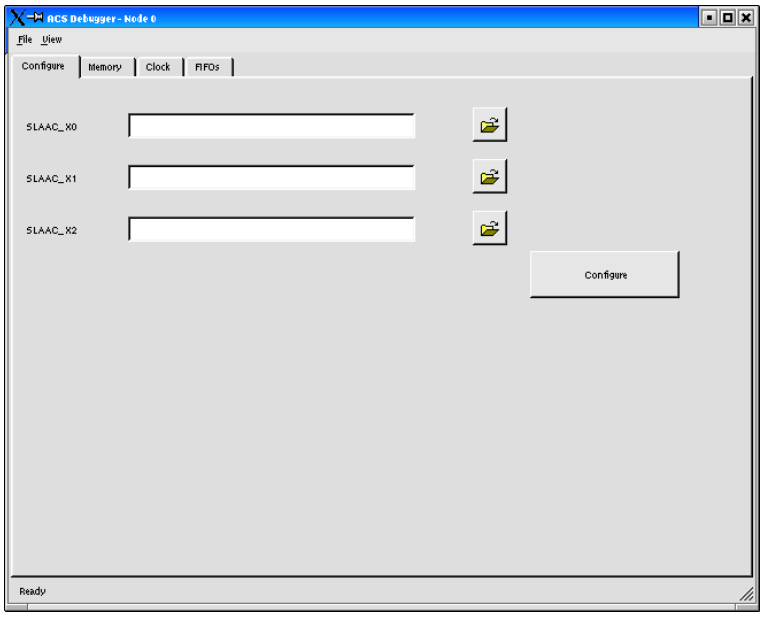

Figure 4.1: Screen Capture of the Configuration Tab

The ACS GUI program generates the appropriate interfaces for the system by querying each board. The node returns information regarding the number and names of devices on the board. For instance, the SLAAC-1V board has three processing elements named X0, X1 and X2. The tab that allows the user to configure the processing elements is able to display the three sets of objects for the node, as shown in Figure 4.1.

The configure tab has a list of processing elements. Each processing element has a text box for entering the configuration file. Near each text box is a button connected to a file dialog. The file dialog is a graphical method for traversing the directory structure and finding the appropriate configuration file. A configuration button, when pressed, signals the system to read the text boxes for the names of the files and configures the appropriate PEs.

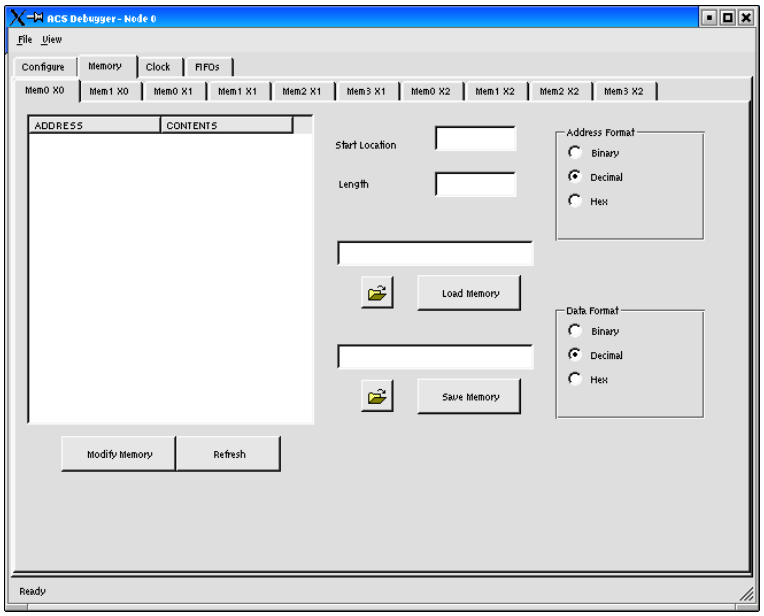

Figure 4.2: Screen Capture of the Memory Tab

The memory tab, shown in Figure 4.2, contains a set of tabs, one for each memory on the board. Text boxes are provided for specifying the starting address and the number of locations the user wishes to read or write. The user may either transfer data to/from a table

residing in the tab or a file. The data may be represented in binary, decimal, or hexadecimal format by choosing a radio button from the button group.

All values in the memory tab are validated using an ACS_Validator. The ACS_Validator class is derived from the QValidator class. The ACS_Validator reimplements the fixup and validate functions. The validate function is called every time a key is pressed within the field. This function can limit which keystrokes modify the text field. For instance, if the memory is in a binary form, the only valid keystrokes are 0's and 1's. The fixup command is called when focus leaves the text field, implying that the text is in its final form. Fixup modifies the value in the text field in order to make it valid. For instance, if the memory is 16-bits wide and 17 bits of data are entered into the field, the fixup function can modify it to a valid 16-bit value, either by cropping the last digit or by saturating it to the maximum value.

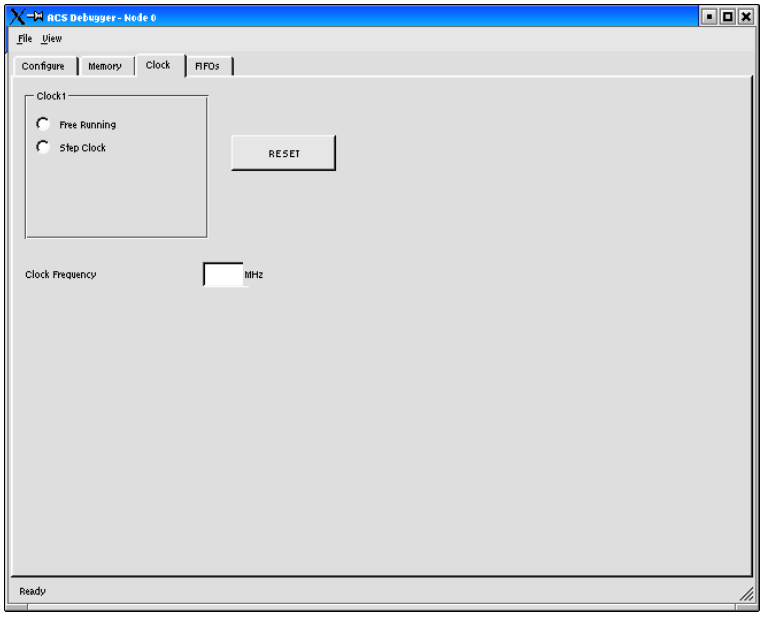

Figure 4.3: Screen Capture of the Clock Tab

The clock tab, shown in Figure 4.3 contains objects for modifying and controlling the clocks

on the FPGA. The clock may be engaged in free-running or stepping mode. If the clock is in free-running mode, a toggle button allows the user to start/stop the clock. If the clock is in stepping mode, the button transforms into a push button that steps the clock each time the button is pressed. The clock tab also contains a button for reseting the board and a text field for modifying the clock frequency.

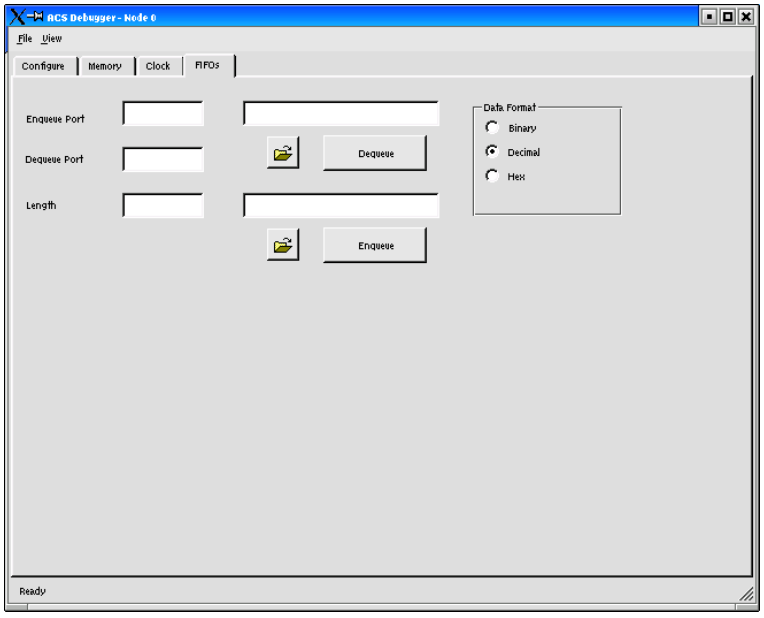

Figure 4.4: Screen Capture of the FIFO Tab

The FIFO tab, shown in Figure 4.4, is used to interface with the streaming data functions of the FPGA board. In the FIFO tab, the user can specify the ports for enqueueing and dequeueing in text boxes. The data for the stream is loaded or saved in a file given by the user in a text field or file dialog. Enqueue and dequeue buttons are used to commit parameters and call the underlying ACS API functions.

The objects in the ACS graphical interface are seamlessly integrated by the Qt tool kit. Qt uses signals and slots to pass information within the program. A function can emit a signal that will traverse the class hierarchy and be caught by all slots that use the signal. Qt

programs are first preprocessed to find all the signals and slots using a meta object compiler. The final application results from compiling all of the meta objects together with the original code. Signals and slots are used in the ACS GUI for such things as sending signals to the status bar in order to display information about the success of ACS API functions.

The ACS graphical interface is a powerful tool in debugging adaptive computing systems. The user can control all functions of the board without learning any of the function calls of the ACS API. While controlling FPGA boards using the ACS graphical interface is slower than that of a program created using C or C++, and even slower than using the textual interface, using the ACS GUI allows users to control distributed adaptive computing systems without any knowledge of the underlying APIs.

Chapter 5

FPGA Pattern Matching Implementation

The following section describes the runtime reconfigurable hardware implementation of the Smith-Waterman [4] genetic code matching algorithm that was adapted from [32]. The first section gives a top-level overview of the hardware design. The following sections describe the parts within the design, namely the processing element, the state machine, the up/down counter, and the data control.

5.1 Overview

The FPGA implementation of the Smith-Waterman algorithm consists of four parts, the processing elements, the state machine, the up/down counter, and data control. The processing element computes the matrix value for each character in the query string. A state machine follows the processing element to convert the output of the processing elements into

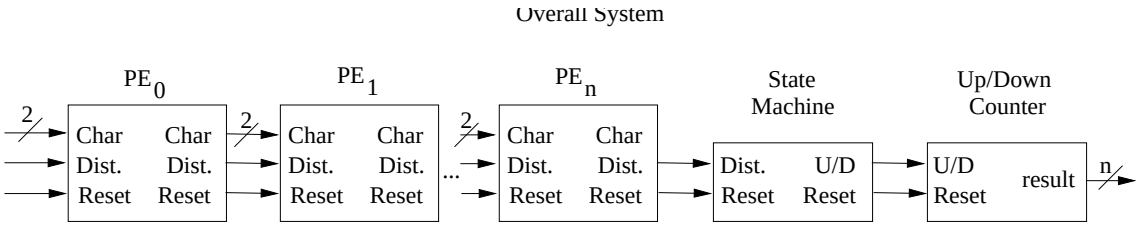

Figure 5.1: Block Diagram of the FPGA Implementation

an up/down signal. The up/down signal connects to an up/down counter that computes the final edit distance value. The method for receiving and sending data through the system is the final part of the implementation. Figure 5.1, adapted from [32], shows the overall system of the implementation.

An important objective in the design for the hardware was to maximize the number of processing elements that fit on an FPGA. If a query sequence cannot fit on one board, it must span either boards or configurations. Nucleotides are represented by a two-bit value as shown in Table 5.1. Insertions or deletions have a penalty of one and substitutions have a penalty of two. Using these penalty values reduces the amount of data that needs to be stored in the PE and sent to the next PE. Because the insertion and deletion penalty is one, the value in the matrix cell a differs from values in cells b and c by exactly one. The d cell differs from the a cell by either zero or two.

A	T	G	C
00	01	10	11

Table 5.1: 2-bit Nucleotide Representation

Intermediate edit distance values are computed and stored in each PE every clock cycle. Because adjacent cells vary by exactly one, the least significant bit can be inferred. The intermediate edit distances are computed modulo four, reducing storage space required for

the matrix value in the processing elements. Because successive edit distance values vary by exactly one, no information is lost if the values are stored modulo four.

5.2 PE Description

The hardware implementation of the Smith-Waterman algorithm has processing elements that compute the values in each column of the edit distance matrix. The query character that is used to compute the column is folded into the hardware logic. Each gene matching PE fits within three slices, or less than one Virtex II CLB.

This implementation of the sequence comparison algorithm uses only local signals. Each processing element requires only the edit distance from the previous column. All of the components in this design are reset synchronously. The database sequence is streamed through the PE, resulting in a single matrix computation each clock cycle per PE.

The block diagram of the inside of a processing element is shown in Figure 5.2, adapted from [32]. The boxes on the right represent flip-flops. The boxes on the left represent LUTs and inside each LUT is the function the LUT generates. The synchronous reset initializes the flip-flops to appropriate values for each processing element. As seen from this diagram, the processing element fits within three slices. The fourth slice of the CLB is left empty for future improvements.

5.3 State Machine

The final processing element sends its local edit distance value into a state machine. The state machine, in combination with the up/down counter, is responsible for converting the

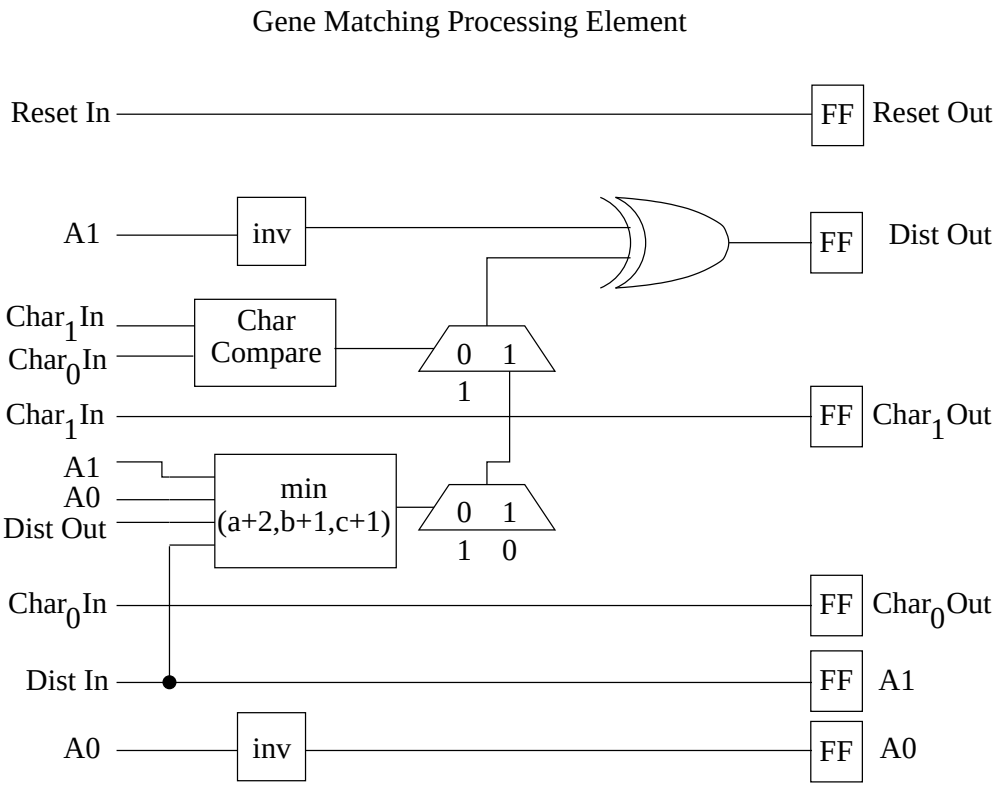

Figure 5.2: Diagram of the Inside of a Processing Element

edit distance from a two-bit, modulo four representation into the full edit distance value. Figure 5.3 shows the state machine used for this gene matching implementation.

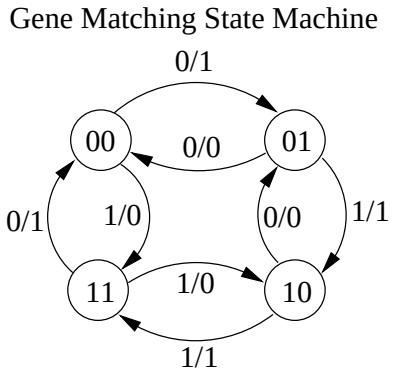

Figure 5.3: Gene Matching State Machine

As shown in Figure 5.3, the state machine has four states. Each state represents the four possible intermediate edit distance values. The initial state depends on the size of the query sequence. The edit distance from the processing elements determines the next state of the system. Only the upper bit of the edit distance is required because the lower bit is inferred. The outer loop in the figure sends an up signal while the inside loop sends the down signal. The up/down signal that is generated is passed to the next stage, the counter.

5.4 Up/Down Counter

At the end of the circuit is an up/down counter that calculates the final edit distance value. The counter receives an up/down signal from the state machine and outputs the 16-bit edit distance and four reset signals. The up/down counter is composed of four 4-bit pipelined counters. The reset signals that are output represent the reset for each of the four stages

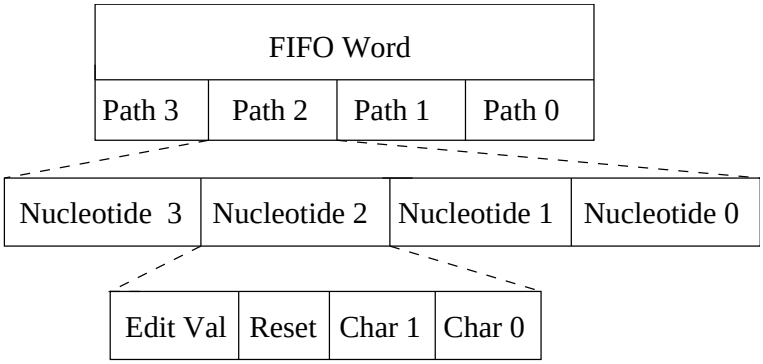

Figure 5.4: FIFO Word

of the pipeline. In order to view the final edit distance value, the signals output from the counter must be captured in four successive clock cycles, starting with the least significant bits.

5.5 Data Control

In order to use the resources of the FPGA efficiently, knowledge of the speed of the devices was required. Any communication between the FPGA board and the host must travel through the PCI bus. An analysis of the design shows that the goal of making the user clock on the FPGA run approximately four times faster than the PCI bus was met. The 64-bit FIFO word, shown in Figure 5.4, is divided into four parts, where each 16-bit part is read into the circuit each clock cycle. The 16-bit part is further divided into four 4-bit nucleotides. Each nucleotide is input into four paths that reside on the FPGA.

The clock used by the system is gated by the FIFO input empty flag, representing no valid data available, and the FIFO output full flag, representing no available space to pass data. FIFO words are written to and read from the host every fourth clock cycle. Due to the four

to one ratio in clock frequencies, the algorithm cannot compute final edit distance values for genes with less than four nucleotides. If a query sequence cannot fit on one FPGA, an intermediate configuration is required. Intermediate configuration files reconstruct the FIFO word in the same fashion as the FIFO word that was input to the board. Final configuration files use all 16 bits dedicated to their path to output the 16-bit edit distance after each database sequence is completely passed through the system.

A second method for transferring the database to the board was implemented using the FPGA board's memories. The host writes to the memories with the database to be streamed through the system and triggers the FPGA by enqueueing a single value to the board. The FPGA then progresses out of its waiting state and streams the data from the memories and through the processing elements. The resulting edit distance values are then written back into memory. After streaming the entire contents of memory, the FPGA enqueues a single value letting the host know that the memories are ready to be read.

Chapter 6

ACS System Implementation

6.1 Overview

One problem that arises in the the solution to the gene-matching problem is that the query sequence embedded in the FPGA design may not fit within a single FPGA configuration. Two solutions to the problem are presented in this chapter. The first system solves the problem by spreading the search sequence across multiple boards and streaming the database sequentially through the boards. The second system uses a single board for each search sequence and reconfigures the board after passing the database through each time. This system scales by distributing the database search over multiple boards.

Before the system can be used, the configuration files must be generated. The hardware described in Chapter 5 embeds the query sequence into the hardware configuration. Each new sequence requires a new configuration to be generated. Figure 6.1 shows the flow of creating new logic core configurations given the query gene sequence and an ASCII representation of the generic configuration design in the Xilinx Design Language (XDL). The query sequence

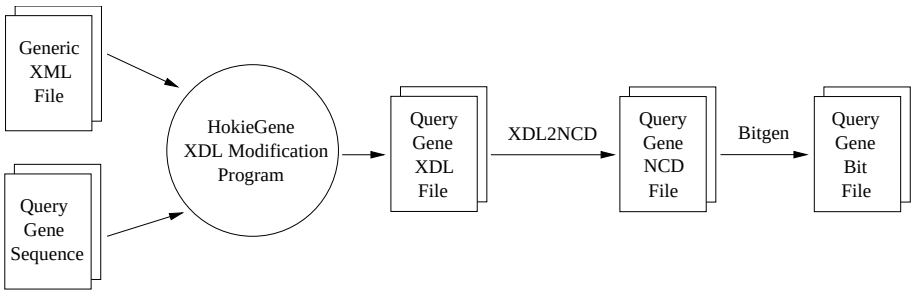

Figure 6.1: Customized Core Generation Flow

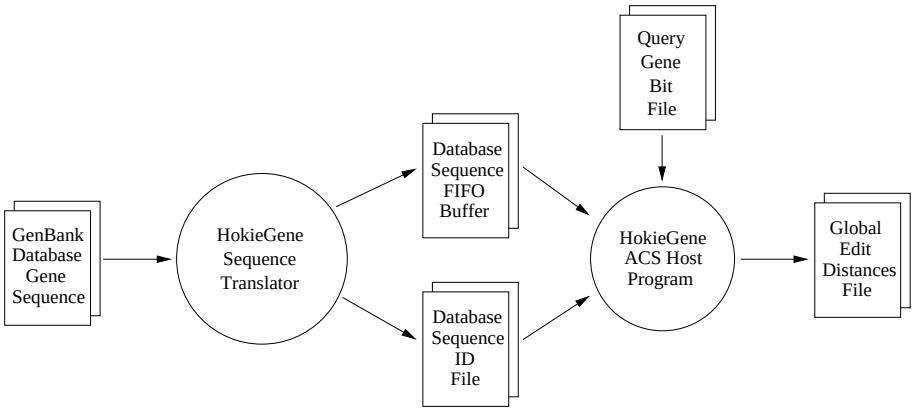

Figure 6.2: Program Flow

XDL file is translated into the native description format (NCD) and then completes the regular Xilinx flow for bitfile generation.

In addition to generating FPGA configuration bit files using the query sequence, the gene sequence database must also be translated into a usable format for the ACS host program. Figure 6.2 shows the flow of the gene sequences in the overall software system. Gene sequence databases can be obtained from one of several gene repositories. The nucleotide sequences used in this thesis were obtained from GenBank [2]. The GenBank representation of the genes is first transformed into the FIFO word format, as shown in in Figure 5.4. This

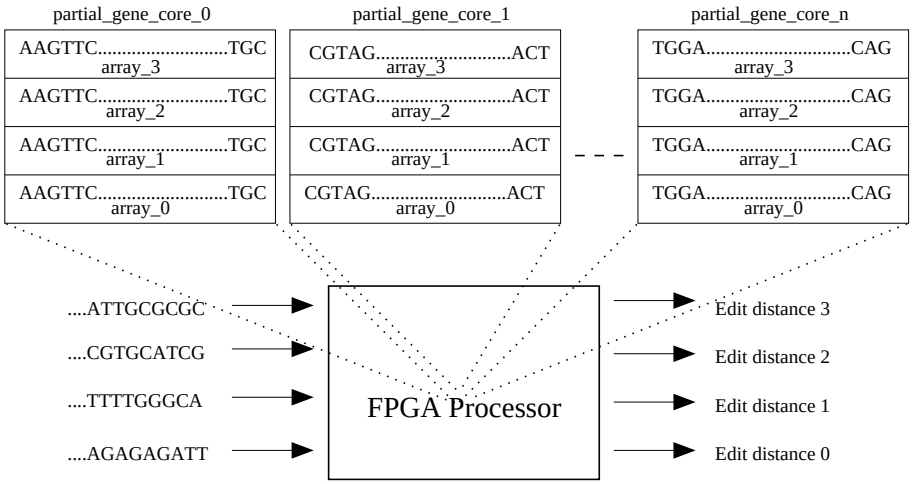

Figure 6.3: Architecture of a Node in the Streaming System

transformation removes all of the gene information that is stored with the sequence in the database. A second file that contains the identification values for each sequence is also generated. These files, along with the query sequence configurations, are input to the ACS host program and the final edit distances of the SW algorithm are computed.

6.2 Streamed System

The streamed system solution spreads a single genetic sequence across multiple boards. The database is then passed through each FPGA board in the system. The final edit distance value is read from the final FPGA board. Figure 6.3 shows a diagram of the distribution of query sequence configurations across the FPGAs in the system.

Once the data is formatted properly, the nucleotide sequences are enqueued into the system through port 0. From the host program, the data is transferred into the *dummy node*. The control process thread on the host machine dequeues the data from the dummy node and

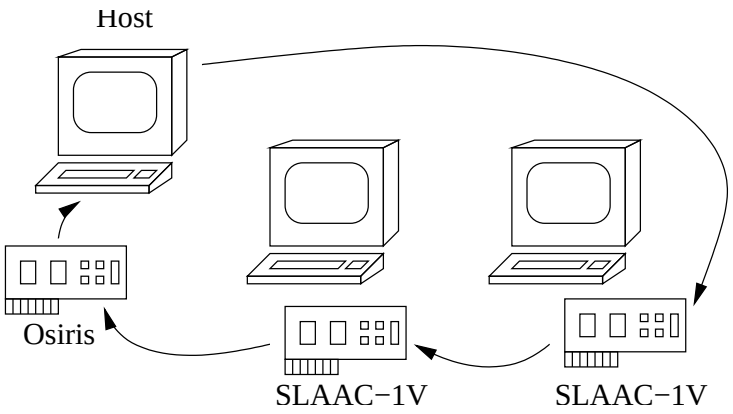

Figure 6.4: Streaming System

places it into the channel buffer. The control process periodically attempts to flush the data in the channel buffer by enqueuing the data into the first node.

The first node processes the data and outputs an intermediate data stream. The data is dequeued from the node by the control process and sent to the next node. This process continues until the data finishes traversing all of the nodes. At the final node, only the edit distance for the gene sequence is output. The final channel dequeues the edit distance and passes it back to the dummy node. The host program can then dequeue the edit distances from port 1 and present the data to the user.

Figure 6.4 shows the three-node implementation of the streaming system. The host machine contains an Osiris Board with the final configuration. The two other computers are accelerated with SLAAC-1V boards. Separate implementations using two and three nodes were developed and tested. The results from the systems are shown in Chapter 7.

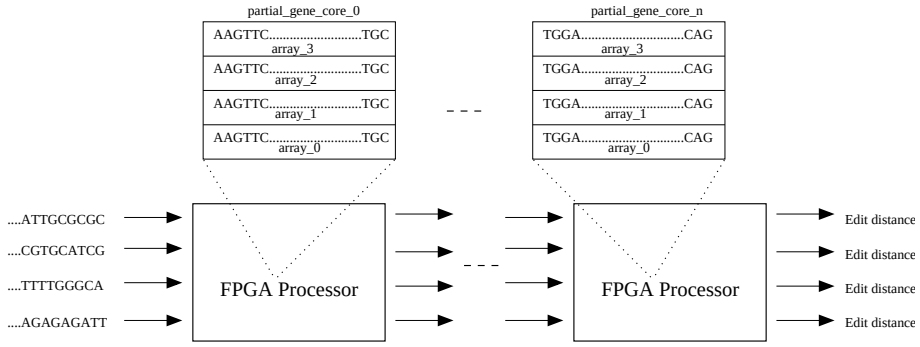

Figure 6.5: Architecture of the Distributed System

6.3 Distributed System

The distributed system solution makes use of multiple boards by streaming data through them independently. In the distributed system, the genetic search sequences are spread across configurations instead of FPGA boards. Depending on the length of the query sequence to be searched, a single FPGA board may need to be reconfigured multiple times. Figure 6.5 shows the distribution of configuration files in a single FPGA for the distributed system.

Once the database of genetic sequences is formatted properly as mentioned in Section 6.2, the host program broadcasts the database sequence to every node in the system. Because each node has an intermediate configuration file, the host program must dequeue and store the intermediate values. Once the entire database has been streamed through the system, the nodes can be configured with the next query sequence core.

The process of streaming data through the nodes, storing the intermediate values, and reconfiguring the board must be repeated until the search sequence is exhausted. When the final configuration file is loaded, the host process dequeues only the final edit distance values and then presents the data to the user.

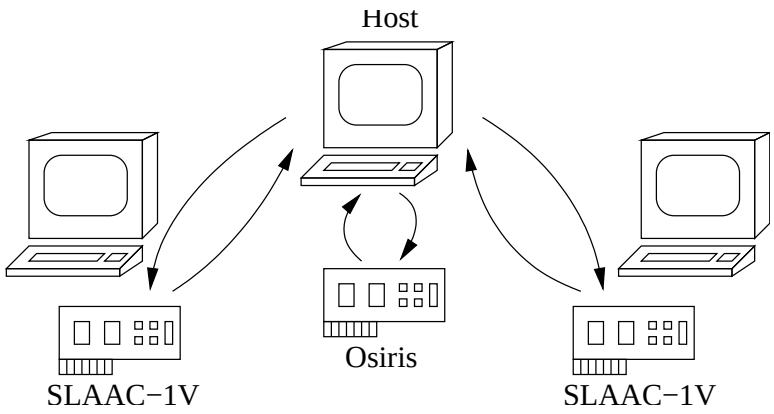

Figure 6.6: Distributed System

Because the host machine can become a bottleneck in the system, the fastest computer was selected for running the host program. The host computer, accelerated with an Osiris board, is networked to two remote machines, each outfitted with a SLAAC-1V. Figure 6.6 shows a diagram of the distributed system.

Chapter 7

Analysis and Results

This chapter presents the results for the genetic sequence matching implementation using heterogeneous distributed adaptive computing systems. The first section compares the results of using the ACS debugger compared to an ACS host program and discusses the use of the debugger in developing an ACS application. The second section discusses the area requirements for the FPGA implementation and the benefits of using a run-time reconfigurable approach. The following section discusses the theoretical results of this implementation and compares those results with other commercial and academic sequence matching systems. The next section analyzes the results of a single node application and compares using local versus using remote nodes and using the ACS API versus using the native SLAAC API. The final two sections show results for two heterogeneous distributed adaptive computing systems and describes the tradeoffs in each system's design.

7.1 ACS Debugger

The ACS textual debugger allows the user to run and debug applications using commands at a prompt or by executing a script. The textual debugger was created using an interpreted language, TCL. This section compares using the textual debugger versus using a standard C++ system program. While the debugger is a useful tool for stepping through applications, it lacks the performance required to make it an efficient method of running a final design.

Table 7.1 shows a comparison of the results using the textual debugger for the SLAAC-1V and the Osiris boards. The entries in the row marked *Configuration Time* represent the time in seconds that it takes to configure all three PEs on the SLAAC-1V or the XP user FPGA in the case of the Osiris. The memory transfer rates show the achieved throughput of a full memory write followed by a full memory read of a single ZBT RAM on each of the boards. The FIFO transfer rates reflect the throughput of a pass-through FIFO in each of the FPGA boards.

	SLAAC-1V		Osiris	
	Host Program	Textual Interface	Host Program	Textual Interface
Configuration Time	1.52s	1.634s	.610s	.670s
Memory Transfer Rate	6.58MB/s	.288MB/s	72MB/s	1.75MB/s
FIFO Transfer Rate	.935MB/s	.036MB/s	111MB/s	.923MB/s

Table 7.1: Results for the Textual Interface

The textual debugger's performance is significantly less than that of an ACS host program for two major reasons. First, the textual debugger is an interpreted language and requires that each command be translated into the native machine language. The second reason for the resulting performance deficiencies is due to the overhead in setting up each function call.

For instance, for each enqueue call, a buffer to hold the data must be created and loaded from the argument string and all of the parameters must be converted from to the appropriate data format. But regardless of the performance results, the textual debugger finds its niche by being able to step through commands without any recompiling. The result is a simple to use tool for debugging ACS applications.

7.2 Area

This section analyzes the area requirements for the configuration described in Chapter 5. Included in this section is a theoretical discussion of the potential area usage on the SLAAC-1V and Osiris boards, as well as an analysis of the configurations developed during the testing of the gene matching implementation. The area limitation of the FPGA is an important metric when considering gene matching. Because genetic sequences may contain a large number of nucleotides, one goal of the hardware design is to maximize the number of nucleotides that can be embedded in a configuration, thus reducing the number of boards/cofigurations a query sequence must span.

The state machine, up/down counter, and data management circuitry use a negligible amount of area on the FPGA. The combination of these three parts of the design requires less than 100 Slices or .3% of a Virtex II 6000 device. The bulk of the configuration is composed of the array of gene processing elements. The gene matching processing elements, therefore, are the portions of interest when considering area.

Each processing element in the hardware configuration represents a single nucleotide in the query sequence. Because the nucleotide is folded into the circuitry, the size of each processing element is very small; each processing element requires only three Virtex slices. Because the value of the nucleotide is embedded into the circuitry, the FPGA must be reconfigured

for each new query. The overhead of reconfiguration is traded for the ability to fit many processing elements in a single configuration. A non-reconfigurable implementation would require five Virtex slices, effectively reducing the number of processing elements on an FPGA by three fifths.

The SLAAC-1V has three Virtex 1000 parts that each have 6,144 CLBs. Each Virtex CLB contains two slices. Because each nucleotide in the query sequence, translated into hardware, requires three slices, the X1 and X2 PE can fit approximately four thousand query nucleotides. The X0 PE can fit approximately half as many query nucleotides on the device because the X0 PE already contains the interface circuitry for the FPGA board. By streaming data through all three processing elements, a SLAAC-1V has the ability to compute the edit distance of a 10,000 nucleotide sequence without reconfiguration. An implemented X1 design was developed with four paths of 700 nucleotides. After mapping the configuration to the Virtex chip, the entire design required 10,416 slices, or 84% of the X1 device.

The Osiris board has one user FPGA, a Virtex II 6000 part. The user FPGA device has 8448 CLBs. Each Virtex II CLB contains four slices. Therefore, the Osiris board can have a configuration of approximately 7000 gene-matching processing elements. The implementation synthesized for this thesis has four paths each with 1700 processing elements for a total of 6,800 nucleotides. The mapped configuration resulted in 87% of the board, 29,338 slices, being used.

7.3 FPGA Throughput

The primary goal of the implementation was to reduce the time required to compare genetic sequences using the Smith-Waterman algorithm. This was accomplished by maximizing the number of query sequences on an FPGA, and by maximizing the throughput of the

system. These two metrics are combined to create the standard commercial metric of Smith-Waterman cell updates per second.

One reason the Smith-Waterman algorithm was chosen to be implemented on FPGA technology was that the SW algorithm uses only local signals. Because all signals are local, the FPGA board can be run at extremely high frequencies. During development, it became clear that the PCI bridge would be the bottleneck in determining the maximum frequency of the circuit.

After synthesizing the Osiris implementation, the circuit was determined, by the synthesizing software, to perform correctly when the user clock is less than 180 MHz. The I/O clock on the board is limited to 45MHz. Because each FIFO word has four paths of four characters, the circuit efficiently uses all of the PCI bandwidth. Theoretically, if the FIFO is kept full, the FPGA circuit can achieve 1.26 trillion cell updates per second.

The SLAAC-1V implementation uses a 32-bit 33MHz PCI bus. By following the same logic of the Osiris implementation, the SLAAC-1V implementation is capable of performing at approximately one quarter the speed of the Osiris board. The circuit was designed such that the user clock could run at 66 MHz. If the FIFO is kept full, the SLAAC-1V board can achieve 462 billion SW matrix updates per second.

Table 7.2 compares the theoretical results of the design presented in this thesis to previous implementations. Both Paracel and TimeLogic are commercial products. Paracel accelerates their system using ASICs, while the remaining designs use FPGAs. Splash-2 is an early FPGA board that was used to implement a one time reconfigurable solution to the gene matching problem. The use of runtime reconfiguration for optimization of the processing elements greatly increases the number of gene processing elements that fit on a device. However, at this time, it takes approximately nine minutes to convert an XDL file into a bitfile configuration using the Xilinx foundation tools. This time is quite large considering

the process is simply converting the file from one format to another without requiring any routing or placing. The bitfile creation can be performed on multiple machines to ensure that FPGA boards do not need to wait for the configurations.

	Year	Processors per Device	Devices per Node	Updates per second
Pentium III - 1.4GHz	2002	1	1	82M
Paracel(ASIC)	2001	192	144	276B
TimeLogic (FPGA)	2000	6	160	25B
Splash 2 (XC4010)	1993	14	272	43B
SLAAC1V (XCV1000)	2002	2800	2.5	462B[1]
Osiris (XC2V6000)	2002	7000	1	1.260T[1]
SLAAC1V(acheived)	2002	2800	2.5	23B
Osiris(acheived)	2002	7000	1	389B

Table 7.2: Performance and Hardware Size for Various Systems

7.4 Single Node Throughput

This section discusses the performance of the gene matching system using a single node. The Osiris and SLAAC-1V single node designs were implemented using the SLAAC API, the native API designed for the board, and the ACS API. The ACS API performance results are shown for the single local node as well as the single remote node system. Results are shown for two methods of transferring information to the board, through FIFOs and through memory transfers.

[1]Values represent theoretical results computed using maximum achievable clock speed and area.

$$t_{\text{db}} = \left\lceil \frac{n}{N} \right\rceil * t_{config} + (n+m) * max(t_{clk}, t_{fifo}(D)) \tag{7.1}$$

$$t_{\text{db}} = \left\lceil \frac{n}{N} \right\rceil * t_{config} + (n+m) * (t_{clk} + t_{mem}(D)) \tag{7.2}$$

$$t_{\text{db}} = \left\lceil \frac{n}{N} \right\rceil * t_{config} + (n+m) * max(t_{clk}, t_{fifo}(D), \psi * t_{net}(P)) \tag{7.3}$$

$$t_{\text{db}} = \left\lceil \frac{n}{N} \right\rceil * t_{config} + (n+m) * (t_{clk} + t_{mem}(D) + \psi * t_{net}(P)) \tag{7.4}$$

where,

t_{db}	time to compute edit distance of a data base,
t_{config}	configuration time,
t_{clk}	FPGA clk period,
t_{fifo}	time per nucleotide to stream D bytes block of data to/from the FPGA,
t_{mem}	time per nucleotide to send D bytes block of data to/from the FPGA,
t_{net}	round-trip time per nucleotide of a P-sized packet across the network ,
ψ	remote function overhead factor,
N	maximum number of nucleotides per FPGA,
P	MPI packet size transferred across the network,
D	block size of data transferred to the board,
n	size of the query sequence, and
m	size of the database sequences.

The time required to compare a query sequence to a database of sequences in a single local node is calculated by Equation 7.1 and Equation 7.2. When remote nodes are used, a network communication factor is added, as shown in Equation 7.3 and Equation 7.4. The network

communication term is multiplied by a ψ factor in order to compensate for the overhead in packing and unpacking parameters for the remote function calls. If FIFOs are used to transfer database sequences to the board, data transfer can be overlapped with the FPGA computation. In contrast, the memory transfer approach does not overlap computation and I/O. The time to load a configuration on a remote system is approximately equal to the time it takes to reconfigure on a local node because only a small amount of data, the configuration file names, must be passed across the network. Each of the remaining terms are multiplied by the size of the database sequence plus the size of the query sequence.

	SLAAC-1V	Osiris
t_{config}	1.52s	.610s
t_{clk}	1.52e-8s/B	5e-9s/B
t_{fifo}	1.07e-6s/B	9.01e-9s/B
t_{mem}	1.53e-7s/B	1.48e-8s/B
$t_{\text{net}}(fifo)$	1.82e-7s/B	2e-6s/B
$t_{\text{net}}(memory)$	1.82e-7s/B	1.82e-7s/B
$\psi(fifo)$	2.20	1.03
$\psi(memory)$	1.10	3.83

Table 7.3: Values for the Single Node Model

Tablel 7.3 shows the values obtained for the single node models. The configuration for each board was timed to produce the configuration time. The clock time is the speed of the user FPGA clock set by the host program. By running a pass-through configuration on the FPGA, the time to stream data through the FIFOs was found. The memory time was obtained by sending a block of data the size of the ZBT memory to the board and reading it back. The network time was obtained by running an MPI program that sends and receives data from a single remote node. The program calculates the network throughput for various

sized packets. The values shown in the table represent the network throughput for sending memory-sized packets and FIFO-sized packets. The ψ factor is computed using the time required to stream data and read/write memory on a remote node.

The results for this thesis were obtained using an Osiris board in a machine that contains a Pentium III 1.4 GHz processor with 512MB of RAM. The results for the single node SLAAC-1V implementation were shown using a Pentium II 300MHz machine with 256MB of RAM. A second SLAAC-1V, used in the multiple node systems, is in a Pentium II 333MHz machine with 384MB of RAM. Each SLAAC-1V accelerated machine is networked together by a 10/100 switched ethernet connection and the Osiris board is connected through a gigabit network.

		SLAAC-1V			Osiris		
	Database	SLAAC	ACS		SLAAC	ACS	
	Size	Local	Local	Remote	Local	Local	Remote
GBUNA	.159MB	.115s	.183s	.743s	.002s	.002s	.0319s
GBPHG	3.27MB	2.34s	3.75s	15.0s	.030s	.031s	.279s
GB14	11.2MB	7.96s	12.7s	51.2s	.103s	.101s	.936s
GBGSS4	19.2MB	13.9s	21.4s	88.1s	.177s	.173s	1.60s
GB36	28.8MB	20.4s	31.7s	133s	.267s	.259s	2.41s
GB48	38.4MB	27.3s	41.1s	178s	.355s	.345s	3.22s
GBPRI20	47.5MB	33.7s	50.9s	219s	.438s	.425s	3.97s
Throughput		1.4MB/s	.94MB/s	.22MB/s	108MB/s	111MB/s	12MB/s

Table 7.4: FIFO Results for the Single Node

The ACS API and SLAAC API host programs are very similar. The main differences lie in the construction and initialization of the systems. The exact same software algorithm

is used in each version of the program resulting in a good direct comparison of the results. Table 7.4 shows the results when using the FIFOs to transmit data to the board. The sample query gene was compared to four different databases of genBank genes, GBUNA, GBPHG, GBGSS4, and GBPRI20. In addition to the genBank genes, three additional databases GB14, GB36, and GB48 were used. These three gene databases are subsets of the GBPRI20 database created to show additional points of comparison. The size of the database sequences after the transformation into FIFO/Memory word format is shown in Table 7.4. The system throughput represents the slope of lines shown in Figure 7.1 and Figure 7.2. The system throughput of the Osiris board is limited by the PCI bus speed in the local case and limited by the network speed in the remote case. Baseline tests for the network were experimentally shown to be approximately 6MB/s for the SLAAC-1V and 15MB/s for the Osiris in the best case (transferring large packets of data) and only about 1MB/s in the worst case (small packets of data). These values were obtained by using a MPI program that varies the send/receive sizes for node to node communication. Because the SLAAC-1V driver only allows 250 word FIFO transfers, the system becomes inefficient due to the overhead of setting up each transfer.

Figure 7.3 and Figure 7.4 shows the time versus database size for the SLAAC-1V. The slope of the lines represent the system throughput and the y-intercept represents the time required to configure the boards. The graphs show the time difference of running the system using the ACS API and SLAAC API. The theoretical results, calculated using the local and remote node models, are also shown.

Table 7.5 shows the values for using memory to transfer data to the board. The memory transfer sizes are limited by the size of the memories on the board. The SLAAC-1V can transfer 256K words to the board for each iteration. The Osiris board has 512K word memories and both boards memories are effectively 32 bits wide. The performance results for the SLAAC-1V are better than the FIFO implementation because the transfer sizes are

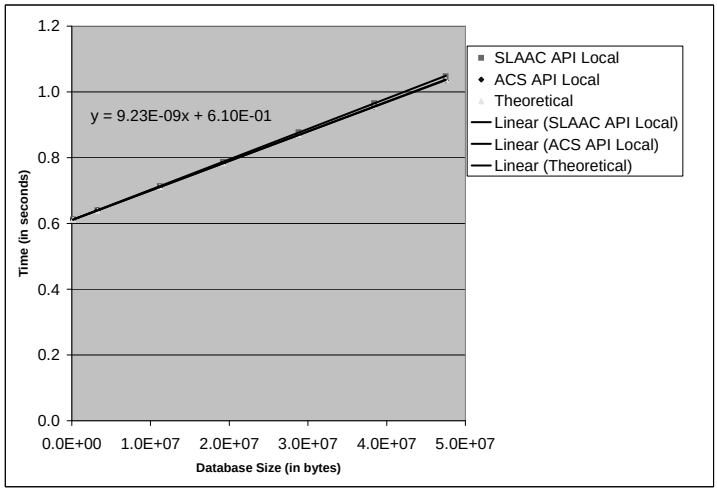

Figure 7.1: Single Local Osiris Board using FIFOs

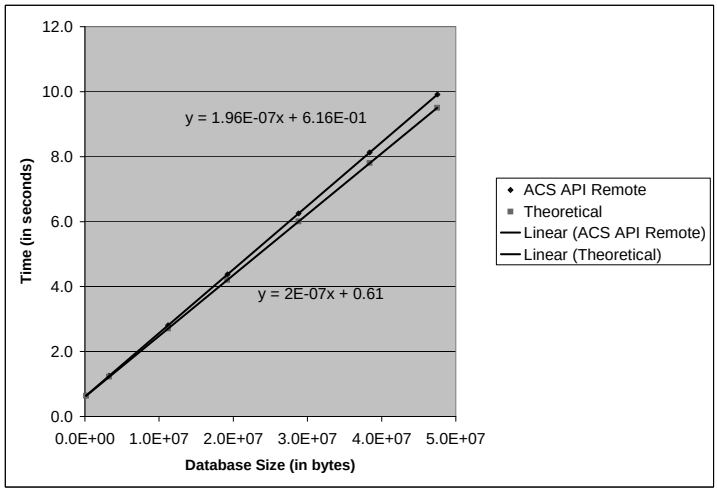

Figure 7.2: Single Remote Osiris Board using FIFOs

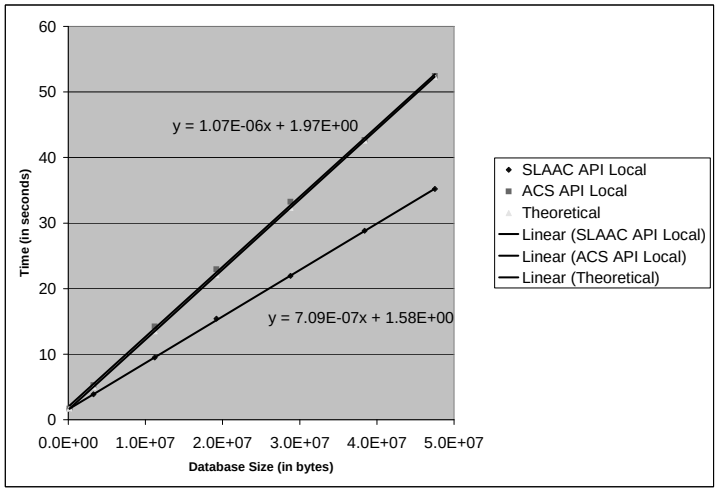

Figure 7.3: Single Local SLAAC-1V Board using FIFOs

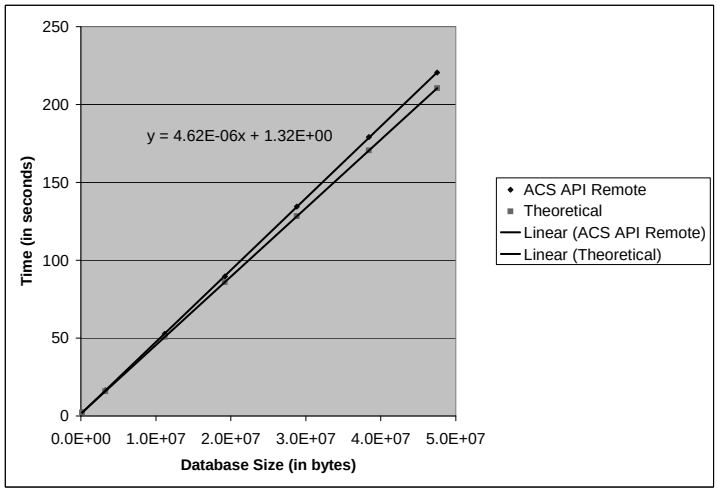

Figure 7.4: Single Remote SLAAC-1V Board using FIFOs

Board		SLAAC-1V			Osiris		
API	Database	SLAAC	ACS		SLAAC	ACS	
	Size	Local	Local	Remote	Local	Local	Remote
GBUNA	.159MB	.122s	.129s	.347s	.026s	.004s	.211s
GBPHG	3.27MB	.611s	.648s	1.72s	.051s	.060s	3.11s
GB14	11.2MB	3.06s	1.94s	5.14s	.153s	.200s	10.4s
GBGSS4	19.2MB	7.34s	3.24s	8.62s	.254s	.336s	17.7s
GB36	28.8MB	1.84s	4.80s	12.7s	.380s	.506s	26.8s
GB48	38.4MB	4.53s	6.35s	16.8s	.505s	.672s	35.3s
GBPRI20	47.5MB	6.00s	7.78s	20.6s	.608s	.834s	43.3s
Throughput		6.5MB/s	6.2MB/s	2.3MB/s	80MB/s	57MB/s	1.1MB/s

Table 7.5: Memory Results for the Single Node

larger, reducing the transaction setup overhead. Similarly, the Osiris board's performance decreases because the data transfer sizes from the host processor to the board are decreased.

Figure 7.5 and Figure 7.6 show the sequence comparsion time versus the size of the database for the memory transfer method of communicating with the Osiris board. The slope of the line corresponds to the throughput of the system. The theoretical results shown are calculated using Equation 7.2 and Equation 7.4 for the local and remote systems.

Figure 7.7 and Figure 7.8 show the results of running the sequence comparing algorithm on the SLAAC-1V board. Included in the graphs are the theoretical results using the memory transfer model.

The Osiris board achieves better performance while streaming data than transferring data through memory. Using the FIFOs, the realized Osiris board implementation is capable

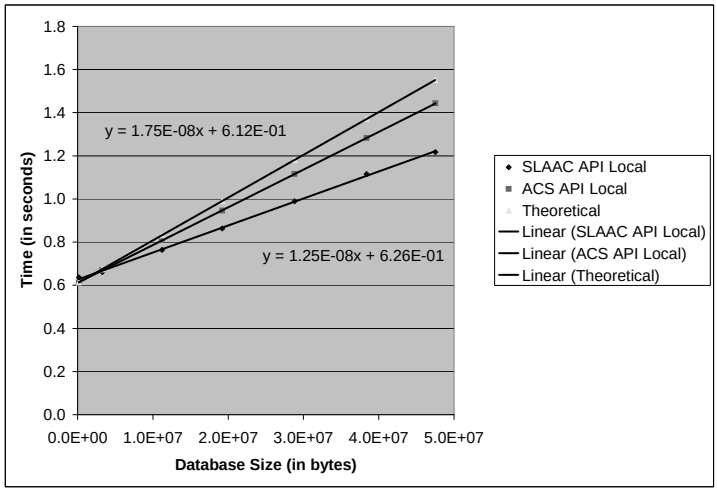

Figure 7.5: Single Local Osiris Board using Memory Transfers

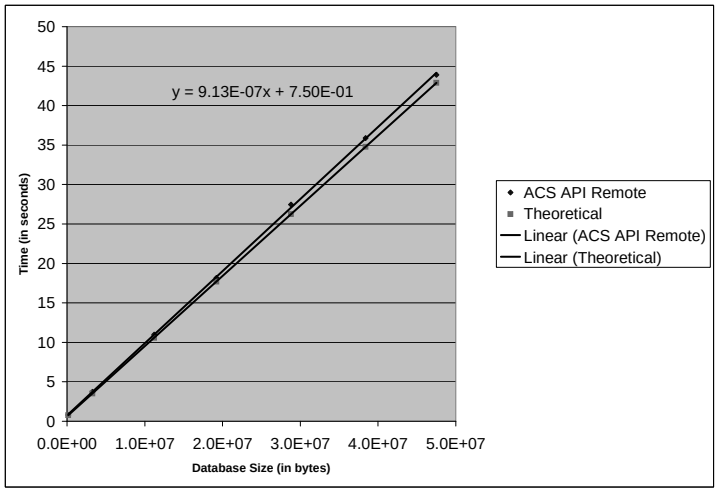

Figure 7.6: Single Remote Osiris Board using Memory Transfers

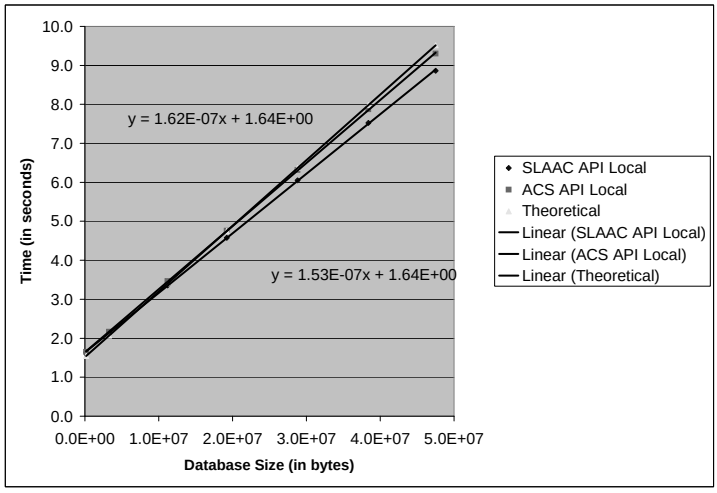

Figure 7.7: Single Local SLAAC-1V Board using Memory Transfers

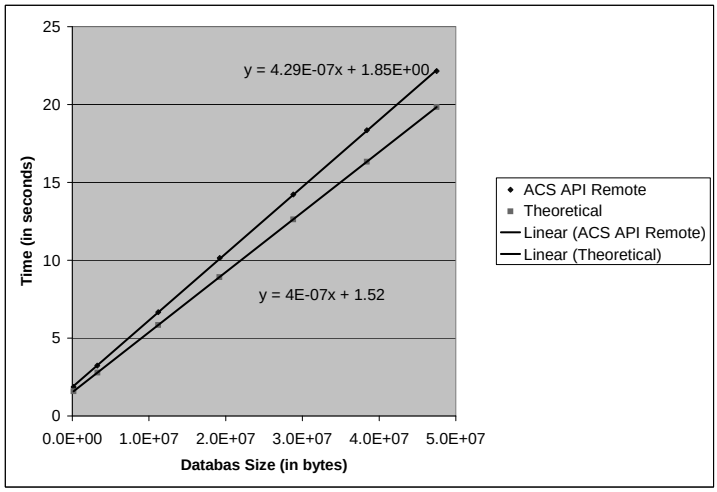

Figure 7.8: Single Remote SLAAC-1V Board using Memory Transfers

of 389 billion Smith-Waterman matrix cell updates per second compared to the theoretical 1.26 trillion cell updates per second. Conversely, the SLAAC-1V board handles memory transfers better than streamed data. Using the memory to transfer database sequences, the SLAAC-1V board is capable of 23 billion Smith-Waterman matrix cell updates per second compared to the theoretical 462 billion.

7.5 Streaming Approach

This section presents the results from the streaming approach to using multiple boards. When a gene sequence to be searched does not fit within a single board's configuration, the gene sequence is spread across multiple boards and the database is streamed through.

	Two Boards	Three Boards
GBUNA	.210s	.738s
GBPHG	4.21s	11.4s
GBGSS4	24.8s	67.1s
GBPRI20	162s	164s
System Throughput	.76MB/s	.59MB/s
CUPS	6.46B	7.97B

Table 7.6: Results for the Streaming Implementation

Table 7.6 presents the results for the streaming systems using the four genBank database sequences. In the streaming approach, the overall performance degrades to the throughput of the slowest node making this system a poor design for heterogeneous nodes. The two-board system uses the Osiris board in the host computer and streams data to the SLAAC-1V on the remote system. The speed of the system is clamped by the speed of the remote SLAAC-1V.

When a second SLAAC-1V board is added to the system, the performance degrades further. The degradation is due to the fact that the second SLAAC-1V system is even slower than the first. Both the 2-board and 3-board systems perform close to the speeds of the slowest node. An additional problem with this system is that the system can only scale to the size of the query sequence. Eighty percent of the sequences in the databases can be compared using a single configuration.

7.6 Distributed Approach

This section presents the results from the distributed approach to comparing DNA sequences. In each system, the host machine contains an Osiris board and the two other machines that contain SLAAC-1V boards have a network connection to the host. The database is streamed separately through each board. When a query sequence does not fit on one board, the output from the first configuration is stored in the host processors memory while the board is configured with the next part of the query sequence.

Table 7.7 shows the results for the distributed system using two and three boards. While the multiple node system has some performance gains, the network congestion limits the overall system performance. The Osiris board is on a separate PCI bus from the network card and allows the system to perform better than one that has a single PCI bus. However, each added node must vie for network bandwidth and since all traffic goes through the single network card, the host machine becomes a bottleneck. As seen in the table, adding the second board does not acheive the same performance gain as adding the first node.

	One Board	Two Boards	Three Boards
GBUNA	289B CUPS	267B CUPS	259B CUPS
GBPHG	379B CUPS	388B CUPS	393B CUPS
GBGSS4	380B CUPS	395B CUPS	397B CUPS
GBPRI20	380B CUPS	397B CUPS	398B CUPS

Table 7.7: Throughput for the Distributed Implementation

Chapter 8

Conclusions

8.1 Summary

This thesis presents the design and implementation of a gene matching algorithm developed on a distributed environment of adaptive computing boards using the ACS API. The development process was streamlined by implementing the system using the ACS API. Development tools such as the ACS textual and graphical interfaces allowed configurations to be debugged quickly and easily.

The sequence comparing hardware configuration was created and the benefits of using a run-time reconfigurable approach were shown. The configuration was optimized to increase the number of processing elements that fit in a single FPGA configuration.

Two designs for distributing FPGA boards in a system were presented and compared. In the streaming approach, the speed of the system is determined by the slowest node. This approach lends itself better to homogeneous systems. The distributed approach avoids the bottleneck of the slowest node in the system, allowing each board to run uninhibited. How-

ever, if the network shares the same bus as the FPGA board, then the overall system will slow down.

In conclusion, the ACS API is a simple programming environment that hides the complexity of a system. The increased usability does not add significant delays when using local nodes. The ACS API creates a uniformity among adaptive computing systems and allows tools to be developed that can be used on several different platforms. The gene matching application was able to be designed and created quickly using the ACS API.

8.2 Future Work

The gene matching systems developed and presented in this thesis compute the global edit distance. The next step is to develop an implementation that can compute and find interesting local edit distances. The local edit distance is a minima in the SW matrix. The local alignment implementation will pair counters with each processing element. Assuming 16-bit counters are used, two to three times fewer processing elements will fit in a local alignment FPGA configuration.

In addition to modifying the design presented in this thesis to compute local alignments, the design can be modified to compute edit distances for proteins as well. Proteins require five bits for each character. Therefore, the protein implementation will require at least two and a half times as much area as the DNA sequence matching implementation.

The gene matching configurations constructed in this thesis were developed statically prior to runtime. The final implementation requires runtime reconfiguration. At the time of this thesis, significant progress has been made to implement the runtime generation of configurations by manipulating a generic XDL file. JBits [33], once completed for Virtex-II parts,

can be used for the runtime reconfiguration environment to further improve the design.

The ACS API performs well when using local nodes. However, a significant cost is incurred each time data must travel across the network. Many of the problems that occurred in increasing the performance across the network were a result of deficiencies in the drivers for the boards. More specifically, the ACS API requires that the FPGA drivers efficiently implement non-blocking functions for streaming data. Efforts have already been made to correct these problems; however, it is important that FPGA board designers recognize the needs of an automated environments such as the ACS API. Network traffic can be reduced, as well as increasing processing power, by expanding the ACS API to make use of the remote machines in the system. In this situation, the control processes resident on remote machines could be used to load, manipulate, and/or store information without contacting the host process. A single directive from the host process would trigger events on the remote systems, thus reducing the network traffic. The sequence comparing implementation described in this thesis could be run on several machines each with their own copy (or portion) of the database sequences.

Bibliography

[1] "The human genome project information page." http://www.genome.gov/, 2002.

[2] "National center for biotechnology information." http://www.ncbi.nlm.nih.gov/, 2002.

[3] "Swiss-prot protein knowledge." http://us.expasy.org/sprot/, 2002.

[4] T. Smith and M. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology*, vol. 147, pp. 195–197, 1981.

[5] C. Ulmer, *Configurable Computing: Practical Use of Field Programmable Gate Arrays*. PhD thesis, Georgia Institute of Technology, Atlanta, GA, USA, January 1999.

[6] Xilinx Inc., *The Programmable Logic Data Book*. San Jose, CA, 1999.

[7] "Slaac project home page." http://www.east.isi.edu/projects/SLAAC/, 2002.

[8] USC Information Sciences Institute, *SLAAC1-V User VHDL Guide*, 0.3.1 ed.

[9] P. B. B. Schott and L. Wang, *Osiris Board Architecture and VHDL Guide*. USC Information Sciences Institute, 1.1.1 ed., May 2002.

[10] X. Inc., *Virtex-II Platform FPGA Handbook*. San Jose, CA, December 2001.

[11] D. L. R. Lipton, "A systolic array for rapid string comparison," in *1985 Chapel Hill Conference on Very Large Scale Integeration* (H. Fuchs, ed.), pp. 363–376, 1985.

[12] M. Sievers, "The biologist's guide to paracel's similarity search algorithms."

[13] "Fasta programs." http://fasta.bioch.virginia.edu/, 2002.

[14] "Ncbi blast home page." http://www.ncbi.nlm.nih.gov/BLAST/, 2002.

[15] "Decypher bioinformatics acceleration solution." http://www.timelogic.com/decypher_intro.html, 2002.

[16] "Compugen." http://www.cgen.com/, 2002.

[17] "Paracel, inc. world wide web site." http://www.paracel.com/, 2002.

[18] "The genematcher2 system datasheet." http://www.paracel.com/products/pdfs/gm2_datasheet.pdf, 2002.

[19] "Snort: The open source intrusion detection system." http://www.snort.org/, 2002.

[20] R. Boyer and J. Moore, "A fast string searching algorithm," *Communications of the ACM*, vol. 20, pp. 762–772, October 1977.

[21] N. K. Ratha, K. Karu, S. Chen, and A. K. Jain, "A real-time matching system for large fingerprint databases," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 18, no. 8, pp. 799–813, 1996.

[22] N. K. Ratha, A. K. Jain, and D. T. Rover, "Fingerprint matching on splash 2," Tech. Rep. MSU-CPS-99-20, Department of Computer Science, Michigan State University, East Lansing, Michigan, April 1999.

[23] L. Rabiner, "A tutorial on hidden markov models and selected applications in speech recognition," in *Proceedings of the IEEE*, vol. 77, February 1989.

[24] R. Hughey, "Massively parallel biosequence analysis," Tech. Rep. UCSC-CRL-93-14, 1993.

[25] O. Gotoh, "An improved algorithm for matching biological sequences," *Jornal of Molecular Biology*, vol. 162, pp. 705–708, 1982.

[26] M. Jones, L. Schraf, J. Scott, C. Twaddle, M. Yaconis, K. Yao, and P. Athanas, "Implementing an API for distributed adaptive computing systems," in *IEEE Symposium on FPGAs for Custom Computing Machines* (K. L. Pocek and J. Arnold, eds.), (Los Alamitos, CA), pp. 222–230, IEEE Computer Society Press, 1999.

[27] K. Yao, "Implementing an application programming interface for distributed adaptive computing systems," Master's thesis, Virginia Polytechnic Institute and State University, Blacksburg, VA, USA, May 2000.

[28] "Extensible markup language (xml)." http://www.w3.org/XML/, 2002.

[29] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, Jack Dongarra, *MPI: The Complete Reference*, 1997.

[30] "Tcldeveloper site." http://www.tcl.tk/, 2002.

[31] "Trolltech: Creators of qt, the cross-platform c++ gui toolkit." http://www.tcl.tk/, 2002.

[32] S. Guccione and E. Keller, "Gene matching using jbits." 2002.

[33] S. Guccione and D. Levi, "XBI: A Java-based interface to FPGA hardware," in *Proc. SPIE Photonics East, J. Schewel (Ed.), SPIE - The International Society for Optical Engineering, Bellingham, WA*, November 1998.

Appendix A

Hokiegene Code

This appendix contains code from the gene sequence comparison implementation described in this thesis. Listing A.1 shows the hardware description of a processing element for the Adenine nucleotide. Listing A.2 shows the system configuration file for the streaming system. Listing A.3 shows the host program code for the streaming system. Listing A.4 shows the system configuration file for the distributed system. The final listing, Listing A.5, contains the host program code for the distributed system.

Listing A.1: Processing Element Hardware Description

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.std_logic_unsigned.all;
4 use ieee. std_logic_arith .all;
5 library slaac;
6 use slaac.seq_pkg.all;
7 entity seq_proc_a0 is
8     port (
```

```
9         clk      : in  std_logic;          -- global clock
10        RSTin    : in  std_logic;          -- synchronous reset signal (input)
11        TChin    : in  std_logic_vector (1 downto 0); -- database DNA (input)
12        EDITin   : in  std_logic;          -- edit distance (input)
13        RSTout   : out std_logic;          -- synchronous reset signal (output)
14        TChout   : out std_logic_vector (1 downto 0); -- database DNA (output)
15        EDITout  : out std_logic           -- edit distance (output)
16        );
17 end seq_proc_a0;
18 architecture initial of seq_proc_a0 is
19        signal A0, A0_n, A1, A1_n, B, C, dout, dout_int, comp1_1, comp1_0,
20                comp0_1, comp0_0, data0_1, z0   : std_logic;
21        component comp1 is                 -- Compares 2 DNA characters
22                port (
23                t1in, t0in, s1in, s0in   : in  std_logic;
24                comp     : out std_logic
25                );
26        end component;
27        component comp2 is                 -- Compares values in cells a, b, and c
28                port (
29                a1, a0, b, c      : in  std_logic;
30                comp     :out std_logic
31                );
32        end component;
33 begin
34        --Slice 3
```

```
35        -- pass the synchronous reset through
36      lrststore  :  dff_s  port map (clk => clk, set => RSTin, Din => RSTin,
37            Dout => RSTout);
38        -- compute cell 'a' + 2 value
39      lA1_n : inv port map (a => A1, b => A1_n);
40        -- send either 'a + 2' or value from comp1_1
41      ldout_int  : xor_2 port map (a => A1_n, b => comp1_1, c => dout_int);
42        -- Flip-Flop for edit distance
43      ldout    :  dff_r  port map (clk => clk, rst => RSTin, Din => dout_int,
44            Dout => dout);
45        -- Send edit distance to next processing element
46      EDITout <= dout;
47
48      --Slice 2
49        -- signal mismatch or send value form comp0_1
50      lcomp11 : MUXCY port map (o => comp1_1, DI => '1', CI => comp0_1,
51            S => comp1_0);
52        -- compare database character (TChin) to query character (sXin)
53      lcomp10 : comp1 port map (t1in => TChin(1), t0in => TChin(0), s1in => '0',
54            s0in => '0', comp => comp1_0);
55        -- pass the database character through
56      lt1out   :  dff_r  port map (clk => clk, rst => RSTin, Din => TChin(1),
57            Dout => TChout(1));
58        -- signal to use values for cell 'a', 'b', or 'c'
59      lcomp01 : MUXCY port map (o => comp0_1, DI => '1', CI => '0',
60            S => comp0_0);
```

```
61          -- compare 'a+1', b, and c to determine minimum value to send
62        lcomp00 : comp2 port map (a1 => A1, a0 => A0, b => dout, c => EDITin,
63                comp => comp0_0);
64          -- pass the database character through
65        lt0out  : dff_r port map (clk => clk, rst => RSTin, Din => TChin(0),
66                Dout => TChout(0));
67
68        --Slice 1
69          -- store cell 'b' for use as cell 'a' in next time step
70        lA1 : dff_r port map (clk => clk, rst => RSTin, Din => EDITin,
71                Dout => A1);
72          -- construct lower bit of cell 'a' ( toggles due to scoring format)
73        lA0_n : inv port map (a => A0, b => A0_n);
74          -- store cell 'b' for use as cell 'a' in next time step
75        lA0 : dff_r port map (clk => clk, rst => RSTin, Din => A0_n, Dout => A0);
76  end initial;
```

Listing A.2: Streaming System XML Configuration File

```
1  <?xml version='1.0' encoding='UTF-8'?>
2  <!DOCTYPE config SYSTEM "/project/acs_api/src/config/acs_conf.dtd">
3  <config>
4  <header>
5     <commtype>MPI</commtype>
6     <version>0.03</version>
7     <author>William J. Worek</author>
8     <desc>Streaming System Configuration</desc>
9  </header>
```

```
<boards>
   <osiris board_id="node0" location="local"  ctrl_proc_dir ="/project/acs_api/bin">
      <PE pe_num="0" prog_file="xp_es.bit"/>
   </osiris>
   <slaac1v board_id="node1" location="xanadu" ctrl_proc_dir="/project/acs_api/bin">
      <PE pe_num="0" prog_file="x0.bit"/>
      <PE pe_num="1" prog_file="x1.bit"/>
      <PE pe_num="2" prog_file="x2.bit"/>
   </slaac1v>
   <slaac1v board_id="node2" location="cowbell" ctrl_proc_dir ="/project/acs_api/bin">
      <PE pe_num="0" prog_file="x0.bit"/>
      <PE pe_num="1" prog_file="x1.bit"/>
      <PE pe_num="2" prog_file="x2.bit"/>
   </slaac1v>
</boards>
<channels>
   <channel>
      <src><host port="HOST_OUT"/></src>
      <dest><node board_id="node1" port="SLAAC_FIFO_A1"/></dest>
   </channel>
   <channel>
      <src><node board_id="node1" port="SLAAC_FIFO_B1"/></src>
      <dest><node board_id="node2" port="SLAAC_FIFO_A1"/></dest>
   </channel>
   <channel>
      <src><node board_id="node2" port="SLAAC_FIFO_B1"/></src>
```

```
36      <dest><node board_id="node0" port="OSIRIS_FIFO_A0"/></dest>
37    </channel>
38    <channel>
39      <src><node board_id="node0" port="OSIRIS_FIFO_B0"/></src>
40      <dest><host port="HOST_IN"/></dest>
41    </channel>
42  </channels>
43  </config>
```

Listing A.3: Streaming System Host Program

```
1  /* ************************************************************************
2   * This program for a streaming structure of FPGA boards performing the
3   *     Smith-Waterman global comparison
4   *     Usage: fifo_test   xml_config_file   gene_file
5   * Virginia Tech Configurable Computing Laboratory
6   * Copyright 2002    William J. Worek
7   ************************************************************************/
8  #include <stdio.h>
9  #include <stdlib.h>
10 #include <string.h>
11 #include <fstream.h>
12 #include <iostream.h>
13 #include "acs.h"
14 #include "acs_system.h"
15 #include "acs_xmlconfig.h"
16 /************************************************************************/
17 /*       Constants                                                      */
```

```
18 /*****************************************************************/
19 #define CLOCK_FREQUENCY 50
20 #define BUF_SIZE 0x7fffff
21 /*****************************************************************/
22 /*       Data Structures                                         */
23 /*****************************************************************/
24 struct FIFOWORD {
25     unsigned int top;
26     unsigned int bottom;
27 };
28 /*****************************************************************/
29 /* Dequeue function                                              */
30 /* Loops until all the information is dequeued                   */
31 /*****************************************************************/
32 inline void my_dequeue(unsigned char * in_buffer, int size, int port,
33         ACS_SYSTEM * system, ACS_STATUS status) {
34     int dequeue = 0;
35     int dval;
36     while(dequeue < size)
37     {
38         dval = ACS_Dequeue(&(in_buffer[dequeue]), size-dequeue, port, system, &status);
39         dequeue += dval;
40     }
41 }
42 /*****************************************************************/
43 /* Main Routine                                                  */
```

```
44 /*****************************************************************/
45 void main(int argc, char ** argv)
46 {
47    FIFOWORD fifo_input, fifo_output;   // FIFOWORD formatted data
48    ACS_STATUS status;                  // Status of ACS function
49    ACS_SYSTEM * system;                // ACS system object
50    ACS_CLOCK clock;                    // ACS clock object
51    ACS_XMLConfig config;               // object for configuring from XML
52    ACS_BOARD_INFO board_info[3];       // Board information
53    int   num_sites;                    // Number of nodes in the system
54    int   node_id, node_num;            // ID and index of node
55    int pe_mask, num_pes;               // FPGA board processing element data
56    int   rc;                           // Return code for ACS functions
57    int   i, j;                         // Loop indices
58    int num_sent;                       // Size of datbase
59    int thread_status;                  // Thread status information
60    char filename[40];                  // database file name
61    unsigned char *out_buffer;          // database buffer
62    unsigned char *in_buffer;           // processed data buffer
63    out_buffer = new unsigned char [BUF_SIZE*8];
64    in_buffer = new unsigned char [BUF_SIZE*8];
65    ifstream  infile ;                  // database file
66    int enport = 0;                     // Enqueue port
67    int deport = 1;                     // Dequeue port
68    int send, sendsize;                 // Amount sent; block size to send
69      // Parse command line parameters
```

```
70      // Get database file name
71    if (argc == 3)
72        strcpy(filename, argv[2]);
73    else
74    {
75        printf ("USAGE: %s xml_file gene_file\n", argv[0]);
76        return 1;
77    }
78    strcat(filename ,". gen");
79    //-----------------------------------
80    //-- Construct and start ACS system
81    //-----------------------------------
82      // Set XML configuration file name
83    printf ("Configuring the system using: %s\n", argv[1]);
84    config.setFilename(argv[1]);
85      // Get system object from XML configuration system
86    if (config.getSystem(&system) != ACS_SUCCESS)
87    {
88        printf ("Failed to create the system object, terminating\n");
89        return 2;
90    }
91      // Get number of nodes in the system
92    if (config.getNodeCount(&num_sites) != ACS_SUCCESS)
93        printf ("Failed to get the number of nodes\n");
94      // Start the clocks on the nodes in reverse order
95    for (node_num = num_sites - 1; node_num >= 0; node_num--)
```

```
96  {
97      // Get number and information for this particular node
98      ACS_Get_Board_Info(system, node_num, &board_info[node_num]);
99      if ((rc = config.getNodeNumber(node_num, &node_id)) != ACS_SUCCESS)
100         printf ("Failed to get number for node: %d\n", node_num);
101     pe_mask = 0;
102     for (i = 0; i < board_info[node_num].num_pes; i++)
103         pe_mask = pe_mask | board_info[node_num].pe_mask[i];
104     // Set the frequency of the clock on this node
105     clock.frequency = CLOCK_FREQUENCY;
106     clock.countdown = 0;
107     if ((rc = ACS_Clock_Set(&clock, node_id, system, &status)) != ACS_SUCCESS)
108         printf ("Failure in ACS_Clock_Set, rc = %d\n", rc);
109     // Assert reset signal
110     if ((rc = ACS_Reset(node_id, system, pe_mask, 1, &status)) != ACS_SUCCESS)
111         printf ("Failure in ACS_Reset, rc = %d\n", rc);
112     // Start the board
113     if ((rc = ACS_Run(node_id, system, &status)) != ACS_SUCCESS)
114         printf ("Failure in ACS_Run, rc = %d\n", rc);
115     // Deassert reset signal
116     if ((rc = ACS_Reset(node_id, system, pe_mask, 0, &status)) != ACS_SUCCESS)
117         printf ("Failure in ACS_Reset, rc = %d\n", rc);
118 }
119 //-------------------------------------
120 //-- Read and process database
121 //-------------------------------------
```

```
122   infile .open(filename,ios :: in );
123   while (! infile .eof()) {
124         // Read in database from file
125       num_sent = 0;
126        infile  >> fifo_input.top >> fifo_input.bottom;
127       while ((! infile .eof()) && (num_sent < BUF_SIZE))
128       {
129           ((unsigned int *) outbuffer)[2*num_sent] = fifo_input.top;
130           ((unsigned int *) outbuffer)[2*num_sent+1] = fifo_input.bottom;
131           num_sent++;
132            infile  >> fifo_input.top >> fifo_input.bottom;
133       }
134         // Stream data through system in blocks
135       send = 0;
136       sendsize = 0;
137       count = 0;
138       while (send < num_sent)
139       {
140           if((num_sent-send) > 2500)
141               sendsize = 2500;
142           else
143               sendsize = num_sent-send;
144           ACS_Enqueue(&out_buffer[send*8], 8*sendsize, enport, system, &status);
145           my_dequeue(in_buffer, 8*sendsize, deport, system, status );
146           send += sendsize;
147       }
```

```
148     }
149      infile . close ();
150     //-----------------------------------
151     //-- Free memory and destroy ACS system
152     //-----------------------------------
153     delete(out_buffer);
154     delete(in_buffer);
155       // Destroy the acs_system, release the resource.
156     if ((rc = ACS_System_Destroy(system, &status)) != ACS_SUCCESS)
157         printf ("Call to ACS_System_Destroy unsuccessful, error: %d\n", rc);
158       // finalize the acs world, destroy throughly.
159     if ((rc = ACS_Finalize()) != ACS_SUCCESS)
160         printf ("Call to ACS_Finalize unsuccessful, error: %d\n", rc);
161     printf ("VPI & SU, ECPE, CCM Lab. Spring 2001\n");
162 }
```

Listing A.4: Distributed System XML Configuration File

```
1  <?xml version='1.0' encoding='UTF-8'?>
2  <!DOCTYPE config SYSTEM "/project/acs_api/src/config/acs_conf.dtd">
3  <config>
4  <header>
5     <commtype>MPI</commtype>
6     <version>0.03</version>
7     <author>William J. Worek</author>
8     <desc>Distributed System Configuration</desc>
9  </header>
10 <boards>
```

```
11    <osiris board_id="node0" location="local" ctrl_proc_dir="/project/acs_api/bin">
12        <PE pe_num="0" prog_file="xp_es.bit"/>
13    </osiris>
14    <slaac1v board_id="node1" location="xanadu" ctrl_proc_dir="/project/acs_api/bin">
15        <PE pe_num="0" prog_file="x0.bit"/>
16        <PE pe_num="1" prog_file="x1.bit"/>
17        <PE pe_num="2" prog_file="x2.bit"/>
18    </slaac1v>
19    <slaac1v board_id="node2" location="cowbell" ctrl_proc_dir="/project/acs_api/bin">
20        <PE pe_num="0" prog_file="x0.bit"/>
21        <PE pe_num="1" prog_file="x1.bit"/>
22        <PE pe_num="2" prog_file="x2.bit"/>
23    </slaac1v>
24  </boards>
25  <channels>
26    <channel>
27        <src><host port="0"/></src>
28        <dest><node board_id="node0" port="OSIRIS_FIFO_A0"/></dest>
29    </channel>
30    <channel>
31        <src><node board_id="node0" port="OSIRIS_FIFO_B0"/></src>
32        <dest><host port="1"/></dest>
33    </channel>
34    <channel>
35        <src><host port="2"/></src>
36        <dest><node board_id="node1" port="SLAAC_FIFO_A1"/></dest>
```

```
37    </channel>
38    <channel>
39       <src><node board_id="node1" port="SLAAC_FIFO_B1"/></src>
40       <dest><host port="3"/></dest>
41    </channel>
42    <channel>
43       <src><host port="4"/></src>
44       <dest><node board_id="node2" port="SLAAC_FIFO_A1"/></dest>
45    </channel>
46    <channel>
47       <src><node board_id="node2" port="SLAAC_FIFO_B1"/></src>
48       <dest><host port="5"/></dest>
49    </channel>
50 </channels>
51 </config>
```

Listing A.5: Distributed System Host Program

```
1  /* ************************************************************************
2   * This program for a distributed structure of FPGA boards performing the
3   *     Smith-Waterman global comparison
4   *     Usage: fifo_test  xml_config_file  gene_file
5   * Virginia Tech Configurable Computing Laboratory
6   * Copyright 2002   William J. Worek
7   ************************************************************************/
8  #include <stdio.h>
9  #include <stdlib.h>
10 #include <string.h>
```

```
11 #include "acs.h"
12 #include "acs_system.h"
13 #include "acs_xmlconfig.h"
14 #include <fstream.h>
15 #include <iostream.h>
16 #include <pthread.h>
17 /*****************************************************************/
18 /*      Constants                                                */
19 /*****************************************************************/
20 #define CLOCK_FREQUENCY 50 //Clock frequency for the FPGA boards
21 #define BUF_SIZE 0x7fffff //Buffer size for the database file
22 #define MEM_SIZE 250000 //Size of the Addressable Memory (SLAAC-1V)
23 /*****************************************************************/
24 /*      Data Structures                                          */
25 /*****************************************************************/
26 struct FIFOWORD {
27    unsigned int top;
28    unsigned int bottom;
29 };
30 struct thread_data{
31    int num_sent;
32    int node_id;
33    ACS_SYSTEM * system;
34    unsigned char *out_buffer;
35 };
36 /*****************************************************************/
```

```
37 /* Dequeue function                                                    */
38 /* Loops until all the information is dequeued                        */
39 /*********************************************************************/
40 inline void my_dequeue(unsigned char * in_buffer, int size, int port,
41         ACS_SYSTEM * system, ACS_STATUS status) {
42     int dequeue = 0;
43     int dval;
44     while(dequeue < size)
45     {
46         dval = ACS_Dequeue(&(in_buffer[dequeue]), size-dequeue, port, system, &status);
47         dequeue += dval;
48     }
49 }
50 /*********************************************************************/
51 /* Thread for the Osiris Board.                                       */
52 /* Passes the database through the board using FIFOs.                 */
53 /*********************************************************************/
54 void osiris (void * threadarg)
55 {
56     ACS_SYSTEM * system;      // ACS system Object
57     ACS_STATUS status;        // Status of function call
58     struct thread_data * my_data;// Place holder for thread parameters
59     unsigned char * out_buffer; // database buffer
60     int num_sent;             // number of words in the database buffer
61     int i;                    // loop index variable
62     int enport = 0;           // enqueue port
```

```
63    int deport = 1;                 // dequeue port
64    int node_id;                    // ID value of osiris node
65    unsigned char *in_buffer;       // buffer for the output of the board
66    in_buffer = new unsigned char [BUF_SIZE*8];
67      // Reconstruct parameters from the threadarg
68    my_data = (struct thread_data *) threadarg;
69    system = my_data->system;
70    out_buffer = my_data->out_buffer;
71    num_sent = my_data->num_sent;
72    node_id = my_data->node_id;
73      //Enqueue database and Dequeue Results
74    ACS_EnqueueNode(out_buffer, 8*num_sent, node_id, enport, system, &status);
75    my_dequeueNode(in_buffer, 8*num_sent, node_id, deport, system, status);
76    delete(in_buffer);
77    pthread_exit(0);
78 }
79 /*********************************************************************/
80 /* Thread for the SLAAC-1V Board.                                    */
81 /* Passes the database through the board using Memories.             */
82 /*********************************************************************/
83 void slaac1v (void * threadarg)
84 {
85    ACS_SYSTEM * system;      // ACS system Object
86    ACS_STATUS status;        // Status of function call
87    ACS_ADDRESS address;
88    struct thread_data * my_data;// Place holder for thread parameters
```

```
89   unsigned char * out_buffer; // database buffer
90   int num_sent;               // number of words in the database buffer
91   int i;                      // loop index variable
92   int send, sendsize;         // amount of data sent; size of block to send
93   int enport = 0;             // enqueue port
94   int deport = 1;             // dequeue port
95   int node_id;                // ID value of osiris node
96   unsigned char *in_buffer;   // buffer for the output of the board
97   in_buffer = new unsigned char [BUF_SIZE*8];
98     // Reconstruct parameters from the threadarg
99   my_data = (struct thread_data *) threadarg;
100  system = my_data->system;
101  out_buffer = my_data->out_buffer;
102  num_sent = my_data->num_sent;
103  node_id = my_data->node_id;
104  enport = (node_id-1)*2;
105  deport = enport+1;
106  sendsize = MEM_SIZE;
107  send = 0;
108    // Loop through the database buffer
109  while (send < num_sent)
110  {
111     address.pe = 2;
112     address.mem = 0;
113     address.offset = 0;
114       // Load Memory with portion of database
```

```
115         ACS_Write((void *) &out_buffer[send], sendsize, 4, node_id, &address, system,
116                 &status);
117           // Signal FPGA to pass memory through the system
118         ACS_Enqueue(in_buffer, 8, enport, system, &status);
119           // Wait for FPGA to complete processing
120         my_dequeue((unsigned char *)in_buffer, 8, deport, system, status);
121         address.mem = 1;
122           // Read the computed values from memory
123         ACS_Read((void *) in_buffer, sendsize, 4, node_id, &address, system, &status);
124         send += sendsize;
125     }
126     delete(in_buffer);
127     pthread_exit(0);
128 }
129 /*********************************************************************/
130 /* Main Routine                                                      */
131 /*********************************************************************/
132 void main(int argc, char ** argv)
133 {
134     FIFOWORD fifo_input, fifo_output;   // FIFOWORD formatted data
135     thread_data thread_data_array[3];   // Data for sending to threads
136     ACS_STATUS status;                  // Status of ACS function
137     ACS_SYSTEM * system;                // ACS system object
138     ACS_CLOCK clock;                    // ACS clock object
139     ACS_XMLConfig config;               // object for configuring from XML
140     ACS_BOARD_INFO board_info[3];       // Board information
```

```
141  int  num_sites;                    // Number of nodes in the system
142  int  node_id, node_num;            // ID and index of node
143  int pe_mask, num_pes;              // FPGA board processing element data
144  int  rc;                           // Return code for ACS functions
145  int  i, j;                         // Loop indices
146  int num_sent;                      // Size of datbase
147  int thread_status;                 // Thread status information
148  char filename[40];                 // database file name
149  unsigned char *out_buffer;         // database buffer
150  out_buffer = new unsigned char [BUF_SIZE*8];
151  ifstream  infile ;                 // database file
152  pthread_t  isis_thread , xanadu_thread, cowbell_thread; // disritbuted threads
153    // Parse command line parameters
154    // Get database file name
155  if (argc == 3)
156     strcpy(filename, argv[2]);
157  else
158  {
159     printf (" USAGE: %s xml_file gene_file\n", argv[0]);
160     return 1;
161  }
162  strcat(filename ,". gen");
163  //-----------------------------------
164  //-- Construct and start ACS system
165  //-----------------------------------
166    // Set XML configuration file name
```

```
167   printf ("Configuring the system using: %s\n", argv[1]);
168   config.setFilename(argv[1]);
169     // Get system object from XML configuration system
170   if (config.getSystem(&system) != ACS_SUCCESS)
171   {
172       printf ("Failed to create the system object, terminating\n");
173       return 2;
174   }
175     // Get number of nodes in the system
176   if (config.getNodeCount(&num_sites) != ACS_SUCCESS)
177       printf ("Failed to get the number of nodes\n");
178     // Start the clocks on the nodes in reverse order
179   for (node_num = num_sites - 1; node_num >= 0; node_num--)
180   {
181         // Get number and information for this particular node
182       ACS_Get_Board_Info(system, node_num, board_info[node_num]);
183       if ((rc = config.getNodeNumber(node_num, &node_id)) != ACS_SUCCESS)
184           printf ("Failed to get number for node: %d\n", node_num);
185       pe_mask = 0;
186       for (i = 0; i < board_info[node_num].num_pes; i++)
187           pe_mask = pe_mask | board_info[node_num].pe_mask[i];
188         // Set the frequency of the clock on this node
189       clock.frequency = CLOCK_FREQUENCY;
190       clock.countdown = 0;
191       if ((rc = ACS_Clock_Set(&clock, node_id, system, &status)) != ACS_SUCCESS)
192           printf ("Failure in ACS_Clock_Set, rc = %d\n", rc);
```

```
193         // Assert reset signal
194       if ((rc = ACS_Reset(node_id, system, pe_mask, 1, &status)) != ACS_SUCCESS)
195           printf ("Failure in ACS_Reset, rc = %d\n", rc);
196         // Start the board
197       if ((rc = ACS_Run(node_id, system, &status)) != ACS_SUCCESS)
198           printf ("Failure in ACS_Run, rc = %d\n", rc);
199         // Deassert reset signal
200       if ((rc = ACS_Reset(node_id, system, pe_mask, 0, &status)) != ACS_SUCCESS)
201           printf ("Failure in ACS_Reset, rc = %d\n", rc);
202   }
203   //--------------------------------------
204   //-- Read and process database
205   //--------------------------------------
206    infile.open(filename,ios::in);
207   while (!infile.eof()) {
208         // Read in database from file
209       num_sent = 0;
210        infile >> fifo_input.top >> fifo_input.bottom;
211       while ((!infile.eof()) && (num_sent < BUF_SIZE))
212       {
213           ((unsigned int *) outbuffer)[2*num_sent] = fifo_input.top;
214           ((unsigned int *) outbuffer)[2*num_sent+1] = fifo_input.bottom;
215           num_sent++;
216            infile >> fifo_input.top >> fifo_input.bottom;
217       }
218         // Pack parameters for sending to thread
```

```
219     for (i = 0; i < 3; i++)
220     {
221         thread_data_array[i].out_buffer = out_buffer;   //pointer to database
222         thread_data_array[i].system = system;           //pointer to system
223         thread_data_array[i].num_sent = num_sent;       //size of database
224         thread_data_array[i].node_id = i;               //node ID
225     }
226       // Process database on each node
227     pthread_create(&cowbell_thread, NULL, (void * (*)(void *)) slaac1v,
228             (void *) &thread_data_array[2]);
229     pthread_create(&xanadu_thread, NULL, (void * (*)(void *)) slaac1v,
230             (void *) &thread_data_array[1]);
231      osiris (&thread_data_array[0]);
232       // Wait for each node to complete
233     pthread_join(cowbell_thread,(void **) &thread_status );
234     pthread_join(xanadu_thread,(void **) &thread_status );
235 }
236  infile .close();
237 //--------------------------------
238 //-- Free memory and destroy ACS system
239 //--------------------------------
240 delete(out_buffer);
241   // Destroy the acs_system, release the resource.
242  if ((rc = ACS_System_Destroy(system, &status)) != ACS_SUCCESS)
243     printf (" Call to ACS_System_Destroy unsuccessful, error: %d\n", rc);
244   // finalize the acs world, destroy throughly.
```

```
245     if ((rc = ACS_Finalize()) != ACS_SUCCESS)
246         printf ("Call to ACS_Finalize unsuccessful, error: %d\n", rc);
247     printf ("VPI & SU, ECPE, CCM Lab. Spring 2002\n");
248 }
```

Vita

William J. Worek was born and raised in Clifton, a small town in Virginia. He matriculated to Virginia Tech in 1995 after completing his high school education at Thomas Jefferson High School for Science and Technology. In 1999, he received a summa cum laude Bachelor's degree in honors in Computer Engineering. During his undergraduate career, William spent a semester working at Northern Telecom in the XPM Diagnostic group. William decided to continue at Virginia Tech for his Masters Degree in Computer Engineering. As a graduate student, William worked in the Configurable Computing Machines laboratory developing an API for distributed adaptive computing systems. He finds time to enjoy the outdoors by skiing, playing tennis, and hiking. William spends a lot of time giving back to the community as a Life Member of the Virginia Tech Rescue Squad.