

Implementing an Application Programming Interface for Distributed Adaptive Computing Systems

by

Kuan Yao

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

Mark T. Jones, Chair
Peter M. Athanas
Scott F. Midkiff

May 31, 2000
Blacksburg, Virginia

Keywords: MPI, Adaptive Computing, FPGA, API
©Copyright 2000 Kuan Yao

Implementing an Application Programming Interface for Distributed Adaptive Computing Systems

Kuan Yao

Committee Chairman: Dr. Mark T. Jones

Committee Members: Dr. Peter M. Athanas, Dr. Scott F. Midkiff

The Bradley Department of Electrical and Computer Engineering

(ABSTRACT)

Developing applications for distributed adaptive computing systems (ACS) requires developers to have knowledge of both parallel computing and configurable computing. Furthermore, portability and scalability are required for developers to use innovative ACS research directly in deployed systems. This thesis presents an Application Programming Interface (API) implementation developed in a scalable parallel ACS system. The API gives the developer the ability to easily control both single board and multi-board systems in a network cluster environment. The API implementation is highly portable and scalable, allowing ACS researchers to easily move from a research system to a deployed system. The thesis details the design and implementation of the API, as well as analyzes its performance.

ACKNOWLEDGMENTS

I am indebted to Dr. Mark Jones, my major professor and thesis chair, for his inspiration and guidance at various stages throughout this project. Dr. Jones has generously given much of his time to talk with me and to read and consider the drafts I have prepared. His expertise and advice are invaluable in this project.

I would also like to acknowledge and thank the members of my committee Dr. Athanas and Dr. Midkiff for reading my thesis and providing insightful opinions. For help with the proofreading, I am indebted to Katherine for her helpful suggestions and enormous patience.

A special thanks is due to my parents, my uncle, and my wife, who have always held great faith in me and extended generous support to my study and my life.

Table Of Contents

1. INTRODUCTION	1
1.1 BACKGROUND.....	1
1.2 PROBLEM	2
1.3 STATEMENT	3
1.4 THESIS ORGANIZATION	3
2. INTRODUCTION TO ADAPTIVE COMPUTING AND MPI.....	4
2.1 OVERVIEW	4
2.2 ADAPTIVE COMPUTING DEVICES	4
2.2.1 <i>WildForce</i>	4
2.2.2 <i>Splash 2</i>	5
2.2.3 <i>SLAAC Project</i>	8
2.3 MESSAGE PASSING INTERFACE	9
2.4 SUMMARY.....	11
3. ACS API DESCRIPTIONS	12
3.1 OVERVIEW	12
3.2 SYSTEM MANAGEMENT	13
3.3 BOARD MANAGEMENT	15
3.4 MEMORY ACCESSING AND DATA STREAMING FUNCTIONS	16
3.5 NON-BLOCKING COMMAND	17
3.6 GROUP OPERATIONS	18
3.7 SUMMARY.....	19
4. DESIGN AND IMPLEMENTATION	20
4.1 OVERVIEW	20
4.2 OBJECT ORIENTED DESIGN	23
4.2.1 <i>World object</i>	23
4.2.2 <i>System object</i>	24
4.2.3 <i>Node Object</i>	25
4.2.4 <i>Channel object</i>	28
4.2.5 <i>Communication Object</i>	29
4.2.6 <i>Request object</i>	30
4.2.7 <i>Group object</i>	30
4.3 COMMUNICATION LAYER	31
4.3.1 <i>Establish communication</i>	31
4.3.2 <i>Buffer Management</i>	32
4.3.3 <i>Flow Control</i>	38
4.3.4 <i>Message ordering and serialization</i>	41
4.4 CONTROL PROCESS	43
4.5 NODE/CHANNEL MANAGEMENT	44
4.6 CACHE MANAGEMENT & RUN-TIME RECONFIGURATION	47

4.7 OS INDEPENDENT LAYER	48
4.8 SUMMARY.....	49
5. TESTING APIS	50
5.1 OVERVIEW	50
5.2 TEST PROGRAMS AND PERFORMANCE RESULTS	50
5.2.1 <i>Memory Access Function Test</i>	50
5.2.2 <i>Stream Data Function Test</i>	53
5.2.3 <i>Non-blocking Function Testing</i>	57
5.2.4 <i>Group Function Testing</i>	59
5.2.5 <i>Configuration Cache Testing</i>	59
5.3 TEST TABLE	60
6. CONCLUSION AND FUTURE WORK.....	62
6.1 SUMMARY.....	62
6.2 FUTURE WORK.....	62
REFERENCES.....	64
APPENDIX 1. THE ACS API SPECIFICATION	66
A1.1 DATA STRUCTURES	66
A1.2 SUPPORT FUNCTIONS	68
A1.3 SYSTEM SETUP FUNCTIONS	68
A1.4 MEMORY ACCESS FUNCTIONS	71
A1.5 BOARD MANAGEMENT FUNCTIONS.....	71
A1.6 STREAMING DATA FUNCTIONS.....	75
A1.7 GROUP MANAGEMENT	76
A1.8 NON-BLOCKING FUNCTION	76
APPENDIX 2. TESTING CODE FOR ACS API	80
A2.1 MEMORY ACCESS	80
A2.2 STREAMING DATA.....	83
A2.2.a <i>Measuring RTT Value</i>	83
A2.2.b <i>Measuring throughput</i>	88
A2.3 NON-BLOCKING.....	93
A2.4 TESTING ACS_ALL.....	96
A2.5 CACHE CONFIGURATION FILE	100

List of Figures

FIGURE 1.1: VIRGINIA TECH TOP ARCHITECTURE AND A “RING” BASED	2
FIGURE 2.1: WILDFORCE BOARD ARCHITECTURE [10]	5
FIGURE 2.2: TWO BOARD SPLASH SYSTEM [10].....	6
FIGURE 3.1: CODE FRAGMENT FOR SETTING UP THE SYSTEM	14
FIGURE 3.2: CODE FRAGMENT FOR CONFIGURING BOARD AND WRITING DATA TO THE MEMORY [11].....	16
FIGURE 3.3: CODE FRAGMENT FOR ENQUEUEING AND DEQUEUEING DATA [11].....	17
FIGURE 3.4: USING ACS_ALL TO BROADCAST COMMAND AND DATA TO ALL THE BOARDS	19
FIGURE 4.1: CONTROL THE ACS SYSTEM OVER THE NETWORK	21
FIGURE 4.2: SYSTEM OBJECT AND ITS DATA.....	25
FIGURE 4.3 NODE OBJECT FAMILIES	27
FIGURE 4.4: CHANNEL OBJECT	29
FIGURE 4.5: THE CONFIGURATION FILE USED IN WMPI	31
FIGURE 4.6: PROCEDURES OF RECEIVING MESSAGES	33
FIGURE 4.7: MESSAGES WILL GO INTO A RIGHT BUFFER SIZE IN THE COMMUNICATION OBJECT	35
FIGURE 4.8: A MESSAGE’S FORMAT AND DATA STRUCTURE.....	36
FIGURE 4.9: THE BUFFER STATUS IN SENDING MESSAGES	38
FIGURE 4.10: THE BUFFER STATUS IN SENDING-RECEIVING MESSAGES	38

FIGURE 4.11: THE DESTINATION NODE FIFO IS NOT FULL AND.....	40
FIGURE 4.12: THE ESTINATION NODE FIFO IS FULL AND DATA IS PUT INTO THE DESTINATION NODE'S BUFFER. AFTER THE FIFO IS AVAILABLE TO ENQUEUE MORE DATA, THE DESTINATION NODE EXTRACTS DATA FROM THE CHANNEL BUFFER AND PUTS IT INTO THE FIFO. IT THEN ACKNOWLEDGES THIS TO THE CHANNEL.....	41
FIGURE 4.13: CONTROL PROCESS LOOP.....	43
FIGURE 4.14: THE REMOTE NODE ON THE REMOTE MACHINE TWO.....	46
FIGURE 4.15: THIS TYPE OF CHANNEL IS FROM A NODE TO A HOST.....	46
FIGURE 4.16: CACHE CONFIGURATION FILE IN THE.....	47
FIGURE 4.17: OS INDEPENDENT LAYER	49
FIGURE 5.1: RTT VALUES WITH THE NUMBER OF BOARDS IN THE SYSTEM	55

List Of Tables

TABLE 5.1: AVERAGE MEMORY ACCESS TIME IN LINUX ENVIRONMENT.....	52
TABLE 5.2: AVERAGE MEMORY ACCESSING TIME IN WINDOWS NT ENVIRONMENT.....	53
TABLE 5.3: AVERAGE ROUND TRIP TIME TO PASS THROUGH 1024 BYTES DATA TO SYSTEM	54
TABLE 5.4: SYSTEM AVERAGE THROUGHPUT IN DIFFERENT NUMBER OF BOARDS	56
TABLE 5.5: USING NON-BLOCKING FUNCTIONS REDUCES THE TOTAL EXECUTION TIME.	58
TABLE 5.6: TIME OF RECONFIGURING BOARD	60
TABLE 5.7: API TEST TABLE.....	60

1. Introduction

1.1 Background

Adaptive Computing Systems (ACS) dynamically reconfigure their logic and/or data paths to match application requirements. Instead of using traditional microprocessors, they use reconfigurable computing boards, which can provide significant improvements in speed over general-purpose microprocessors for many applications. Because the modern cluster computing community uses workstations and COTS high-speed networks to build high-performance parallel systems [1], a logical way to build scalable parallel ACS systems is to cluster FPGA-accelerated workstations.

The Tower of Power (ToP) at Virginia Tech is a typical example of a scalable parallel adaptive computing system [2]. The ToP has sixteen Pentium II PCs. Each PC is equipped with a WildForce board [3] and tightly coupled to a Myricom LAN/SAN card [4]. These PCs are connected through a sixteen-port Myrinet switch. A total of 80 Xilinx XC4062XL FPGAs [5] and memory banks are distributed throughout the platform. Applications for the ToP include image processing, large-scale histogram matching, and large matrix multiplication. Figure 1.1 shows the ToP architecture and a pipeline programming model.

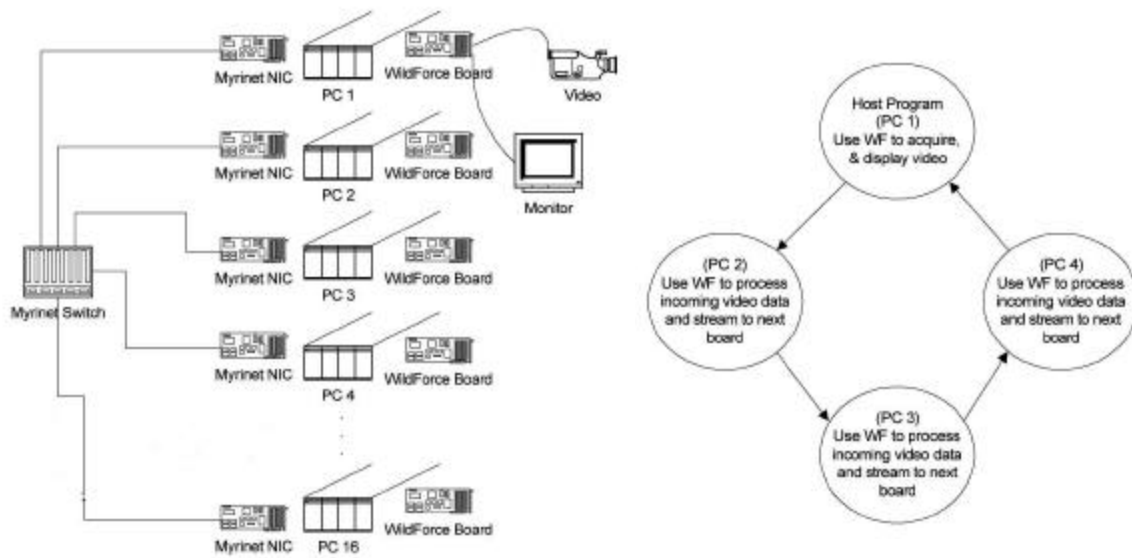


Figure 1.1: Virginia Tech ToP architecture and a “ring” based image-processing pipeline [11]

1.2 Problem

Developing large applications on such a distributed adaptive computing system environment typically requires several steps. First, the developer needs to use VHDL [6] or some other hardware description language to specify the design. This hardware description language is then synthesized to create the FPGA configuration. In addition, the user must write a high-level language program to control the board for performing tasks such as downloading configuration files to the board, setting up clock rates, and sending data to the board [11].

Finally, the developer has to write network communication code to support the parallel system. This last step poses a major problem because building applications in this way is

time-consuming and platform-specific. Moreover, no tool exists to debug and monitor a distributed ACS system. An additional difficulty typically encountered is that developers are generally not familiar with both HDLs and the development of high-performance parallel systems.

1.3 Statement

To address the problems posed in Section 1.2, a scalable Application Programming Interface (API) and a runtime software system was designed and developed to support applications of the network-distributed ACS system. This common API gives the developer the ability to control single board, multi-board, and heterogeneous board systems through the same interface, thus making the application for the ACS system scalable and portable. This thesis will describe in detail the API, its implementation, and the performance of that implementation.

1.4 Thesis organization

The thesis is organized in the following manner. Chapter 2 surveys related research and background material. Chapter 3 gives a description of the API along with a rationale for the design. Chapter 4 provides a detailed description of the implementation of the API. Chapter 5 describes the testing procedures and performance results. Chapter 6 summarizes the work and gives directions for future research.

2. Introduction to Adaptive Computing and MPI

2.1 Overview

A parallel ACS system, such as the Tower of Power, uses multiple adaptive computing boards as its processing elements instead of traditional CPUs. The adaptive computing devices are the key components of an ACS system. This chapter gives an introduction to some of the adaptive computing platforms used in this research project, as well as information on related research. The Message Passing Interface (MPI) [9], a standard message passing parallel programming tool used in this project, is also introduced in this chapter.

2.2 Adaptive Computing Devices

2.2.1 WildForce

The Wildforce board is a reconfigurable board developed by Annapolis Micro Systems Inc. It is used in the Tower of Power (ToP) as attached reconfigurable computing boards. The Wildforce board has five FPGAs, one control-processing element (CPE), and four array-processing elements (PE). A mezzanine connector connects all the memory elements in each PE. Both the PE and host CPU can access these memory elements through a Dual Port Memory Controller (DPMC). All PE's in the board are connected by

a crossbar interconnect network. The crossbar can be configured by the control PE to allow the broadcasting or transferring of data between specified PEs. Each PE can also communicate with its neighboring PE through the systolic bus. The control PE, PE1, and PE4 provide FIFO buffers that can exchange data with the host CPU. They also can communicate with an external I/O. Figure 2.1 below shows the architecture of the Wildforce board [10].

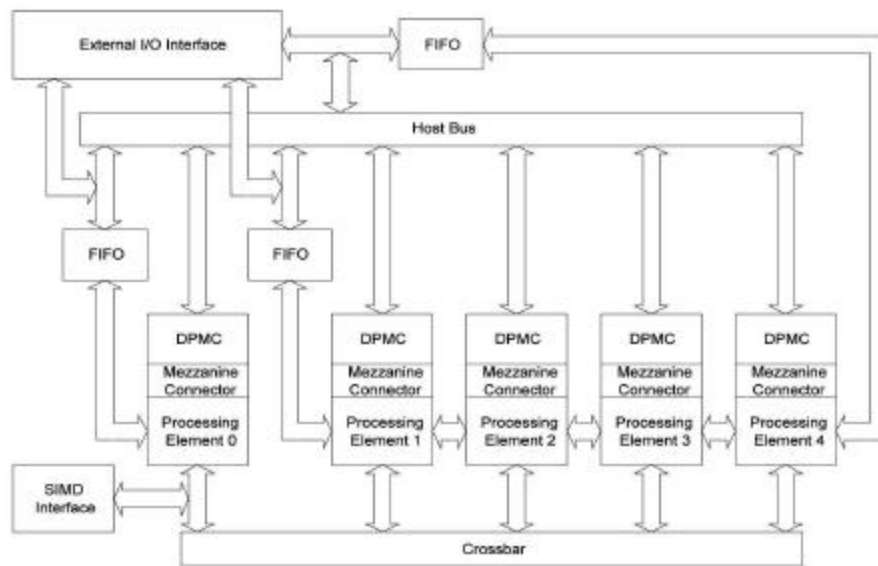


Figure 2.1: Wildforce board architecture [10]

2.2.2 Splash 2

The Splash 2 [12] project began at the Supercomputing Research Center in September 1991 and ended in the spring of 1994. The Splash 2 is an attached processor system using Xilinx XC4010 FPGAs as its processing elements. Figure 2.2 illustrates the system architecture of Splash 2. The system consists of three parts: a SPARCstation 2 host computer with an Adaptive Board; a Splash 2 Interface board; and up to 13 Splash 2

Array Boards. The Adaptive Board extends the host computer's address and data bus to allow the host program to directly access the memory in the Splash 2 system. Each Array Board contains 17 Xilinx XC4010 FPGA chips as its processing elements. Sixteen of them, X1 through X16, form the processing array and are connected by a crossbar. X0 provides the control functions to the crossbar and broadcasts data in SIMD bus to all the other FPGAs.

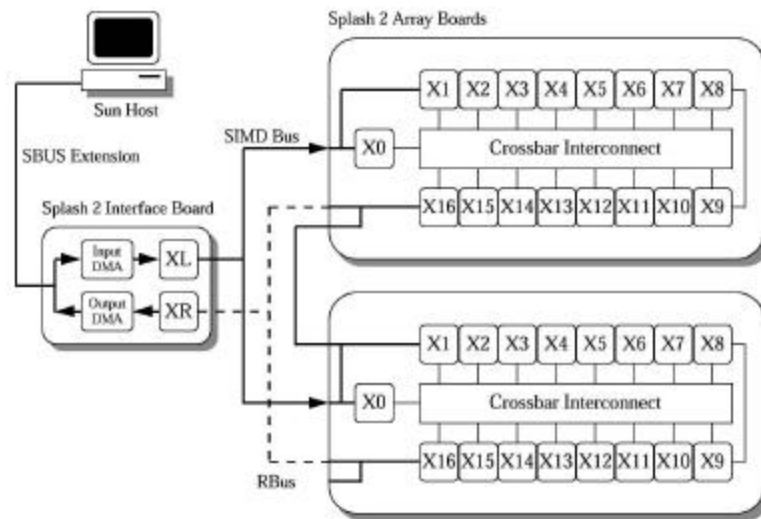


Figure 2.2: Two board Splash system [10]

The Splash 2 system was primarily designed to support applications that have SIMD or a broadcast-of-data model and a linear model. In the SIMD model, the 36-bit-wide data path from the Interface Board to each Array Board serves as a SIMD bus. The Xilinx chip X0 on each Array Board can then broadcast the SIMD Bus data to the other FPGAs on its Array Board. Pattern recognition is a typical application using this model [12].

In the linear model, the 36-bits wide data path can be used to transmit data from the Interface Board to the first FPGA on the first Array Board. The data can then be moved linearly through X1 to X16, then to the first FPGA on the second board, and so forth. Finally, when the data reaches the last FPGA on the last board, it goes back to the Interface Board. This computation model is good for applications that can be deeply pipelined, for example, image processing.

Every Splash 2 application may be divided into three main parts: the portions that run on the Array Boards; the Interface Board; and the host computer. To build a Splash 2 application, the developer first needs to use VHDL as a programming language to write configuration files for the Processing Elements, X1 through X16, the Control Element, X0, and the crossbar in each Array Board. The developer then needs to write configuration files for the Interface Board. Finally, he needs to provide a C program running on the host computer, which is responsible for downloading the configuration data to the FPGAs, establishing input and output data streams, and controlling the clock.

Developing an application for Splash 2 is like designing a program for a massive parallel computer. In the data parallel (SIMD) model, the programmer must control the data layout among the Array Boards, which is critical to the performance of parallel computing. In the pipelined model, the control layout is critical. Therefore, a good algorithm must be used to partition the tasks and data among the PEs so as to maximize the efficiency of the inter-PE communication. However, there are no automated tools that can do this task, the programmer must perform the partitioning manually.

2.2.3 SLAAC Project

The System Level Application of Adaptive Computing (SLAAC) project is sponsored by the Defense Advanced Research Projects Agency (DARPA) Adaptive Computing System program [8]. The goal of SLAAC is to define an open, distributed, scalable, adaptive computing system architecture based on a high-speed network cluster of heterogeneous, FPGA-accelerated nodes.

The SLAAC architecture is based upon a high-speed network of ACS-accelerated nodes. A high-speed network cluster of traditional desktop PCs, where each is accelerated with some form of an ACS accelerator (such as a PCI FPGA-board), is called a Research Reference Platform (RRP). The RRP is an inexpensive, readily available platform for ACS development that tracks advances in workstations, adaptive computing, and cluster computing [14]. The Tower of Power at Virginia Tech is a good example of an existing RRP.

Similar to Splash 2 and Wildforce, the SLAAC architecture is an attached processor system consisting of FPGA's and fast local memories. SLAAC-1 is an attached FPGA-based accelerator on a full-sized 64-bit PCI board. SLAAC-1 features one user-programmable Xilinx 4085 device, two user-programmable Xilinx 40150 devices, and ten 256Kx18 100MHz ZBT synchronous SRAM's. The SLAAC-1 architecture is composed of one single interface FPGA (labeled IF), and three user-programmable FPGA's (labeled 'X0', 'X1', and 'X2'). The IF chip serves as a stable bridge to the host system bus. It provides the configuration, clock, and control logic for the 'X0', 'X1' and 'X2'. The role of the attached system is to program the user FPGA's and control the

system. The SLAAC-1 can execute either synchronously with the host or asynchronously with the DMA channels to transfer data to and from the host memory. Through the clock generator and FIFO's from the IF, the user FPGA's are allowed to operate from a single data-synchronous clock in either mode of operation [14].

SLAAC-2 is an attached FPGA-based accelerator on a 6U VME mezzanine board.

SLAAC-2 features two user-programmable Xilinx 4085 devices, four user-programmable Xilinx 40150 devices, and twenty 256Kx18 100MHz ZBT synchronous SRAM's.

SLAAC-2 is actually two "independent" SLAAC-1 bit-file compatible accelerators on a single board [14].

As part of the SLAAC project, the Configurable Computing Lab at Virginia Tech is building a common API which supports developing applications for the SLAAC system. The development of this API is discussed in detail in later chapters.

2.3 Message Passing Interface

Message passing is a programming paradigm used widely on parallel computers, especially on scalable parallel computers with distributed memory. Message passing is also used on networks of workstations. The Message Passing Interface (MPI) is used as a standard for writing message-passing programs. The goal of MPI is to develop a widely used standard for writing message-passing programs. As such, the interface should establish a practical, portable, efficient, and flexible standard for message passing.

With interfaces to programming languages such as Fortran and C, MPI has great portability. This means that the same message-passing source code can be executed on a variety of machines as long as the MPI library is available. MPI also has the ability to run transparently on heterogeneous systems, where the processors have distinct architectures. Through the ability to span such a heterogeneous system, MPI provides a virtual computing model that hides many architectural differences. The user need not worry whether the code is sending messages between processors of like or unlike architectures. The MPI implementation will automatically do any necessary data conversion and utilize the correct communication protocol.

MPI was designed to enable the overlap of communication and computation, thus hiding communication latencies. This is achieved by the use of non-blocking communication calls which execute communication in the background. MPI also supports scalability through several of its design features. For example, an application can create subgroups of processes that allow collective communication operations only on this subgroup, such as broadcast, multicast, gather, and scatter. MPI guarantees that the underlying transmission of messages is reliable. The user need not check if a message is received correctly [15].

The Tower of Power is a network cluster system. Each workstation in the system has its own memory and processing unit. Thus message passing is the only way to run applications on the system (versus shared memory). There are two major reasons that MPI was chosen for the underlying communication routines of the ACS API implementation. First, MPI provides the ability to control the distributed system at a high

level. In the Splash 2, the programmer has to control multiple boards through configuration files for setting up the Control Element and the crossbar, in marked contrast to MPI. This high-level control ability makes an ACS API application easy to develop and highly scalable. Second, MPI vendors provide a variety of libraries to support different platforms. This makes the ACS application very portable to different environments, including embedded processing environments.

2.4 Summary

The Wildforce, SLAAC-1, and SLAAC-2 are all adaptive computing devices that can dynamically change their logic and data path to match the applications' requirements.

They are used by the SLAAC system as attaching processing elements to create supercomputer-like performance in various applications. The ACS API is developed to support writing applications to run on such systems. MPI, as a standard for writing message-passing programs, is used in the ACS API for low-level communications because of its portability and ease-of-use.

3. ACS API Descriptions

3.1 Overview

The design goal of the ACS API is to provide the developer with a simple, system-level API to control a complex distributed system. The system-level of the API focuses on two important issues in the ACS application: scalability and portability. Scalability requires that the application source code can be easily ported and scaled from small research platforms to large field deployable platforms. Portability requires that the API should provide a common interface for a range of ACS platforms, including embedded system and cluster-based adaptive computing systems. The simplicity of the API is achieved by providing a set of reasonable default behaviors for controlling the system. Meanwhile, through a wider range of API functions, an advanced user can alter these default behaviors – thus attaining more control of the system. The API implementation is open-source, which gives the users the ability to easily add new nodes and new functionality to the API.

The programming model defined in this API is a single application program written in high level language, usually C, that allocates and controls a heterogeneous set of distributed adaptive computing devices. Based on the functionality, the APIs can be divided into five categories: system management, board management, data accessing, group functions, and non-blocking functions. The following sections give a short description of the ACS API, its structures, components, and show how a program

controls the ACS system through the API. The full specification of the API can be viewed in Appendix A1. ACS API Specification.

3.2 System Management

The System Management APIs allow the user to initialize the ACS system. They include *ACS_Initialize()*, *ACS_Finalize()*, *ACS_System_Create()*, and *ACS_System_Destroy()*.

The primary component of the API is to let the programmer specify and create a *system* object, which consists of *node* objects and *channel* objects. A node is defined as a computational unit, for example, a reconfigurable board such as the Wildforce board. A channel is defined as a logical FIFO queue between two nodes. Most of the remaining API calls use this *system* object for controlling the ACS system.

The first step of a host program is to initialize the ACS system. This includes parsing command lines, invoking processes in other workstations, establishing network communication, and initializing global data structures. This step is accomplished by calling *ACS_Initialize()*. The program then makes a call to *ACS_System_Create()* to create the system object, along with the node and channel objects. The user needs to provide node and channel information, such as the location of the node in the network or the source and destination of a channel. The host program then follows an arbitrary user program with more API calls. Finally, the host program will call *ACS_System_Destroy()* to destroy the system object and *ACS_Finalize()* to shutdown the ACS system.

Figure 3.1 shows a fragment of an initialization procedure for an ACS program. The first line calls *ACS_Initialize()* to initialize the ACS system. Lines 3 to 5 are for setting up the node information and telling the system in which workstation the node resides. Lines 8 to 25 are for setting up the channel information. The channels link the node into a ring, as shown in Figure 1.1. Lines 31 and 32 are to close and terminate the ACS system.

```
1.  ACS_Initialize(&argc, &argv, &status);
2.
3.  for (int i=0; i<4; i++) {
4.      memset((void *)&nodes[i], 0, sizeof(ACS_NODE));
5.      nodes[i].site = i;
6.  }
7.
8.  channels[0].src_node    = ACS_HOST_NUM(0);
9.  channels[0].src_port   = 0;
10. channels[0].des_node    = 0;
11. channels[0].des_port    = WF4_Fifo_Pel;
12.
13. for (int i=1; i<4; i++) {
14.     channels[i].src_node = i-1;
15.     channels[i].src_port = WF4_Fifo_Pe4;
16.     channels[i].des_node = i;
17.     channels[i].des_port = WF4_Fifo_Pel;
18.     channels[i].channel_window = 10;
19.     channels[i].dequeue_size  = 1024;
20. }
21.
22. channels[4].src_node    = sites - 1;
23. channels[4].src_port   = WF4_Fifo_Pe4;
24. channels[4].des_node    = ACS_HOST_NUM(0);
25. channels[4].des_port    = 0;
26.
27. ACS_SYSTEM * system;
28. ACS_System_Create(&system, nodes, 4, channels, 5, &status);
29. ...
30. ...
31. ACS_System_Destroy(system);
32. ACS_Finalize();
```

Figure 3.1: Code fragment for setting up the system

3.3 Board management

After creating the system object, the user needs to configure and control the adaptive computing board in the system. The ACS API provides routines to download and read back the configuration file to the board, set up the clock rate, send interrupt and reset signals, and start or stop running the board. Figure 3.2 shows a code fragment that uses the board management API calls to control the board.

Typically, the host program first needs to send configuration files to each board in the system. The configuration files include information to configure the processing elements and board-level configuration such as crossbar switch settings. The second line in Figure 3.2 is sending configuration files to each board, in the local and in the remote machine.

After the configuration is completed, the code fragment sets the clock speed (*ACS_Clock_Set*), starts the clock (*ACS_Run*), and sends a reset signal (*ACS_Reset*) to each node in the system. Finally, as specified in Section 3.4, the code fragment uses memory access functions to write data into the board memory. Figure 3.2 also illustrates the use of *ACS_Interrupt()* to send interrupt signals to the board [11].

Although the boards are distributed in different workstations, the ACS API has completely hidden the network features from the user. To control the board, either on the local or on the remote machine, the user makes exactly the same API call.

```

1.  for (int i=0;i<4;i++) {
2.      ACS_Configure(config[i],i,ring,&status); // send bitstream for each board
3.      ACS_Clock_Set(clock,i,ring,&status); // set clock speed
4.      ACS_Run(i,ring,&status); // start clock
5.      ACS_Reset(i,ring,&status); // send reset signal
6.  }

7.  for (int i=0;i<4;i++) {
8.      // write initial data to each board' s memory
9.      ACS_Write(databuff[i], datalen[i], i, brd_addr[i],ring,&status);
10.     /* then send an interrupt (or inta) signal #1 to the board */
11.     ACS_Interrupt(i,1,ring,&status);
12.  }

```

Figure 3.2: Code fragment for configuring board and writing data to the memory [11]

3.4 Memory Accessing and Data Streaming Functions

The memory access functions include *ACS_Read()* and *ACS_Write()*. They support the host computer in transferring most of the control and data to an adaptive computing device. By accessing memory-mapped data and control registers, the function can also operate a board in a local system bus.

The API supports both explicit data transfer operations, such as *ACS_Write()* and *ACS_Read()*, and implicit data transfer operations such as *ACS_Enqueue()* and *ACS_Dequeue()*. The data stream functions enable the host application to put streaming data into the system and receive streaming data from the system. In the channel-based communication model, the user only needs to explicitly specify the initial entry and final exit of the system port. The *ACS_Enqueue()* and *ACS_Dequeue()* are two primary commands for controlling the communication. Figure 3.3 describes how these commands

control the data flow in a ring system. After a user calls *ACS_Enqueue()* to put data into the system, the data will go linearly through each of the boards in the system, and finally go back to the system. By adding more initializing properties when the system creates channel objects, the user can change the channel behaviors. As shown on lines 18 and 19 in Figure 3.1, the user can change the channel behaviors at creation time through specifying more detailed properties, such as the channel buffer, and the size of data being dequeued. By default, however, the user is not required to do so.

```
1.  /* use the ring to process the required number of images */
2.  for (int i=0;i<NUM_IMAGES;i++) {
3.      /* send image onto channel associated with port 0*/
4.      ACS_Enqueue(image[i],IMAGESIZE,0, ring,&status);
5.      /* get resulting image from channel associated with port 1 */
6.      ACS_Dequeue(result_image[i], RESULT_SIZE,1,ring,&status);
7.  }
```

Figure 3.3: Code fragment for enqueueing and dequeuing data [11]

3.5 Non-blocking Command

An advanced feature of the API is a set of non-blocking commands. The common API functions are blocking commands. Taking the *ACS_Write()* in Figure 3.1 as an example, the host program will be blocked when executing each write. The *ACS_Request()* function can specify a sequence of the API functions, which will be executed as a set. The sequence is called “a request”, which can include commands such as reading/writing memory or raising a reset line.

Once initiated, the request can be committed to execution using *ACS_Commit()*. *ACS_Commit()* issues the commands, creates a handle, and returns control to the user. During the execution of these commands, the user may perform other operations. Using *ACS_Test()*, the user can easily monitor the completion of the set of commands. Using *ACS_Wait()*, on the other hand, the user can make commands wait in a blocking mode. A request may be committed to execution multiple times. The request mechanism improves efficiency by overlapping user task execution with API task execution. For example, a user can submit a non-blocking request for executing tasks in a remote node that overlaps execution on the local node. A further discussion will be given in the next chapter. By combining multiple commands and re-using command sequences, both the network and the API overhead will decrease significantly [11].

3.6 Group Operations

Functions of the ACS group management allow for creating subsets of nodes from a system with an alternative logical ordering. By using groups, the ACS API supports broadcasting and multicasting of data rather than simple point-to-point transfers. Such a broadcast function would be quite useful on a homogeneous system where all the boards are the same. For example, the programmer can configure all the boards at one time instead of calling each individually.

The *ACS_ALL* is a predefined group that represents all nodes in the system. It is used to define a broadcast operation. Figure 3.4 shows using *ACS_ALL* simplified the procedures for sending commands to each node. Lines 1 to 8 use loops to configure each

board individually, while Lines 9 to 11 use ACS_ALL to setup the clock, run the board, and send a reset signal in just one call.

```
1. for (i = 0; i < Number_Of_Boards; i++) {
2.     for (i = 0; i < WF4_MAX_PES; i++) {
3.         config.bitstream = readBitFile (configFileName,
4.             &(config.count));
5.         config.serial_no = serial_no++;
6.         config.pe_mask = WF4_PE (i);
7.         ACS_Configure(&config, node_id, system, &status);
8.     }
9. ACS_Clock_Set(&clock, ACS_ALL, system, &status);
10. ACS_Run(ACS_ALL, system, &status);
11. ACS_Reset(ACS_ALL, system, ACS_PE1, TRUE, &status);
```

Figure 3.4: Using ACS_ALL to broadcast command and data to all the boards

3.7 Summary

Through the ACS API, a programmer can write a simple application that controls a complex network distributed adaptive computer system. The applications developed by the ACS API are easy to port to new systems. This chapter gave an introduction to the API and explained the rationale of the API design. The detailed specifications and descriptions of the ACS API can be viewed in Appendix A1.

4. Design and implementation

4.1 Overview

The design goal of the ACS API is to provide a simple API for the control of a complex distributed adaptive computing system. The API should also be scalable, extendible, and portable. In this chapter, there will be a detailed discussion on how the API was implemented to match with these requirements.

The machine is running a process to control the system across multiple computers. The host program, along with a control thread, controls the host machine. Other computers in the system run a *control* process, which is responsible for executing commands initiated by API calls, monitoring the local adaptive computing board, and communicating with other control processes. Each control process is single-threaded, while the host process is multi-threaded for concurrent execution of the host program. Section 4.4 gives a detailed description for the control process. Figure 4.1 illustrates how the host program and control processes control the system over the network.

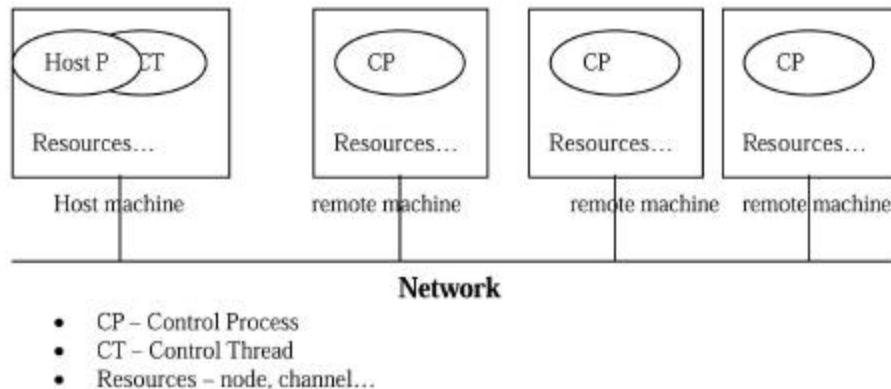


Figure 4.1: Control the ACS system over the network

Because technology is changing rapidly in the adaptive computing community, the implementation of the API requires extensibility. This goal can be achieved by using an object-oriented approach. Using objects follows naturally from the specification of the API. As outlined earlier, the *system* object is a collection of *node* and *channel* objects. A channel object represents a FIFO queue connecting one node to another node. A channel can contain a buffer and settings specifically for controlling flow on this FIFO. A node object represents a computational device in the system. Section 4.5 gives a detailed discussion about how the system object controls and manages the node and channel objects.

Two objects not directly viewed or manipulated by the user are the *communication* object and the *world* object. The *communication* object provides communication between processes on different computers. Section 4.3 gives a detailed introduction of how the ACS communication layer was built. It also discusses the buffer/memory allocation, message ordering, and flow control problems in the ACS system.

The *world* object is used to encapsulate and maintain information of the computing environment when the API is running. For example, the world object contains the network address of all node objects and channel objects. Further, it contains a list of all the adaptive computing boards managed by each control process.

The core of the API implementation is written on these objects. The classes associated with these objects, including virtual function definitions, are defined as part of the core implementation. By taking advantage of inheritance and encapsulation, the distinction between local and remote boards is easily hidden, and new types of boards and communication systems can be seamlessly included. For example, to extend the API to allow control of a new board, a developer just creates a class that inherits the node object and implements all of the virtual functions to allow for control of the new board; the rest of the API implementation remains unchanged [10]. Section 4.2 gives an introduction to the objects used in the ACS API implementation.

Section 4.6 discusses the cache configuration techniques used in run-time reconfiguration. The operating system independent layer, as part of the portability of the ACS API, is presented in Section 4.7.

4.2 Object Oriented Design

4.2.1 World object

The ACS_World object is a starting point for running an ACS system. It initializes the underlying communication, creates the control processes, and starts the performance monitor. When the program is running, it collects runtime information such as network topology, node, and channel objects location. ACS_World is a global object that shares its data with all other objects. When the program terminates, the ACS_World object releases all resources it held, and closes the underlying communication.

There is only one instance of the ACS_World object, *acs_world*. However, each machine in the network has a copy of this object. Any changes to the data in any copy of the *acs_world* will cause changes in all the others. It thus keeps the shared data identical in all the machines.

ACS_World contains the following data:

- Node location table – stores information about the node object location. The table has two columns. One is for the node's ID; the other is where the node is located.
- Channel location table – stores information about the channel object location.
- Configurable computing board information.

ACS_World interface:

- Node registration – Each node needs to register at the world object so that all other objects can view its information. The location is important to the communication object.
- Channel registration – Same as the node registration.

Related APIs:

- *ACS_World_Info()*
- *ACS_Initialize()*
- *ACS_Finalize()*

4.2.2 System object

The ACS_System object is a collection of node objects and channel objects that have been allocated by the host application through the user-level. It manages node and channel objects. The entire system has only one instance of the ACS_System object, *acs_system*, which is on the host machine. It can manage the node and channel objects in both the local and the remote machines. There will be a discussion of the node/channel management in detail in later sections.

Figure 4.2 shows the ACS_System data, which include:

- Node array – stores all nodes – node object or virtual node object.
- Channel array – stores channel objects in the host machine. The control processes keep channels in the remote machines.
- Request array – stores all request objects.

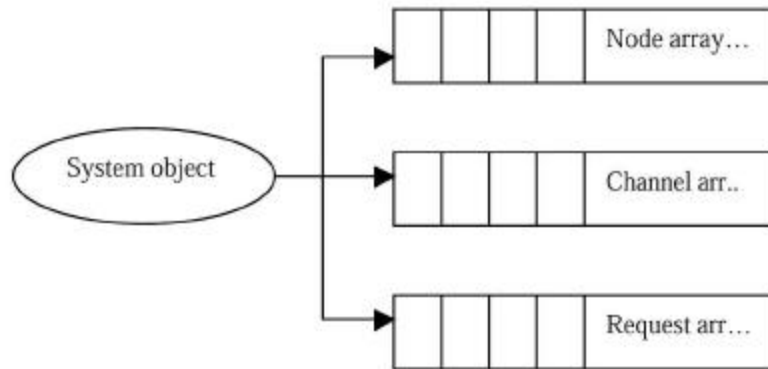


Figure 4.2: System object and its data

ACS_System interface:

- Add/Delete node object
- Add/Delete channel object
- Enqueue/Dequeue

Related APIs:

- *ACS_System_Create()* / *ACS_System_Destroy()*
- *ACS_Node_Add()* / *ACS_Node_Remove()*
- *ACS_Channel_Add()* / *ACS_Channel_Remove()*
- *ACS_Enqueue()* / *ACS_Dequeue()*

4.2.3 Node Object

Node objects represent nodes which are the actual computing. In the ACS system, in particular, nodes refer to the adaptive computing devices that were introduced in Chapter

Two. A base node object is defined as having certain basic properties of a node; e.g., ID, board type. Other nodes can be derived from this base node. For example, consider the concept of a “local” node – a node located on the host machine. Furthermore, a WildForce node and a SLAAC node representing the Wildforce board and the SLAAC board, respectively, are derived from the local node. By inheriting the interface from the base class, accessing different nodes uses the same routines. Thus, at the API level, the user calls the function *ACS_Read()* to read data from an arbitrary board without concern for its type. This isolation of the node development from the API enables the developer to create a class that inherits the node object and implements the entire interface for control of a new board, while keeping the rest of the API implementation unchanged.

The virtual node object is introduced to represent nodes in the remote machine. The virtual node has the same interface as the local node object. It does not, however, perform the actual computation. The purpose of the virtual node is to act as a bridge to the remote node. For example, when the system object accesses the remote node, it sends commands to the virtual node. The virtual node then forwards the command to the remote node through the network. After executing the command, the remote node may reply to a message via the virtual node to the system object. By using the virtual node, accessing a distributed object becomes transparent to the higher layers; thus, making the distributed system easily managed. Moreover, it hides the underlying communication process from the API system. This feature is essential to the ACS API for transporting the implementation to other network environments. For example, the current implementation uses MPI for the underlying network communication. In the future, to use a low level

Myrinet API [17], the developer can simply replace the communication layer while the rest of the API implementation remains unchanged.

Figure 4.3 shows the Node library family's infrastructure. By using virtual functions,

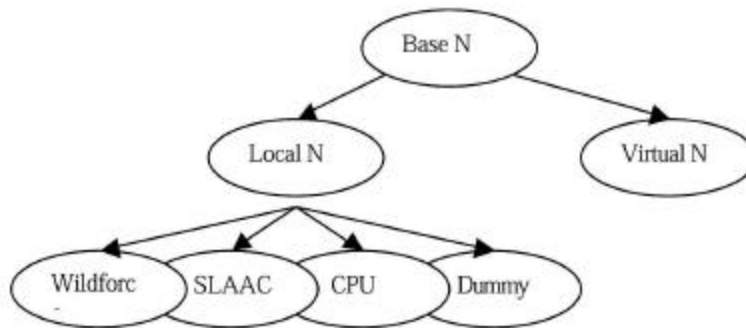


Figure 4.3 Node object families

the access of different nodes is through the same interface. All nodes can recognize the same commands from the system object, even if those nodes are of different types.

Node Library

The class for each type of node has been built into a separate library. Because they inherit the same interfaces from the base node class, it is easy for a user to connect a particular type of node with the system. For example, if the Wildforce board needs to be replaced with the SLAAC board in the application, the developer only needs to declare a new board type in the program header, and link the SLAAC instead of Wildforce library with

the program. The current version of ACS API includes four types of node libraries, Wildforce, WildforceF, SLAAC-1, and RCM.

The WildforceF library was revised from the Wildforce. It has enhanced the FIFO architecture to improve the FIFO throughput. The RCM board differs from other adaptive computing devices in that it uses the Context Switching Reconfigurable Computing (CSRC) chip as one of its Processing Elements. The CSRC technology was developed by Sanders, Inc. The CSRC has the ability to change between a number of programmed functions at a high-speed without adding additional FPGAs. The speed of the context switching is much faster than the speed of reconfiguration in current commercial FPGA technology. In addition, the CSRC FPGA provides the ability for sharing data between contexts, while in commercial FPGAs, the resident data is always destroyed when reprogramming the FPGA [16].

4.2.4 Channel object

A channel is a persistent communication path between two nodes or a node and a host. The endpoints of a channel are marked by the node number (or host number) and a port number. A channel object represents the channel defined above.

A channel object has the following properties:

- A pointer to the source node object along with a source port number;
- A pointer to a destination node object along with a destination port number;
- A buffer to store data that has to be transmitted to the destination;

- And properties of the channel: The size of the buffer, channel window size, which is used in flow control, and the maximum data dequeue from the source node.

A channel with a host as a source feeds data only when the host calls enqueue. Similarly, a channel with a host as a destination clears data only when the host calls dequeue. Inside the system, a channel linked by two nodes is driven by the control process to automatically exchange data. The channel object provides interfaces that can feed data into the destination node and check data from the source node. Figure 4.4 illustrates how the channel object connects the node objects in the system.

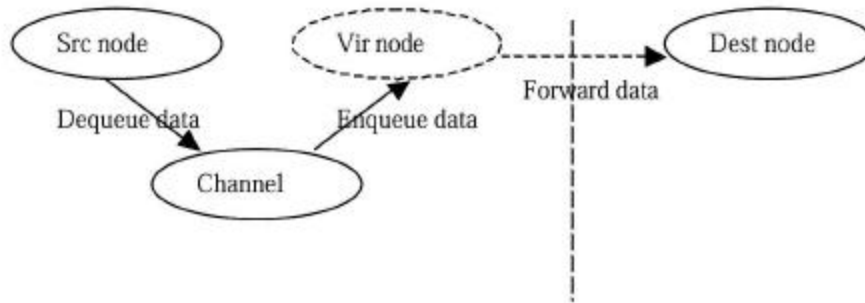


Figure 4.4: Channel object

4.2.5 Communication Object

The communication features are encapsulated into a communication object, which is in charge of all the communication procedures and provides all interfaces like sending, receiving, and broadcasting data.

The communication object:

- Establishes the communication between each other when the system starts up;
- Sends, receives, and broadcasts messages;
- Manages buffer and memory;
- And controls data stream and avoids deadlock.

The communication layer is discussed in detail in Section 4.3

4.2.6 Request object

A Request object is used for non-blocking calls. It is initialized when the user calls *ACS_Create_Request()*. After that, each time when the user calls a non-blocking function, the system will add one entry in the request object's entry table, which includes the non-blocking function's name, parameters, and a field for the result. After the user calls *ACS_Submit()* (*is this the right call, I thought it was commit, not submit???*), the control process takes over the request object and executes the functions in the request object one by one. Because the control process exists as a thread in the host machine, the host program will not be blocked.

4.2.7 Group object

A Group object is created at the time a user calls *ACS_Create_Group()*. This object collects node objects and provides group operations. It does not, however, create any nodes. Instead, it records the nodes' IDs. For example, *ACS_ALL* is a global group object that represents all of the nodes in the system. It collects all of the nodes' IDs.

When the user wants to run the boards in all machines, instead of calling *ACS_Run()* for each of the nodes, he can simply call *ACS_Run(ACS_ALL)*.

4.3 Communication layer

The communication object is responsible for all communication issues in the ACS API.

The communication object uses the Message Passing Interface (MPI) for its low-level communication routines.

4.3.1 Establish communication

Recall that MPI has built-in features to support parallel computing. To invoke multiple processes among the network stations, a user needs to tell MPI the name, the number of processes, and the location of each program. MPI will then call these programs and establish the connection. The MPI program takes the command line arguments, which include the configuration file and the runtime setup.

```
local 0
10.0.0.8 1 \\tuba\home\mpi\controlprocess.exe
10.0.0.9 1 \\tuba\home\mpi\controlprocess.exe
10.0.0.7 1 \\tuba\home\mpi\controlprocess.exe
```

Figure 4.5: The configuration file used in WMPI

Figure 4.5 shows the configuration files in the Windows NT environment. The configuration files in other operating systems are similar. Each line of the configuration file consists of information for one process. The first column is the network address of a

machine. Here, the network address is an IP address. The second column is the number of processes to be run on that machine. The third column is the program's name along with the file path in that machine. In Figure 4.5, there are four machines in the system. The first is the host program, running on the host machine. The others are control processes, running on the specified machines.

4.3.2 Buffer Management

Incoming messages must be placed in the memory and passed to the control process or to the user program for processing. Meanwhile, when the control process or the user program generates output, it must be stored in the memory and passed to the communication object for transmission. The communication object accepts outgoing messages and passes incoming data to the higher layer, both in memory. The efficiency with which the communication object processes messages depends on how it manages the memory to hold the messages. A good design allocates space quickly and avoids copying data as packets move between layers.

Receiving message:

As shown in Figure 4.6, the communication object runs a typical message receiving cycle as follows:

1. Posts a receiving request to the system and waits for messages to come in;
2. Receives messages, storing them into the buffer;
3. Reads the messages and executes commands;
4. And releases the buffer and posts another receive request.

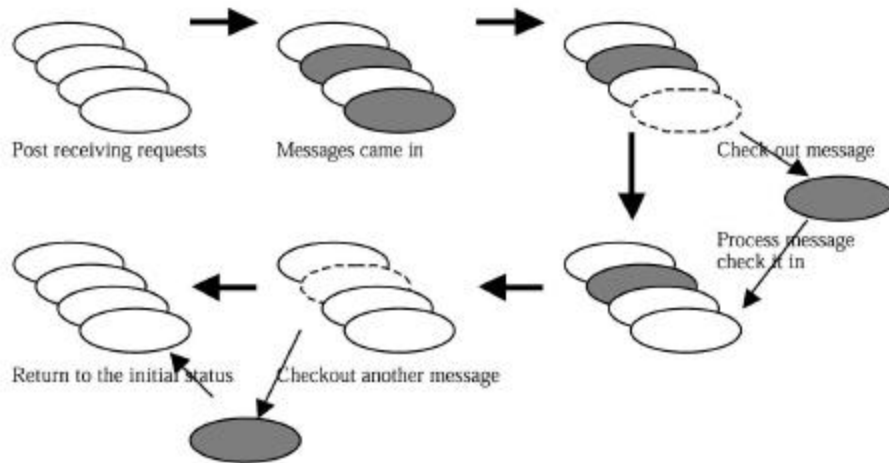


Figure 4.6: Procedures of receiving messages

When the messages come in, the communication object needs to provide buffers to store them. The fixed partitioning is a memory allocation method that lets the main memory be divided into a number of same size partitions. Each partition is for storing one message. A buffer table is used to record the buffer usage. Although it is easy to implement, the weakness is the inefficiency of using memory due to the unpredicted length of the message. For example, if 256 bytes are used for each buffer, messages of only a few bytes will result in poor memory usage.

Dynamic allocation can overcome some of the difficulties associated with fixed allocation. With dynamic allocation, each incoming message is allocated exactly the size it needs. However, the weakness is that it is difficult to maintain the fragmented memory. Extra overhead is needed when compacting the fragment memory.

To solve these problems, the ACS API classifies the messages into several types based on their length. There are a total of five types of messages, which are in length between 1 to 16, 16 to 256, 256 to 4096, 4096 to 65536 and 65536 to 4M bytes respectively.

Accordingly, the communication object has five types of buffers, which have 16, 256, 4096, 65536 and 4M bytes, respectively. Each type of buffer has several copies. A particular size of message can only go into a particular size of buffer, based on its length.

As Figure 4.7 shows, if an incoming message is smaller than 16 bytes, then the communication object uses 16 bytes of buffer to receive it. If it is larger than 16 but smaller than 256, then it uses 256 bytes of buffer to receive it. This process is followed to assign messages to buffers efficiently.

An MPI tag can make the message go into a buffer of the correct size. MPI uses a tag to match a pair of sending and receiving messages. For example, the first site posts a receiving request with Tag A. If the second site wants to send a message to the first one, it must send the message with the same tag; otherwise, even if the destination is correct, the first site cannot get that message due to the mismatch of the tag. In the ACS API implementation, the communication object posts forty receiving requests at the same time. These forty requests are divided into five equal groups. Each group uses a particular size of buffer and tag. If one communication object is going to send a message to another, it checks the size of that message, and attaches it to the corresponding tag for that size, i.e. if the message is smaller than 16 bytes, it tags to the 16-bytes buffer; if the message is larger than 16, but smaller than 256 bytes, it tags to the 256. Thus, only the receiver tagged for 256 will get the message.

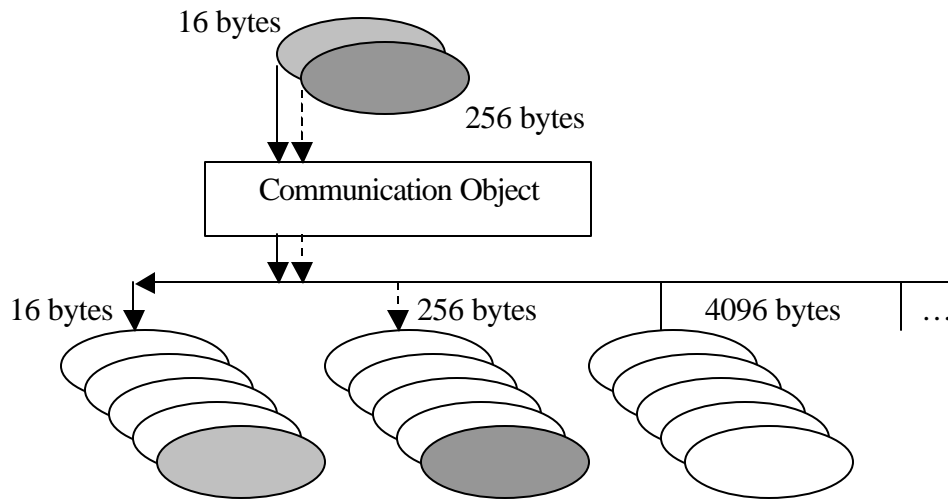


Figure 4.7: Messages will go into a right buffer size in the communication object

Buffer Pool data structure

Each buffer consists of four parts: the Buffer Header, Message Header, Command Header, and the data part. The Buffer Header includes buffer ID, tag, the size of the buffer, and a field to indicate whether this buffer is in use. The Message Header includes the destination address, source address, and sequence number. The Command Header includes command ID, source/destination node ID, message type, data length, and others.

Figure 4.8 below shows the buffer format and data structure.

Except for the in-use indicator, buffer header information is filled when initializing the communication object. The in-use indicator is changed each time the buffer is checking-in or checking-out. After checking out the buffer, the user program or the control process fills out the command header information. Then the buffer will be passed to the

communication object. The communication object then fills out the message header and sends out the message.

Buffer Check-In and Check-Out

The buffers are recycled. After sending or receiving messages, the buffer is returned to the communication object. This is done by buffer check-in and check-out procedures.

That is, when sending messages, the communication object checks out a suitable buffer size. After sending out the message, it checks the buffer back into the buffer pool. The receiving procedure is similar.



```

typedef struct {
    int    tag;                // buffer tag, to distinguish
                                // different size of buffer
    int    buffer_id;         // buffer id
    int    in_use;           // whether the buffer is in use
    void * internal_buffer;   // a pointer to it's buffer
    int    internal_buffer_size; // size of the buffer
} ACS_BUFFER_HEADER;

typedef struct {
    int    comm_des;         // network address used by comm obj
    int    comm_src;        // network address used by comm obj
    int    msg_seq;         // message sequence
} ACS_MESSAGE_HEADER;

typedef struct {
    int    command;
    int    id;              // command id
    int    des;             // destination node/channel id
    int    src;             // source node/channel id
    int    type;            // blocking/non-blocking
    int    obj;             // communication object: node or channel
    int    data_length;     // the data's lenght in the buffer
    void * external_buffer; // user space buffer
    int    external_buffer_size; // user space buffer size
    int    return_value;    // return value
} ACS_COMMAND_HEADER;

```

Figure 4.8: A message's format and data structure

Sending message

There are two situations to consider when sending a message.

1. Message send only – user sends commands or data to a remote node (i.e. write memory to remote node, send configuration files to remote node);
2. Message send and receive – user sends a command to a remote node to retrieve some data. This procedure consists of two steps, sending a request to a remote site and waiting for a reply (i.e. read memory from remote nodes, or a blocking call such as running a board, synchronization, etc.).

In the first case, a user or control process checks out a buffer based on the size of the command, fills out the command header, and copies the data to the buffer. It then passes the buffer to the communication object. The communication object fills out the message header, sends it out, and then checks in the buffer.

The second scenario includes two steps: sending the command to the remote site and waiting for the response back. The first step is straightforward. In the second step, after the communication object sends the message, the object posts an extra receiving request with a buffer from user space rather than from the buffer pool. It uses a different TAG with normal receiving. When the receiver sends the message back, it also uses that special TAG so that the incoming message will go directly into the user space. This saves the time of copying memory from the buffer pool to the user space. Figures 4.9 and 4.10 illustrate the buffer status changes in sending messages.

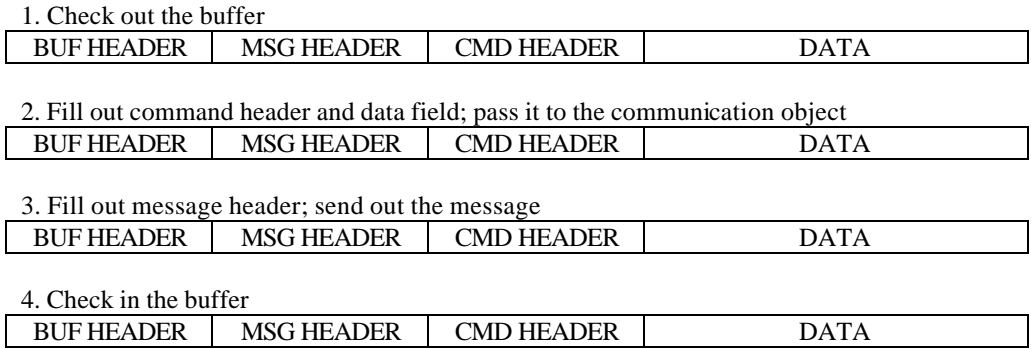


Figure 4.9: The buffer status in sending messages

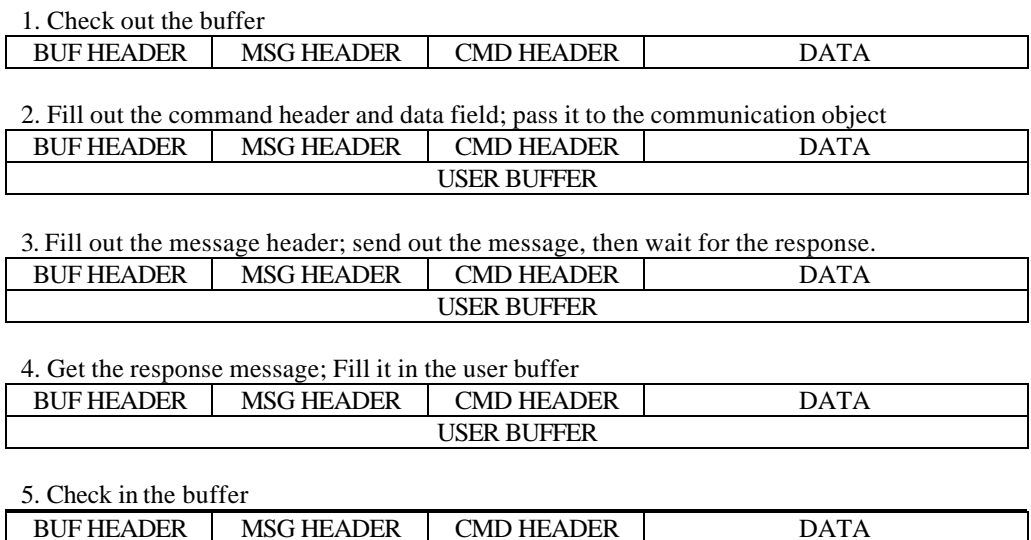


Figure 4.10: The buffer status in sending-receiving messages

4.3.3 Flow Control

The channel is a persistent data link between two nodes. Sometimes the data rates are so fast that the transmission needs to be controlled. For example, one site sends an overwhelming number of messages to the other site. Since the communication object processes the messages one by one, unprocessed messages will have to wait in the queue.

If the sender continually sends messages, and the receiver processes messages slower than they are sent, either some messages have to be dropped, or the sender has to be blocked. Flow control is used here to control the speed of sending and receiving messages.

TCP uses window-based flow control to handle network delay, throughput, and packet loss. When the TCP process on the receiving machine sends an acknowledgement, it includes a window advertisement in the segment to tell the sender how much buffer space the receiver has available for additional data. The window advertisement always specifies the data that the receiver can accept. When the sender fills the advertised window, the value in the acknowledgement field increases and the value in the window field may become smaller until it reaches zero. TCP uses window advertisements to control the flow of data across a connection. A receiver advertises small window sizes in order to limit the data that a sender can generate. In the extreme case, advertising a window size of zero halts transmission altogether[12].

The ACS API uses a similar concept to control the data rate between two nodes. The channel object keeps an integer called a buffer window. It represents the maximum number of messages that can be sent to the destination node. When the channel object sends a message to the destination node, the buffer window decreases by one. When the buffer window reaches zero, no further messages will be sent to that site. In the remote site, when the remote node gets a message and puts it into the FIFO, it acknowledges the sender by sending back an ACK message. Upon receiving the ACK, the channel object increases the buffer window by one. So the channel may continue sending messages to

the remote node. If the remote node gets the message but cannot put the data into the FIFO, i.e. the FIFO is full, then the remote node puts the data into its channel buffer without acknowledging the sender. In such situations, if the channel object continues sending data to the remote node, the buffer window will be reduced to zero because there is no acknowledgment from the remote node. Thus, no further messages will be sent. And in the remote site, the remote node has put all the received, but not processed messages, into its buffer. The number of channel buffers in the remote node is the same as the buffer window in the channel. Conversely, when the channel buffer in the remote node is full, the buffer window in the channel object is zero and sending data is blocked until the remote node sends an acknowledgement back to the channel object. After the FIFO is empty, the remote node will move the data from its channel buffer into FIFO. It then acknowledges to the channel object, allowing more data to be sent. Figures 4.11 and 4.12 illustrate how the channel buffer is used for data flow control.

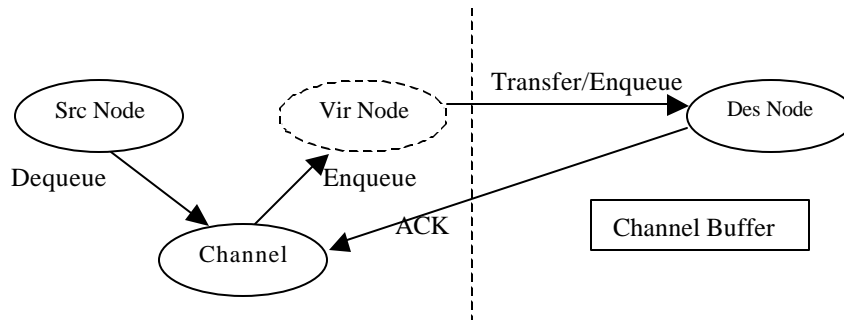


Figure 4.11: The destination node FIFO is not full and data is enqueued directly to the destination node.

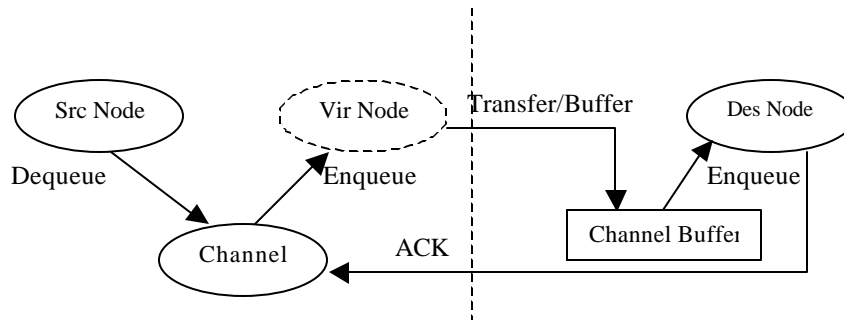


Figure 4.12: The destination node FIFO is full and data is put into the destination node's buffer. After the FIFO is available to enqueue more data, the destination node extracts data from the channel buffer and puts it into the FIFO. It then acknowledges this to the channel.

4.3.4 Message ordering and serialization

Because the communication object will receive multiple messages at the same time, it is important to process the message in the sequence of messages sent. For example, one sends out two messages for setting the frequency of the board and then for running it. If the communication object executes the run board command before setting the board up, an error will occur.

The API uses a sequence number in each message's header to indicate the order in which the message is received. The communication object keeps a table for each site to record what the next message's sequence is from that site. If an incoming message has an unexpected sequence number, the communication object will suspend it and wait for the correct message.

Because each site keeps its own sequence number table, there is no global sequence number. Although using individual sequence numbers can keep the messages from the same site in sequence, it cannot guarantee that the messages sent by different sources keep their original order. For instance, when site A and site B send two commands respectively to site C, there will be a total of six possible sequences in which site C may receive these messages.

Fortunately, the ACS API is a simple parallel application. When the program is running, most of the network traffic is for passing channel data. Because the FIFO in the reconfigurable board is usually designed to process a single source of data, one assumes that each FIFO in a node will not be linked by more than two channels. Thus, although a board can have multiple channels to connect, receiving messages from different channels will not cause any conflicts. The messages can be distinguished by the FIFO name.

Another situation is when the host machine sends commands to other machines. That typically occurs during the setup or configuration time. The host machine will communicate with multiple machines at the same time. However, most of the messages are send-only. That means the host machine sends messages without acknowledgement. Therefore, no machine in the system will receive messages from more than two sources at the same time. Those messages which require a reply, such as *ACS_Read()* and *ACS_Run()*, are sent in a blocking fashion. That means the host program will not send further messages until it gets a reply from the machine to which it sent the message. Thus, all acknowledged messages received in the host machine are serialized and do not conflict with each other.

4.4 Control Process

The Control Process is not accessible by the user. Still it is an integral part of the ACS API. The control process runs on each PC and is responsible for all nodes and channels on that PC. It accepts commands from the host program or other control processes to control the nodes.

As Figure 4.13 shows, the control process runs an infinite loop to perform various tasks continually until the program is terminated. It checks the incoming messages, interprets them, and manipulates the board based on the commands. It drives the channel to read data from the source node and send it to the destination node, and maintains a request object.

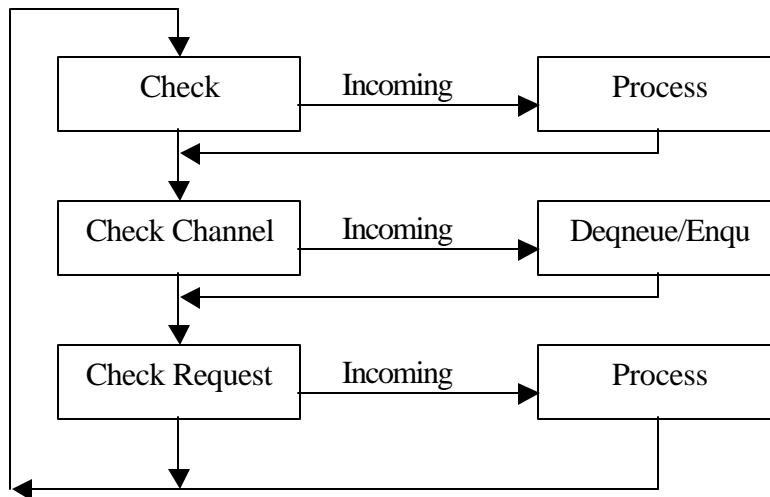


Figure 4.13: Control Process Loop

4.5 Node/Channel Management

Node and channel objects are managed by the system object and the control process.

Both the system object and control process maintain two arrays: one is for nodes and the other is for channels.

To add a node into the system object, the user needs to call *ACS_System_Create()* or *ACS_AddNode()*. The node information is specified in the ACS_NODE data structure.

Upon receiving a request, the system will register the node to the world object. Because the world object is a global object, the information about the node is broadcast to all sites. Therefore, all of the control processes know where the node resides.

If the node is on the host machine, the system object creates a node object and puts it into its node array. If the node is on the remote machine, then the system object creates a virtual node object in its node array and sends a command to the corresponding control process on which that node should reside. That control process then adds a node object to its node array.

To add a channel into the system object, the user needs to call *ACS_System_Create()* or *ACS_AddChannel()*. The channel information is specified in the ACS_CHANNEL data structure, which includes source node ID, the source port, destination node ID, destination port, and other properties such as the maximum size of data to dequeue from the FIFO and the channel buffer size in ACS_CHANNEL. If the user does not specify the channel information, a default value will be assigned. Similar to adding a node object, the

channel object needs to be registered at the world object so that all control processes will know where the channel resides.

Based on the type of source and destination, the channel has three types:

- **Host to node** – This type of channel is used as the beginning of a channel link. The source endpoint of the channel is a “host port.” When the user puts data into that “port,” it will go through that channel. The system may have several of these types of channels so as to provide multiple channel links. This type of channel is only used in the host machine.
- **Node to node** – This type of channel links two nodes. The channel object is connected with one node object as a source node and a virtual node object as the destination node.

If the source node is on the remote machine and the destination node is also on the remote machine, the channel object needs to create a virtual node to represent that destination node. In this case, although the host machine already has a virtual node object for that destination node, the system needs to create another virtual node object in the channel source node machine. Thus, there will be more than one virtual node object linking to one node object in the system. Figure 4.14 shows this situation.

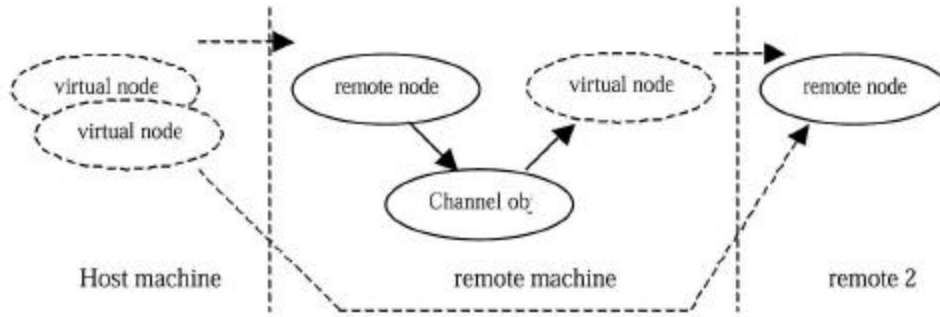


Figure 4.14: The remote node on the remote machine two has two copies of virtual node objects.

- Node to Host** – This type of channel is used for the endpoint of a channel link. When users create such a channel and point it to a host “port,” the system will create a dummy node automatically. That dummy node is used as an extra buffer to receive returned data. Therefore, a channel linking a node and a host is actually between a computing node and a dummy node. Figure 4.15 shows this.

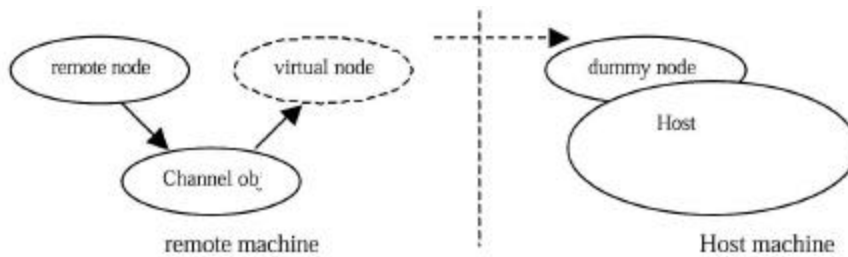


Figure 4.15: This type of channel is from a node to a host.

Deleting a Channel

If the channel to be deleted is in the host machine, then the system object will remove it from the channel array. If the channel is on a remote machine, then the system object

sends a command to the corresponding control process. That control process removes the channel object from its array and deletes the virtual node object linked to it.

4.6 Cache Management & Run-time Reconfiguration

In some cases, the node (FPGA board) needs to be configured at run-time. The user will send different configuration files to the board in order to change its functionality. Often, the usage pattern of configuration files demonstrates temporal locality. To reduce the network transmission time, the configuration files are cached in the remote site. If on the second occasion, the user tries to send old configuration files, the system will just send the configuration file's header rather than the data to the remote site.

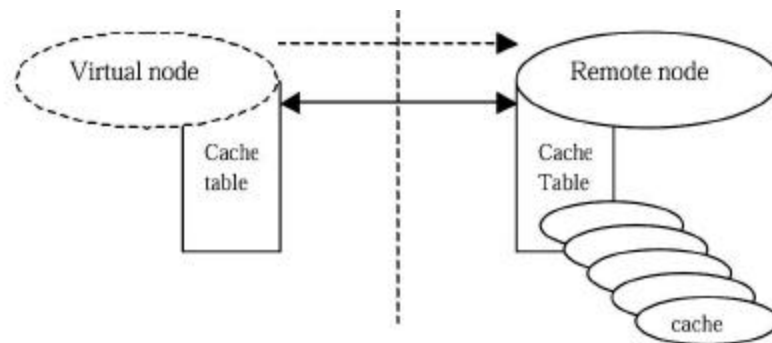


Figure 4.16: Cache configuration file in the remote machine to reduce the network transmission

As Figure 4.16 above shows, the virtual node in the local site and the node in the remote site both maintain a cache table. The two tables are identical in content, but only the remote node caches the actual data. When a new configuration file comes in, the virtual

node searches its cache table to see if this configuration file has been cached. If it has, the virtual node will not send any actual data but only the configuration file head. In the remote site, if the node gets a configuration message, it checks the configuration file's ID with those in the cache. Because the two cache tables are identical, the remote node will get the cached data in its buffer. If the virtual node did not find the configuration file's ID in its table, by default it uses a replacement algorithm to put the configuration files' header in its cache table. It then sends out the whole configuration file. The remote node then uses the same replacing algorithm to save both the configuration file's header and the data into the cache.

Explicit and implicit control of the cache

The user is responsible for the selection of a configuration ID. If a user sends two different configuration files with the same ID, the system will consider them as the same. The user may have the option to explicitly control the cache by specifying the cache slot in the configuration file's header.

4.7 OS Independent Layer

The ACS API was written in standard C/C++ language. Most of the code can be compiled directly under the Windows NT or Linux environment. However, some sections of the code depend on different operating systems. For example, Windows NT uses *_beginthread* to create a thread. While in Linux, the function is *_pthread_create*. Using the critical section is also different in the two operating systems.

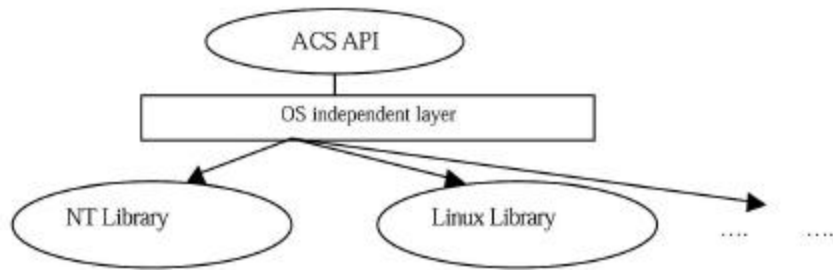


Figure 4.17: OS Independent layer

An OS independent layer has been inserted between the API and the system call. As Figure 4.17 shows, instead of calling the system routine directly, the ACS API calls the OS independent routine. For example, the function `_createthread` takes the place of `_beginthread` or `_pthread_create`. The implementation of `_createthread` varies by operating system.

Through the OS independent layer, the source code of ACS API can remain unchanged. To change an operating system, the developer just needs to provide the implementation of the OS independent layer.

4.8 Summary

This chapter discussed the detailed implementation of the API, and told the user how the API was implemented to be scalable, portable, and extendible. Appendix A3 shows the objects class header files used in the API. The next chapter will discuss how to test the API and analysis the API performance.

5. Testing APIs

5.1 Overview

The ACS API has been tested and analyzed by running various test applications on the Virginia Tech Tower of Power platform. This chapter describes the test procedures and analyzes the API performance. The API has been tested under three different environments: Windows NT with 100 Mbps switched Ethernet, Linux with 100 Mbps switched Ethernet, and Linux with 1.2 Gbps switched Myrinet. All tests used the Wildforce board. The version of MPI used for Windows NT is WMPI 1.2, MPICH 1.2.0 for Linux with switched Ethernet, and MPICH_GM 1.1.2.13 for Linux with Myrinet. Based on the API functions, the test programs have been divided into five categories. Section 5.2 discusses the testing programs and analyzes the performance of the API. Section 5.3 categorizes all the API functions that were involved in the test programs into a table.

5.2 Test Programs and Performance Results

5.2.1 Memory Access Function Test

The program listed in Appendix A2.1 tests the memory access functions, *ACS_Read()* and *ACS_Write()*. The program tests the memory access functions both in local and remote machines. The program calls *ACS_Write()* to write various sizes of data into the board, local and remote. It then reads data back from the boards through *ACS_Read()*. After that, the program compares the data written with the data that was read to ensure

the operation was performed correctly. Tables 5.1 and 5.2 show the average time for accessing 4 bytes to 1 Mega-byte of data in local and remote boards. Table 5.1 gives the results from Linux environment, and Table 5.2 from Windows NT. The memory-access rate and network data transfer rate are calculated by the equations below. Because the memory accessing includes both reading and writing, the rate is doubled.

$$Access_Rate = \frac{Data_Size \times 2}{Access_Time}$$

$$Data_Transfer_Rate = \frac{Data_Size \times 2}{(Remote_Board_Access_Time - Local_Board_Access_Time)}$$

The average memory-access rate for the local board in the Linux environment is about 9.3 MB/s. The access rate for the remote board is 3.7 MB/s in the switched Ethernet, and 5.4 MB/s in the Myrinet. The difference is primarily from the overhead of transferring data through the network. Other overheads are from the latencies in reading and writing data into the memory. When the data size is very small, these overheads are very apparent. As shown in Table 5.1, the access rate is only 0.031 MB/s for the local board and even less for the remote board when the data size is 4 bytes. The results also showed that using a high-speed network could significantly improve the system performance.

The results from the Windows NT with switched Ethernet environment are much better than in the Linux. The average memory-access rate for the local board is about 64 MB/s, which is almost six times larger than in the Linux environment. The average memory-access rate for the remote board is about 6.4 MB/s, which is also larger than in the Linux environment.

Table 5.1: Average memory access time in Linux environment

Data Size (bytes)	Local Board		Remote Board in 100Mbps Switched Ethernet		Remote Board in 1.2Gbps Switched Myrinet	
	Access Time (s)	Access Rate (MB/s)	Access Time (s)	Access Rate (MB/s)	Access Time (s)	Access Rate (MB/s)
4	0.000251	0.031872	0.042591	0.000186	0.016468	0.000484
132	0.011041	0.023910	0.013815	0.019110	0.039479	0.006686
260	0.009086	0.057230	0.011905	0.043676	0.020014	0.025980
388	0.008062	0.096258	0.010853	0.071500	0.001508	0.514480
516	0.006130	0.168352	0.010906	0.094628	0.001440	0.716662
644	0.004196	0.306980	0.012923	0.099666	0.001465	0.879178
772	0.002276	0.678282	0.011033	0.139938	0.001476	1.046062
900	0.000357	5.037314	0.011109	0.162030	0.001479	1.216752
102400	0.016783	12.202824	0.052144	3.927610	0.038432	5.328898
204800	0.028370	14.437786	0.094570	4.331168	0.077488	5.285978
307200	0.044703	13.743942	0.165127	3.720764	0.114831	5.350458
409600	0.068220	12.008268	0.230258	3.557744	0.147832	5.541438
512000	0.112960	9.065182	0.284365	3.601006	0.194632	5.261220
614400	0.114396	10.741666	0.349091	3.520002	0.204280	6.015262
716800	0.171892	8.340120	0.391599	3.660888	0.255927	5.601604
819200	0.179795	9.112618	0.458176	3.575916	0.301366	5.436586
921600	0.214502	8.592926	0.494910	3.724312	0.340540	5.412576
1024000	0.220141	9.303114	0.556146	3.682484	0.379426	5.397632

One explanation for the difference is that the network transfer rate in Windows NT is higher than in Linux. However, examination of the actual network data transfer rates through the equation above reveals that the data rate under Windows NT is close to the data rate under Linux with switched Ethernet but even slower than the Linux with Myrinet. The network data transfer rate under Linux with switched Ethernet is 6.1 MB/s, 12.8 MB/s under Myrinet, and 7.8 MB/s under Windows NT with switched Ethernet. The correct explanation is the DMA read and write in Linux is much slower than in Windows NT. This happened in both reading/writing memory and in the FIFO operation. Thus, although the network data transfer rate is the Myrinet is high, the Linux environment has low performance.

Table 5.2: Average memory accessing time in Windows NT environment

Data Size (bytes)	Local Board		Remote Board	
	Access Time (s)	Throughput (MB/s)	Access Time (s)	Throughput (MB/s)
4	0.000630	0.012680	0.003782	0.002114
132	0.000342	0.771160	0.002397	0.110122
260	0.000329	1.580540	0.018572	0.027998
388	0.000329	2.356260	0.002470	0.314126
516	0.000332	3.111540	0.002392	0.431438
644	0.000332	3.879500	0.002485	0.518308
772	0.000326	4.736180	0.002384	0.647560
900	0.000396	4.541620	0.002376	0.757574
102400	0.003398	60.276661	0.041407	4.945984
204800	0.006513	62.892818	0.083702	4.893550
307200	0.009832	62.491940	0.103384	5.942874
409600	0.013147	62.310799	0.117353	6.980628
512000	0.016134	63.468460	0.158898	6.444400
614400	0.020586	59.691059	0.198057	6.204284
716800	0.021722	65.997597	0.207016	6.925058
819200	0.024627	66.528603	0.245347	6.677888
921600	0.027513	66.992981	0.273556	6.737934
1024000	0.030177	67.866997	0.291504	7.025624

5.2.2 Stream Data Function Test

The stream data test programs listed in Appendix A2.2 test the *ACS_Enqueue()* and *ACS_Dequeue()* functions. They also test the board management functions, including *ACS_Configure()*, *ACS_Reset()* and *ACS_Run()*. Like the steps shown in Figure 3.2 and 3.3, the test program links the nodes into a ring and uses board management functions to configure the board, start the clock, and send the RESET signal to the board. It then calls *ACS_Enqueue()* to put data into the system object. The data should pass through each board in the ring automatically and finally go back to the system object. The test program then calls *ACS_Dequeue()* to get the data back from the system object.

The performance of the data stream functions is measured in two ways by this program: the round trip time (RTT) and the throughput. The RTT is the total time for putting data into the system, passing through and processing data in each board, and getting data back from the system. Obviously, the RTT varies by the number of the boards in the system. The program in Appendix A2.2a tests the RTT in one to eight machines in both the Windows NT and Linux environments. The RTT values are listed in Table 5.3. They are based on the average of 1000 repetitions of transmitting 1024 bytes data through the system.

Table 5.3: Average round trip time to pass through 1024 bytes data to system

Number of Boards	Average RTT in Win NT with Ethernet	Average RTT in Linux with Ethernet	Average RTT in Linux with Myrinet
1	0.00038	0.002255	0.002490
2	0.00284	0.020193	0.004389
3	0.00414	0.034057	0.011979
4	0.00953	0.052175	0.015276
5	0.01083	0.070366	0.021949
6	0.01355	0.097696	0.023934
7	0.01688	0.123697	--
8	0.02220	0.154830	--

Taking the results from Windows NT, as shown in Table 5.3, the RTT was increased approximately by 0.002 ~ 0.004 seconds for each additional board in the system. The results from the Linux also show a similar linear increasing. Because there is no network transaction in a one-board system, the RTT is much lower than in the others. Table 5.3 shows a big increase in time from a one-board to a two-board system. The RTT in Linux is about five times larger than in Windows NT as shown in the first line of Table 5.3. This is because of the low efficiency of the DMA access in the Linux environment. This affected the entire system as shown in Figure 5.1, the slope of Linux with switched

Ethernet is much steeper than the Windows NT. However, with the compensation from the high-speed network, the curve for Linux with Myrinet is close to that for Windows NT.

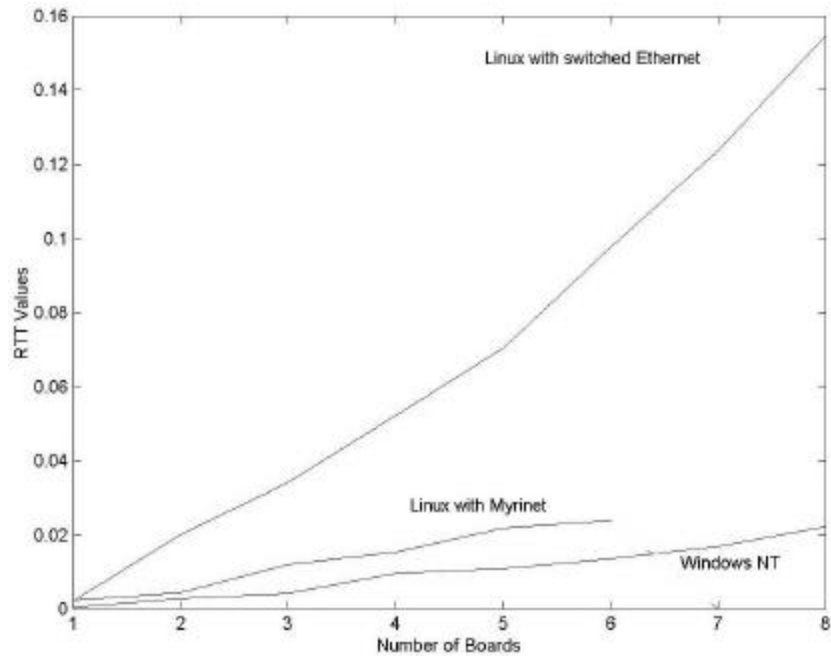


Figure 5.1: RTT Values with the number of boards in the system

Appendix A2.2b contains the program for testing system throughput. It differs from the RTT test program in that it creates a thread to dequeue the data instead of calling *ACS_Dequeue()* in the host program. The host program then enqueues five megabytes of data into the system and records the total time elapsed. Table 5.4 lists the average system throughput tested in the three environments. As seen from the results, the throughput for Windows NT and for Linux with Myrinet remains constant as the network size increases. This is essential to the scalability of the ACS system. That means the user can increase

the power of the ACS system by adding more processing elements while at the same time keeping the same data processing speed.

Table 5.4: System Average Throughput in different number of boards

Number of Boards	Average Throughput In Windows NT with Ethernet	Average Throughput In Linux with Ethernet	Average Throughput In Linux with Myrinet
1	$2.5 * 10^6$	191242	227607
2	563834	71088	220951
3	567826	45079	224177
4	547496	33253	228139
5	575025	28865	224794
6	572278	22955	--
7	622225	23440	--
8	586686	18163	--

When testing system throughput, the data rate in the system is fast because the data continues to pass from one node to another. The log file in this test shows that the flow control effectively controlled the data rate. When data generates too quickly from the first node, as shown in Figure 5.1, the channel window rapidly drops to zero, thus blocking the first node. Meanwhile, in the second node, since the FIFO is full, the data has been saved into the channel buffer. After FIFO is available, as shown in Figure 5.2, line 10, it pulls data out from the buffer and puts it into the FIFO. It then acknowledges it to the channel in the host machine (Figure 5.2, line 11). In the first machine, when the channel gets the acknowledgement, it increases the channel window by one and continues to send data to the second node (Figure 5.1, lines 8 to 9).

```

1. ...
2. Feed 1024 bytes to remote site, channel windows size shrunked to 5
3. Feed 1024 bytes to remote site, channel windows size shrunked to 4
4. Feed 1024 bytes to remote site, channel windows size shrunked to 3
5. Feed 1024 bytes to remote site, channel windows size shrunked to 2
6. Feed 1024 bytes to remote site, channel windows size shrunked to 1
7. Feed 1024 bytes to remote site, channel windows size shrunked to 0
8. Got ack, windows size is 1 now
9. Feed 1024 bytes to remote site, channel windows size shrunked to 0
10. Got ack, windows size is 1 now
11. Feed 1024 bytes to remote site, channel windows size shrunked to 0
12. ...

```

Figure 5.2: Log file in host machine when testing large stream data

```

1. ...
2. FIFO is full, saved data to buffer, #4
3. FIFO is full, saved data to buffer, #5
4. FIFO is full, saved data to buffer, #6
5. FIFO is full, saved data to buffer, #7
6. FIFO is full, saved data to buffer, #8
7. FIFO is full, saved data to buffer, #9
8. FIFO is full, saved data to buffer, #10
9. ...

10. Pull data out from the buffer
11. ack to channel 1
12. Got ack, windows size is 1 now
13. Feed 1024 bytes to remote site, channel windows size shrunked to 0
14. Pull data out from the buffer
15. ack to channel 1
16. Got ack, windows size is 1 now
17. Feed 1024 bytes to remote site, channel windows size shrunked to 0
18. Pull data out from the buffer
19. ack to channel 1
20. Got ack, windows size is 1 now
21. Feed 1024 bytes to remote site, channel windows size shrunked to 0
22. Pull data out from the buffer
23. ack to channel 1
24. ...

```

Figure 5.3: Log file in remote machine when testing large stream data

5.2.3 Non-blocking Function Testing

The non-blocking test program in Appendix A2.3 is revised from the memory access test program. The first part of the program uses normal blocking functions *ACS_Write()* and *ACS_Read()* for accessing the memory in the local and remote boards. The second part

illustrates how the user takes advantage of using non-blocking functions to reduce the total execution time by overlapping the computation and communication. In that part, the program first calls the non-blocking functions for accessing the memory in the remote board. It then calls *ACS_Commit()* to execute the requests in the background. After that, the program uses the blocking functions for accessing the memory in the local board and waits for the non-blocking request to finish. Because accessing the memory in the remote board is executed concurrently with the access in the local board, the time for transferring data through the network is overlapped with the time for writing data into the local board. Comparing the first part, which uses blocking functions for accessing both the local and remote boards, the total execution time is reduced.

Because of the frequent context switching in the concurrent execution, using non-blocking functions introduces some extra overhead. The use of a large number of non-blocking calls should be carefully considered to ensure that the extra overhead does not counteract the time saved from concurrent execution. As the last line in Windows NT column in Table 5.5 shows, when more and more non-blocking functions were called, the total execution time became worse than if blocking functions were used.

Table 5.5: Using non-blocking functions reduces the total execution time.

Times of Accessing Memory in Remote Board	Windows NT with Switched Ethernet		Linux with Myrinet	
	Total Execution Time When Using Blocking Function	Total Execution Time When Using Non-blocking Function	Total Execution Time When Using Blocking Function	Total Execution Time When Using Non-blocking Function
5	3.222588	2.245843	12.492957	10.630059
10	4.662546	3.876939	14.655513	10.941236
15	6.278506	6.026721	16.395372	10.967671
20	7.731978	8.663711	18.280171	11.356604

Comparing the results from the NT, the total execution time with non-blocking functions in Linux did not increase significantly as the number of remote memory accesses increased. The total execution time with blocking calls in Linux did increase significantly. The reason is because the Linux version takes more time to access memory than the Windows NT version and the time used for accessing memory is much larger than the time for transferring data through network. Thus, the time for accessing remote memory was entirely overlapped with accessing the local board in the Linux environment.

5.2.4 Group Function Testing

This program is also revised from the data stream test program. Instead of using an individual node ID in calling the board management functions, it uses `ACS_ALL`, so the user only needs to call those functions once. The program is listed in Appendix A2.4.

5.2.5 Configuration Cache Testing

In this test, the host program sends three different configuration files to the remote node repeatedly. Since the configuration files have been cached in the remote machine the first time it was configured, the system will not send any actual configuration data to the remote machine in the later `ACS_Configure()` calls. As shown in Table 5.6, the times for reconfiguring the remote boards have been significantly reduced after the first call. The program is listed in Appendix A2.5.

Table 5.6: Time of reconfiguring board

Board Number	Time of first configuration	Time of second configuration
1	--	--
2	0.140349	0.031716
3	0.136511	0.031293
4	0.381436	0.032106

5.3 Test Table

All the APIs have been successfully tested. The Test Table shows the APIs involved in the above test programs.

Table 5.7: API Test Table

	Memory Access	Data Stream	Non-Block	Group Function	Config Cache	Other
<i>Support Functions</i>						
ACS_Version	X	X	X	X	X	
ACS_Initialize_Log					X	
ACS_World_Info	X	X	X	X	X	
<i>System Setup Function</i>						
ACS_Initialize	X	X	X	X	X	X
ACS_Finalize	X	X	X	X	X	X
ACS_System_Create	X	X	X	X	X	X
ACS_System_Destroy	X	X	X	X	X	X
ACS_Channel_Add						X
ACS_Node_Add						X
ACS_Is_Local						X
ACS_Is_Remote						X
<i>Memory Access Function</i>						
ACS_Read	X					
ACS_Write	X					

Board Management

Function

ACS_Configure		X		X	X	
ACS_Readback						X
ACS_Clock_Set		X		X	X	
ACS_Clock_Get						X
ACS_Run		X		X	X	
ACS_Stop						X
ACS_Reset		X		X	X	
ACS_Reset_Toggle						X
ACS_Interrupt						X
ACS_InterruptPoll						X
ACS_Reg_Read						X
ACS_Reg_Write						X

Data Stream Function

ACS_Enqueue		X	X	X	X	
ACS_Dequeue		X	X	X	X	

Group Access Function

ACS_Group_Create				X		
ACS_Group_Destroy				X		

Non-blocking Function

ACS_Request_Create			X			
ACS_Request_Destroy			X			
ACS_Commit			X			
ACS_Wait			X			
ACS_Test			X			
ACS_ReadN			X			
ACS_WriteN			X			
ACS_ResetN			X			
ACS_ConfigureN			X			

6. Conclusion and Future Work

6.1 Summary

In this research, a scalable API and runtime software system was designed and developed to support applications of the network distributed ACS system. This thesis work discussed in detail the design and implementation of the API, as well as the testing and analysis of the performance of the API implementation.

The latest version of the ACS API is 1.2. This release was tested and released on January 2000. It supports the SLAAC-1, SLAAC-2, RCM, and Wildforce boards. It uses MPI 1.1 for interprocessor communication. This API implementation has been tested in the Linux and Windows NT environments. The API specifications, implementation source code, and supporting documents can be accessed on Virginia Tech Configurable Computing lab web site at <http://www.ccm.ece.vt.edu/slaac/>.

6.2 Future Work

One of the issues mentioned in the previous chapters is that the control process takes a significant amount of CPU time to check the messages from the network card and the data from the computing devices. Compared with the CPU time, the message and data rates are very slow. Thus, most of the CPU time has been wasted on checking messages and data. The next step of the API implementation is to integrate the control process into the network and reconfigurable computing cards. Although the API and the host program

still need support from the operating system, the control process no longer needs support from the operating system and requires no CPU time.

Efficient support for run-time reconfiguration (RTR) in the current version of the API allows for the control processes to cache several configurations to reduce the network transfer requirements. This mode is called host-initiated RTR because the host process is always the initiator of reconfiguration. The next step is to develop a more complex data-driven RTR, in which the reconfiguration is driven by the data that is encountered. The host process cannot drive such reconfiguration efficiently.

Another step of the ACS API development is to integrate the existing version with other high-level tools. For example, JHDL provides a portable, integrated environment for programming a single adaptive computing board [7]. With JHDL, a user can write or debug applications on the FPGA board through a high-level host program. Because JHDL uses Java as its programming environment, it could be integrated with the ACS API. Such integration can further simplify the steps of writing applications in distributed adaptive computing systems [11].

References

- [1] D.Ridge, D.Becker, P.Merkey, and T.Sterling, "Beowulf: Harnessing the Power of Parallelism in a Pile-of-PCs," in *Proceedings, IEEE Aerospace*, 1997.
- [2] Virginia Tech Configurable Computing Lab, "Tower of Power," <http://www.ccm.ece.vt.edu/slaac/>, 1998
- [3] Annapolis Micro System, Inc., "Wildforce Reference Manual," *Annapolis Micro System, Inc.*, 1997
- [4] Myrinet, Inc., "Myrinet Reference Manual," *Myrinet, Inc.*,
- [5] Xilinx, Inc., "The Programmable Logic Data Book," *Xilinx Inc.*, 1993.
- [6] J.R.Armstrong and F.G.Gray, "Structured Logic Design with VHDL," *Prentice Hall*, 1993.
- [7] P. Bellows and B. Hutchings, "JHDL-An HDL for Reconfigurable Systems," *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April 1998.
- [8] Information Sciences Institute – East, "Systems Level Applications of Adaptive Computing", <http://www.east.isi.edu/SLAAC/>, 1998.
- [9] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard", *International Journal of Supercomputing Applications*, Vol. 8(3/4), 1994.
- [10] Allison L. Walters, "A Scaleable FIR Filter Implementation Using 32-bit Floating-Point Complex Arithmetic on a FPGA Based Custom Computing Platform", *Virginia Tech Electronic Thesis and Dissertation*. 1998

- [11] M. Jones, L. Scharf, J. Scott, C. Twaddle, M. Yaconis, K. Yao, P. Athanas, and B. Schott, "Implementing an API for Distributed Adaptive Computing Systems," *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines, Napa, CA, April, 1999.*
- [12] Duncan A. Buell, Jeffrey M. Arnold, Walter J. Kleinfelder, "Splash 2, FPGAs in a Custom Computing Machine", *IEEE Computer Society Press, 1996*
- [13] Douglas E. Comer, David L. Stevens, "Internetworking with TCP/IP, Volume II, Design, Implementation, and Internals", *Prentice Hall, 1994*
- [14] B. Schott, S. Crago, R. Parker, L. Carter, C. Chen, J. Czarnaski, M. French, I. Hom, T. Tho, and T. Valenti, "Reconfigurable Architectures for System-Level Applications of Adaptive Computing," *submitted to VLSI Design special issue on reconfigurable computing.*
- [15] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, Jack Dongarra, "MPI: The Complete Reference", *The MIT Press, 1997.*
- [16] Stephen M. Scalera, Jose R. Vazquez, "The Design and Implementation of a Context Switching FPGA", *FCCM, 1998.*

Appendix 1. The ACS API Specification

A1.1 Data Structures

TYPEDEF ACS_ADDRESS

Description	ACS_ADDRESS designates a memory location for read and write functions.		
Structure	Int	Pe	Processing Element ID number
	Int	Mem	Memory location of PE
	Int	Offset	Used to allow indirect addressing

TYPEDEF ACS_CHANNEL

Description	ACS_CHANNEL is a structure that allows the user to define a channel. An array of these structures is passed to <i>ACS_System_Create()</i> .		
Structure	Int	Src_node	Logical node number of data source
	Int	Src_port	Logical port number of data source
	Int	Des_node	Logical node number of destination
	Int	Des_port	Logical port number of destination
	Int	Window_size	Number of channel window used in flow control
	Int	Dequeue_size	Size of data dequeued from FIFO
	Int	Number	Serial number
	Void *	Dev	Reserved

TYPEDEF ACS_NODE_TYPE

Description	ACS_NODE_TYPE defines the different type of node	
enum	E_WF4	Represents Wildforce 4 board
	E_SLAAC1	Represents SLAAC-1 board
	E_SLAAC2	Represents SLAAC-2 board
	E_RCM	Represents RCM board

TYPEDEF ACS_NODE

Description	ACS_NODE is a structure that allows the user to define a type of node to be allocated by the system. An array of these structures is passed to <i>ACS_System_Create()</i> .		
structure	Int	Number	Serial number
	ACS_NODE_TYPE	Node_type	Type of node defined in the enum ACS_NODE_TYPE.
	Int	site	Logical network address
	Void *	Dev	reserved

TYPEDEF ACS_CLOCK

Description	ACS_CLOCK is a structure that allows the user to define the properties of a clock synthesizer on an ACS board.		
structure	Double	Frequency	Synthesizer clock frequency
	Int	Countdown	Countdown timer value
	Void *	Dev	reserved

TYPEDEF ACS_CONFIG

Description	ACS_CONFIG is a structure that contains information related to a configuration.		
structure	Int	Serial_no	Configuration file ID. System uses this ID to check files in cache
	Int	Mask	Indicates which fields to set or get
	Char [255]	Label	Configuration "name" string
	Void *	Bitstream	Configuration bit data
	Int	Count	Number of bytes in configuration data
	Int	Pe_mask	Destination address or FPGA number
	Int	Cache_slot	This field is used for explicit control of cache. Configuration file will go into designate cache slot

TYPEDEF ACS_STATUS

Description	ACS_STATUS allows for a more detailed error information to be returned.		
structure	Int	Code	Error code value
	Int	Severity	Severity
	Char [255]	Text	Text string for printing

TYPEDEF ACS_WORLD_INFO

Description	ACS_WORLD_INFO is a structure that contains global and runtime information for the ACS system.		
structure	Int	Site_number	Number of workstations
	Int	Board_available	Number of boards in the system
	Int	Pe_available	Number of PEs in each board

A1.2 Support Functions

int ACS_Version(char * ver);

Description:	Get current version of the API.			
Parameters:				
	OUT	char *	ver	
Return Value:	ACS_SUCCESS			

int ACS_Initialize_Log(char * log_path);

Description:	Set network path for log; All machines in LAN should be able to access that path			
Parameters:	IN	Char *	Log_path	
Return Value:	ACS_SUCCESS / ACS_FAILURE			

int ACS_World_Info(ACS_WORLD_INFO * world_info);

Description:	Get global data information; See ACS_WORLD_INFO Data structure			
Parameters:	IN			
	OUT	ACS_WORLD_INFO*	World_info	
Return Value:	ACS_SUCCESS			

A1.3 System Setup Functions

int ACS_Initialize(int *, char *, ACS_STATUS *);**

Description:	ACS_Initialize is responsible for parsing command line arguments, interpreting system environment variables, initializing global resources, and creating the ACS_World object; this function should be called exactly once prior to any other ACS function call.			
Parameters:	IN	Int *	Argc	command line parameters
	IN	Char ***	Argv	command line parameters
	OUT	ACS_STATUS *	Status	Get error status
Return Value:	ACS_SUCCESS / ACS_FAILURE			

int ACS_Finalize();

Description:	ACS_Finalize is the mirror of ACS_Initialize; it guarantees that allocated resources are properly freed. This includes both the system memory on the host, and allocated nodes on the network. This function should be called exactly once and the last ACS function called before exit.			
Parameters:	IN			
	OUT			
Return Value:	ACS_SUCCESS / ACS_FAILURE			

int ACS_System_Create(ACS_SYSTEM **, ACS_NODE *, int, ACS_CHANNEL *, int, ACS_STATUS *);

Description:	ACS_System_Create is responsible for allocating the nodes specified in the node list array and creating the channels specified in the channel list array. An ACS_SYSTEM object is created and the pointer is returned in the system handle. All processors must call this function collectively to allow for appropriate information sharing.			
Parameters:	IN	ACS_SYSTEM **	System	Handle to system object pointer
	IN	ACS_NODE []	Nodes	List of nodes to allocate
	IN	Int	Node_count	Number of nodes
	IN	ACS_CHANNEL[]	Channels	List of channels to allocate
	IN	Int	channel_count	Number of channels
	OUT	ACS_STATUS *	Status	Command status
Return Value:	ACS_SUCCESS / ACS_FAILURE			

int ACS_System_Destroy(ACS_SYSTEM * system, ACS_STATUS * status);

Description:	ACS_System_Destroy releases all nodes allocated by the system and frees all memory associated with the system object.			
Parameters:	IN	ACS_SYSTEM *	System	System to destroy
	OUT	ACS_STATUS *	Status	Command status
Return Value:	ACS_SUCCESS / ACS_FAILURE			

int ACS_Channel_Add(ACS_CHANNEL * channel, ACS_SYSTEM * system);

Description:	This function is dynamically adding a channel into the system. It appends the specified channel to the channel list in the system object.			
Parameters:	IN	ACS_CHANNEL *	Channel	Structure describing the starting and ending points and attributes of the channel to be created.
	IN	ACS_SYSTEM *	system	The system where to create this channel.
Return Value:	ACS_SUCCESS / ACS_FAILURE			

int ACS_Channel_Remove(int channel_num, ACS_SYSTEM * system);

Description:	This function removes a channel from a system.			
Parameters:	IN	Int	Channel_num	The logical channel number of the channel to be removed.
	IN	ACS_SYSTEM *	System	The system from which to remove this channel.
Return Value:	ACS_SUCCESS / ACS_FAILURE			

int ACS_Node_Add(ACS_NODE * node, ACS_SYSTEM * system);

Description:	This function allocates a node and appends it to the system object dynamically.			
Parameters:	IN	ACS_NODE *	Node	Structure describing the attributes of the node to be created
	IN	ACS_SYSTEM *	System	The system in which to create this node
Return Value:	ACS_SUCCESS / ACS_FAILURE			

int ACS_Is_Local(int node_num, ACS_SYSTEM *);

Description:	Returns ACS_SUCCESS if node described by node_num is in the host machine (local) else ACS_FAILURE is returned.			
Parameters:	IN	Int	Node_num	The logical node number
	IN	ACS_SYSTEM *	System	The system from which to query this node
Return Value:	ACS_SUCCESS / ACS_FAILURE			

int ACS_Is_Remote(int node_num, ACS_SYSTEM *);

Description:	Returns ACS_SUCCESS if node described by node_num is remote else ACS_FAILURE is returned.			
Parameters:	IN	Int	Node_num	The logical node number
	IN	ACS_SYSTEM *	System	The system from which to query this node
Return Value:	ACS_SUCCESS / ACS_FAILURE			

A1.4 Memory Access Functions

```
int ACS_Read(void * buffer, int count, int node, ACS_ADDRESS * address,
            ACS_SYSTEM * system, ACS_STATUS * status);
```

Description:	This function reads count bytes from the given address of the specified node of the system and places the data in the buffer.			
Parameters:	OUT	Void *	Buffer	Destination buffer for data
	IN	Int	Count	Number of bytes to read
	IN	Int	Node	Logical node number
	IN	ACS_ADDRESS *	Address	Physical address at node to read
	IN	ACS_SYSTEM *	System	System object
	OUT	ACS_STATUS *	Status	Command status structure
Return Value:	ACS_SUCCESS / ACS_FAILURE			

```
int ACS_Write(void * buffer, int count, int node, ACS_ADDRESS *
            address, ACS_SYSTEM * system, ACS_STATUS * status);
```

Description:	This function writes count bytes from the given buffer at the address of the specified system node.			
Parameters:	IN	Void *	Buffer	source buffer for data
	IN	Int	Count	Number of bytes to read
	IN	Int	Node	Logical node number
	IN	ACS_ADDRESS *	Address	Physical address at node to read
	IN	ACS_SYSTEM *	System	System object
	OUT	ACS_STATUS *	Status	Command status structure
Return Value:	ACS_SUCCESS / ACS_FAILURE			

A1.5 Board Management Functions

```
int ACS_Configure(ACS_CONFIG * config, int node, ACS_SYSTEM * system,
                ACS_STATUS * status);
```

Description:	This function downloads a configuration bit-stream to the FPGAs of a node			
Parameters:	IN	ACS_CONFIG *	Config	Configuration data
	IN	Int	Node	Logical node number
	IN	ACS_SYSTEM *	System	System object
	OUT	ACS_STATUS *	Status	Command status structure
Return Value:	ACS_SUCCESS / ACS_FAILURE			

```
int ACS_Readback(ACS_CONFIG * config, int node, ACS_SYSTEM * system,
                ACS_STATUS * status);
```

Description:	This function read back a configuration file from a FPGA board.			
Parameters:	OUT	ACS_CONFIG *	Config	Buffer for readback data
	IN	Int	Node	Logical node number
	IN	ACS_SYSTEM *	System	System object
	OUT	ACS_STATUS *	Status	Command status structure
Return Value:	ACS_SUCCESS / ACS_FAILURE			

```
int ACS_Clock_Set(ACS_CLOCK * clock, int node, ACS_SYSTEM * system,
                 ACS_STATUS * status);
```

Description:	This function sets the clock attributes of a device.			
Parameters:	IN	ACS_CLOCK *	Clock	Clock data
	IN	Int	Node	Logical node number
	IN	ACS_SYSTEM *	System	System object
	OUT	ACS_STATUS *	Status	Command status structure
Return Value:	ACS_SUCCESS / ACS_FAILURE			

```
int ACS_Clock_Get(ACS_CLOCK * clock, int node, ACS_SYSTEM * system,
                 ACS_STATUS * status);
```

Description:	This function gets the clock attributes of a device.			
Parameters:	out	ACS_CLOCK *	Clock	Clock data
	IN	Int	Node	Logical node number
	IN	ACS_SYSTEM *	System	System object
	OUT	ACS_STATUS *	Status	Command status structure
Return Value:	ACS_SUCCESS / ACS_FAILURE			

```
int ACS_Run(int node, ACS_SYSTEM * system, ACS_STATUS * status);
```

Description:	This function starts the board clock.			
Parameters:	IN	Int	Node	Logical node number
	IN	ACS_SYSTEM *	System	System object
	OUT	ACS_STATUS *	Status	Command status structure
Return Value:	ACS_SUCCESS / ACS_FAILURE			

```
int ACS_Stop(int node, ACS_SYSTEM * system, ACS_STATUS * status);
```

Description:	This function stops the board clock.			
Parameters:	IN	Int	Node	Logical node number
	IN	ACS_SYSTEM *	System	System object
	OUT	ACS_STATUS *	Status	Command status structure
Return Value:	ACS_SUCCESS / ACS_FAILURE			

```
int ACS_Reset(int node, ACS_SYSTEM * system, int pe_mask, int enable,
             ACS_STATUS * status);
```

Description:	This function either enables or disables a RESET signal on a node device.			
Parameters:	IN	Int	Node	Logical node number
	IN	ACS_SYSTEM *	System	System object
	IN	Int	Pe_mask	Mask used to designate processing elements affected by reset.
	IN	Int	Enable	If nonzero, reset signal is enabled, otherwise disabled
	OUT	ACS_STATUS *	Status	Command status structure
Return Value:	ACS_SUCCESS / ACS_FAILURE			

```
int ACS_Reset_Toggle(int node, ACS_SYSTEM * system, int pe_mask,
                    ACS_STATUS * status);
```

Description:	Sends a reset pulse to a node device.			
Parameters:	IN	Int	Node	Logical node number
	IN	ACS_SYSTEM *	System	System object
	IN	Int	Pe_mask	Mask used to designate processing elements affected by reset.
	OUT	ACS_STATUS *	Status	Command status structure
Return Value:	ACS_SUCCESS / ACS_FAILURE			

```
int ACS_Interrupt(int node, int pe, int number, ACS_SYSTEM * system,
                 ACS_STATUS * status);
```

Description:	This function causes an interrupt signal of the specified number to be sent to a node device. This function has no effect on WildForce nodes.			
Parameters:	IN	Int	Node	Logical node number
	IN	Int	Pe_mask	
	IN	Int	Number	Interrupt number
	IN	ACS_SYSTEM *	System	System object
	OUT	ACS_STATUS *	Status	Command status structure
Return Value:	ACS_SUCCESS / ACS_FAILURE			

```
int ACS_InterruptPoll(int node, int pe, int * result_mask, int number,
    ACS_SYSTEM * system, ACS_STATUS * status);
```

Description:	Checks the interrupt status of specified processing elements on a node device.			
Parameters:	IN	Int	Node	Logical node number
	IN	Int	Pe_mask	Mask used to designate processing elements affected by reset.
	OUT	Int *	Result_mask	Return a 0 when all processing elements designated by pe_mask have raised interrupt. Otherwise, gives mask of PEs that have not yet raised interrupts
	IN	Int	Number	Interrupt number
	IN	ACS_SYSTEM *	System	System object
	OUT	ACS_STATUS *	Status	Command status structure
Return Value:	ACS_SUCCESS / ACS_FAILURE			

```
int ACS_InterruptAck(int node, int pe, int number, ACS_SYSTEM * system,
    ACS_STATUS * status);
```

Description:	Acknowledges interrupts of specified processing elements on a node device.			
Parameters:	IN	Int	Node	Logical node number
	IN	Int	Pe_mask	Mask used to designate PEs ack'd
	IN	Int	Number	Interrupt number
	IN	ACS_SYSTEM *	System	System object
	OUT	ACS_STATUS *	Status	Command status structure
Return Value:	ACS_SUCCESS / ACS_FAILURE			

```
int ACS_Reg_Read(ACS_SYSTEM * system, int node_num, int reg_id,
    ACS_REGISTER *);
```

Description:	Read data from node register			
Parameters:	IN	ACS_SYSTEM *	System	System object
	IN	Int	Node	Logical node number
	IN	Int	Reg_id	Register id number
	IN	ACS_REGISTER *	Reg	Register data structure
Return Value:	ACS_SUCCESS / ACS_FAILURE			

```
int ACS_Reg_Write(ACS_SYSTEM * system, int node_num, int reg_id,
ACS_REGISTER *);
```

Description:	Write data into node's register			
Parameters:	IN	ACS_SYSTEM *	System	System object
	IN	Int	Node	Logical node number
	IN	Int	Reg_id	Register id number
	IN	ACS_REGISTER *	Reg	Register data structure
Return Value:	ACS_SUCCESS / ACS_FAILURE			

A1.6 Streaming Data Functions

```
int ACS_Enqueue(void * src, int size, int port, ACS_SYSTEM *,
ACS_STATUS *);
```

Description:	This function puts data into a specific host port number.			
Parameters:	IN	Void *	Src	Source buffer for data
	IN	Int	Size	Number of bytes to write
	IN	Int	Port	System port number to write
	IN	ACS_SYSTEM *	System	System object
	OUT	ACS_STATUS *	Status	Command status structure
Return Value:	ACS_SUCCESS / ACS_FAILURE			

```
int ACS_Dequeue(void * des, int size, int port, ACS_SYSTEM *,
ACS_STATUS *);
```

Description:	This function takes data from a specific host port number			
Parameters:	IN	Void *	des	Destination buffer for data
	IN	Int	Size	Number of bytes to read
	IN	Int	Port	System port number to read
	IN	ACS_SYSTEM *	System	System object
	OUT	ACS_STATUS *	Status	Command status structure
Return Value:	ACS_SUCCESS / ACS_FAILURE			

A1.7 Group Management

```
int ACS_Group_Create(ACS_GROUP ** group, int nodes[], int count,
    ACS_SYSTEM * system);
```

Description:	ACS_Group_Create is the ACS_GROUP object constructor. The nodes argument is an array of logical node numbers (integers) of length count. The array index defines the new logical node number in the new group context. Group creation can be nested (system can be an object of type ACS_GROUP).			
Parameters:	OUT	ACS_GROUP **	Group	Handle to return created group object.
	IN	Int []	Nodes	Array of logical node numbers
	IN	Int	Count	Length of the nodes array
	IN	ACS_SYSTEM *	System	System object
Return Value:	ACS_SUCCESS / ACS_FAILURE			

```
int ACS_Group_Destroy(ACS_GROUP * group);
```

Description:	ACS_Group_Destroy is the ACS_GROUP object destructor. Unlike ACS_System_Destroy, this function does not free allocated nodes or other resources associated with the system.			
Parameters:	IN	ACS_GROUP *	Group	Group object to be free
	OUT			
Return Value:	ACS_SUCCESS / ACS_FAILURE			

A1.8 Non-blocking function

```
int ACS_Request_Create(ACS_REQUEST ** request, int number);
```

Description:	ACS_Request_Create is the ACS_REQUEST object constructor. The number argument specifies the number of commands the request object can store.			
Parameters:	IN	ACS_REQUEST **	request	Handle to request object pointer
	IN	Int	Number	Number of command entries to allocate
Return Value:	ACS_SUCCESS / ACS_FAILURE			

int ACS_Request_Destroy(ACS_REQUEST * request);

Description:	ACS_Request_Destroy is the ACS_REQUEST object destructor. It frees the memory associated with a request object.			
Parameters:	IN	ACS_REQUEST *	Request	Object to destroy
	OUT			
Return Value:	ACS_SUCCESS / ACS_FAILURE			

int ACS_Commit(ACS_REQUEST * request, ACS_REQUEST_STAT * status);

Description:	ACS_Commit processes the list of commands in the request structure, and issues the commands to the nodes. After ACS_Commit returns, any output buffers may be reused (such as from an ACS_Write or ACS_Enqueue command). However, the input buffers (such as from an ACS_Read) are undefined.			
Parameters:	IN	ACS_REQUEST *	Request	Request to submit
	IN	ACS_SYSTEM *	System	System group on which to commit request
	OUT	ACS_REQUEST_STAT *	Status	Command status array
Return Value:	ACS_SUCCESS / ACS_FAILURE			

int ACS_Wait(ACS_REQUEST * request, ACS_REQUEST_STAT * status);

Description:	This function causes the thread to block until all commands have completed.			
Parameters:	IN	ACS_REQUEST *	Request	Request to submit
	IN	ACS_SYSTEM *	System	System group on which to commit request
	OUT	ACS_REQUEST_STAT *	Status	Command status array
Return Value:	ACS_SUCCESS – all commands in the request have been successfully completed. ACS_FAILURE – the request has been completed. However, some of the commands have returned an error condition. The programmer should check the status structure to determine which commands have failed.			

int ACS_Test(ACS_REQUEST * request, ACS_REQUEST_STAT * status);

Description:	This function tests the request object to determine if all commands have been completed. Compared with ACS_Wait, ACS_Test only tests, but does not wait until the request has been finished. It's a non-blocking call.			
Parameters:	IN	ACS_REQUEST *	Request	Request to submit
	IN	ACS_SYSTEM *	System	System group on which to commit request
	OUT	ACS_REQUEST_STAT *	Status	Command status array
Return Value:	ACS_SUCCESS – all commands in the request have been successfully completed. ACS_FAILURE – the request has been completed. However some of the commands have returned an error condition. The programmer should check the status structure to determine which commands have failed.			

int ACS_ReadN(ACS_REQUEST * request, void * buffer, int count, int node, ACS_ADDRESS * addr, ACS_SYSTEM * system, ACS_STATUS * status);

Description:	This function reads count bytes from the given address of the specified node in the system and places the data in the buffer.			
Parameters:	OUT	ACS_REQUEST *	Request	Request to append to
	OUT	Void *	Buffer	Destination buffer for data
	IN	Int	Count	Number of bytes to read
	IN	Int	Node	Logical node number
	IN	ACS_ADDRESS *	Addr	Address at node to read
	IN	ACS_SYSTEM *	System	System object
	OUT	ACS_STATUS *	Status	Command status structure
Return Value:	ACS_SUCCESS / ACS_FAILURE			

int ACS_WriteN(ACS_REQUEST * request, void * buffer, int count, int node, ACS_ADDRESS * addr, ACS_SYSTEM * system, ACS_STATUS * status);

Description:	This function writes count bytes from the given buffer at the address of the specified system node.			
Parameters:	OUT	ACS_REQUEST *	Request	Request to append to
	OUT	Void *	Buffer	Source buffer for data
	IN	Int	Count	Number of bytes to read
	IN	Int	Node	Logical node number
	IN	ACS_ADDRESS *	Addr	Address at node to read
	IN	ACS_SYSTEM *	System	System object
	OUT	ACS_STATUS *	Status	Command status structure
Return Value:	ACS_SUCCESS / ACS_FAILURE			

```
int ACS_ResetN(ACS_REQUEST * request, int node, ACS_SYSTEM * system,
              ACS_STATUS * status);
```

Description:	This function causes a RESET signal to be sent to a node device.			
Parameters:	OUT	ACS_REQUEST *	Request	Request to append to
	IN	Int	Node	Logical node number
	IN	ACS_SYSTEM *	System	System object
	OUT	ACS_STATUS *	Status	Command status structure
Return Value:	ACS_SUCCESS / ACS_FAILURE			

```
int ACS_ConfigureN(ACS_REQUEST * request, int node, ACS_CONFIG *
                  config, ACS_SYSTEM * system, ACS_STATUS * status);
```

Description:	This function send configuration bit stream to a node device.			
Parameters:	OUT	ACS_REQUEST *	Request	Request to append to
	IN	Int	Node	Logical node number
	IN	ACS_CONFIG *	Config	Configuration file
	IN	ACS_SYSTEM *	System	System object
	OUT	ACS_STATUS *	Status	Command status structure
Return Value:	ACS_SUCCESS / ACS_FAILURE			

Appendix 2. Testing code for ACS API

A2.1 Memory Access

/** This program is to test the memory access functions, ACS_Read() and ACS_Write(). The program tests the memory access function in both local and remote machines. The program calls ACS_Write() to write various sizes of data into the board, local and remote. It then reads data back from the boards through ACS_Read(). After that, the program compares the writing data and the reading data. */

```
#include <stdio.h>
#include <iostream.h>
#include "acs.h"
#include "CWildForce4Node.h"
#include "mpi.h"

#define MEMORY_SIZE          1024
#define PASS_THROUGH        10

_SYSTEM_TYPE(CWildForce4Node);

void Node_initialize(ACS_NODE nodes[], int sites);
void Channel_initialize(ACS_CHANNEL channels[], int sites);

void main(int argc, char ** argv)
{
    ACS_STATUS   status;
    char         version[0x10];

    // Get the SLAAC API version
    ACS_Version(version);
    cout<<"SLAAC APIs version "<<version<<endl;

    // Initialize the SLAAC system.
    cout<<"ACS Initializing...";
    ACS_Initialize(&argc, &argv, &status);
    cout<<"done"<<endl;

    ACS_WORLD_INFO wf;
    ACS_World_Info(&wf);
    int sites = wf.site_number;

    // Construct the network topology
    ACS_CHANNEL     channels[0x10];
    ACS_NODE        nodes[0x10];
    Node_initialize(nodes, sites);

    // Creating the ACS system...
    cout<<"Creating system...";
    ACS_SYSTEM     * system;
    ACS_System_Create(&system, nodes, sites, channels, 0, &status);
    cout<<"done. node number: "<<sites<<endl<<endl;

    cout<<endl<<endl<<endl<<"Now testing Memory function..."<<endl;
```

```

char in_buffer[MEMORY_SIZE];
char out_buffer[MEMORY_SIZE];

// initialize buffer
for (int i=0; i<MEMORY_SIZE; i++)
    out_buffer[i] = (i % 256);

ACS_ADDRESS address;
address.mem          = 0;
address.offset      = 0;

int success;

double start_t, end_t;

for (i=0; i<sites; i++) {
    cout<<"Testing node "<<i<<"..."<<endl;
    for (int k=0; k<3; k++) {
        address.pe = WF4_PE(k);
        for (int j=4; j<MEMORY_SIZE; j+=128) {
            ACS_Write (out_buffer, j, i, &address, system, &status);
            start_t = MPI_Wtime();
            ACS_Read (in_buffer, j, i, &address, system, &status);
            end_t = MPI_Wtime();
            cout<<"Reading PE("<<k<<") in "<<j<<" bytes
            in"<<"\t"<<(end_t-start_t)<<" seconds"<<endl;

            if (memcmp(in_buffer, out_buffer, j) == 0) {
                }
            else {
                cout<<"ERROR!"<<endl;
                success = FALSE;
            }
            memset(in_buffer, 0, j);
        }
    }
}

// Destroy the acs_system, release the resource.
cout<<"Destorying system...";
ACS_System_Destroy(system, &status);
cout<<"done"<<endl;

// finalize the acs world, destroy throughly.
cout<<"ACS_Finalize...";
ACS_Finalize();
cout<<"done."<<endl<<endl;

cout<<"VPI & SU, ECPE, CCM Lab. Fall 1999"<<endl;
}

void Node_initialize(ACS_NODE nodes[], int sites)
{
    for (int i=0; i<sites; i++) {
        memset((void *)&nodes[i], 0, sizeof(ACS_NODE));
        nodes[i].site = i;
    }
}

```

} }

A2.2 Streaming Data

A2.2.a Measuring RTT Value

/** This program is to test the ACS_Enqueue() and ACS_Dequeue() as well as the board management functions, including ACS_Configure(), ACS_Reset() and ACS_Run(). The program first links variety number of nodes into a ring, then uses board management functions to configure the board, start the clock, and send the RESET signal to the board. It then calls ACS_Enqueue() to put data into the system object. The data should pass through each board in the ring automatically and finally go back to the system object. The test program then calls ACS_Dequeue() to get the data back from the system object.

This program measures one fact of the performance for the data stream functions, the round trip time (RTT). The RTT is the point of total time for putting data into the system, passing through and processing data in each board, and getting data back from the system. The RTT is measured by enqueues a range of size of data into the system */

```
#include <stdio.h>
#include <iostream.h>
#include "acs.h"
#include "CWildForce4Node.h"

#define FIFO_SIZE          1024
#define PASS_THROUGH      1000

#define WF4_CLOCK_FREQUENCY 24.576

_SYSTEM_TYPE(CWildForce4Node);

void Node_initialize(ACS_NODE nodes[], int sites);
void Channel_initialize(ACS_CHANNEL channels[], int sites);
void Config_Nodes(ACS_SYSTEM *, int, int);
DWORD *readBitFile (const char *filename, int *numBytes);
double  pt_time[PASS_THROUGH];

void main(int argc, char ** argv)
{
    ACS_STATUS  status;
    char        version[0x10];

    // Initialize the SLAAC system.
    cout<<"ACS Initializing...";
    ACS_Initialize(&argc, &argv, &status);
    cout<<"done"<<endl;

    // Get the ACS world information.
    // How many workstations are in the system.
    ACS_WORLD_INFO wf;
    ACS_World_Info(&wf);
    int sites = wf.site_number;

    // Construct the network topology
    ACS_CHANNEL      channels[0x10];
```

```

ACS_NODE          nodes[0x10];
Node_initialize(nodes, sites);
Channel_initialize(channels, sites);

// Creating the ACS system...
cout<<"Creating system...";
ACS_SYSTEM * system;
ACS_System_Create(&system, nodes, sites, channels, sites + 1, &status);
cout<<"done. node number: "<<sites<<" , linked by a ring"<<endl<<endl;

char   in_buffer[FIFO_SIZE];
char   out_buffer[FIFO_SIZE];
int    success = TRUE;

for (int j=0; j<FIFO_SIZE; j++)
    out_buffer[j] = (j % 256);

for (int i=0; i<sites; i++)
    Config_Nodes(system, nodes[i].number, i);

for (i=0; i<sites; i++)
    ACS_Run(nodes[i].number, system, &status);

int dequeue;
for (i=0; i<PASS_THROUGH; i++) {
    memset(in_buffer, 0, FIFO_SIZE);

    double start_t = MPI_Wtime();
    ACS_Enqueue(out_buffer, FIFO_SIZE, 0, system, &status);

    dequeue = 0;
    while (dequeue != FIFO_SIZE)
        dequeue += ACS_Dequeue(in_buffer, FIFO_SIZE -
            dequeue, 0, system, &status);
    double end_t = MPI_Wtime();
    pt_time[i] = (end_t - start_t);
}

double total_time = 0;
int invalid = 0;
for (i=1; i<PASS_THROUGH; i++) {
    if ((int)(pt_time[i] / pt_time[i-1]) > 2) {
        pt_time[i] = pt_time[i-1];
        invalid ++;
    }
    else {
        total_time += pt_time[i];
    }
}

cout<<"The average RTT values is "<<(total_time / (PASS_THROUGH-
invalid))<<" seconds"<<endl;

cout<<"total invalid numer is "<<invalid<<endl;

// Destroy the acs_system, release the resource.
Sleep(3000);

```

```

    cout<<"Destorying system...";
    ACS_System_Destroy(system, &status);
    cout<<"done"<<endl;

    // finalize the acs world, destroy throughly.
    cout<<"ACS_Finalize...";
    ACS_Finalize();
    cout<<"done."<<endl<<endl;

    cout<<"VPI & SU, ECPE, CCM Lab. Fall 1999"<<endl;
}

void Node_initialize(ACS_NODE nodes[], int sites)
{
    for (int i=0; i<sites; i++) {
        memset((void *)&nodes[i], 0, sizeof(ACS_NODE));
        nodes[i].site = i;
    }
}

void Channel_initialize(ACS_CHANNEL channels[], int sites)
{
    for (int i=1; i<sites; i++) {
        channels[i].src_node      = i-1;
        channels[i].src_port      = WF4_Fifo_Pe4;
        channels[i].des_node      = i;
        channels[i].des_port      = WF4_Fifo_Pe1;
        channels[i].window_size  = 10;
        channels[i].dequeue_size = 1024;
        channels[i].number       = 0;
    }
    channels[0].src_node      = ACS_HOST_NUM(0);
    channels[0].src_port      = 0;
    channels[0].des_node      = 0;
    channels[0].des_port      = WF4_Fifo_Pe1;
    channels[0].window_size  = 10;
    channels[0].dequeue_size = 1024;
    channels[0].number       = 0;

    channels[sites].src_node  = sites - 1;
    channels[sites].src_port  = WF4_Fifo_Pe4;
    channels[sites].des_node  = ACS_HOST_NUM(0);
    channels[sites].des_port  = 0;
    channels[sites].window_size  = 10;
    channels[sites].dequeue_size  = 1024;
    channels[sites].number     = 0;
}

void Config_Nodes(ACS_SYSTEM * system, int node_id, int board_id)
{
    int          i;
    ACS_CONFIG   config;
    ACS_CLOCK    clock;
    char         configFileName[256];
    ACS_STATUS   status;

```

```

char          boardname[256];
int          serial_no = 0;

sprintf(config.label, "Passthrough Test");
sprintf(boardname, "board%d", board_id);

for (i = 0; i < WF4_MAX_PES; i++) {
    sprintf (configFileName, "%s\\%spe%d.x86", boardname, (i)?"": "c", i);
    printf("config File %s\n",configFileName);
    config.bitstream = readBitFile (configFileName, &(config.count));
    config.serial_no = serial_no++;

    if (config.bitstream == NULL) {
        printf ("initSlaac: NULL configuration buffer! Exiting.\n");
        exit (-1);
    }

    config.pe_mask = WF4_PE (i);

    ACS_Configure(&config, node_id, system, &status);
    delete [] config.bitstream;
}

// Start the clock
clock.frequency = WF4_CLOCK_FREQUENCY;
clock.countdown = 0;
if (ACS_Clock_Set(&clock, node_id, system, &status) != ACS_SUCCESS)
    printf ("initSlaac: ClockSet failed.\n");

if (ACS_Reset(node_id, system, ACS_PE1, TRUE, &status) != ACS_SUCCESS)
    printf ("initSlaac: Failed to send reset.\n");

printf ("Clock set to %.3f MHz.\n", WF4_CLOCK_FREQUENCY);
}

DWORD *readBitFile (const char *filename, int *numBytes)
{
    FILE *inFile;
    unsigned char *buffer;
    long fileSize;

    // try to open the input file
    if ((inFile = fopen (filename, "rb")) == NULL) {
        *numBytes = 0;
        printf ("readBitFile: Couldn't open file '%s'\n", filename);
        return NULL;
    }

    // figure out how long the file is
    fseek (inFile, 0, SEEK_END);
    fileSize = ftell (inFile);
    *numBytes = (int) fileSize;
    // make sure numBytes is accurate...
    if (*numBytes != fileSize) {
        printf ("readBitFile: Oh, no! File '%s' is bigger than an
int!\n", filename);

```

```
        fclose (inFile);
        return NULL;
    }

    // make the new buffer
    buffer = new unsigned char [fileSize];

    // read in the bytes
    fseek (inFile, 0, SEEK_SET);
    fread ((void *) buffer, sizeof (unsigned char), fileSize, inFile);
    return (DWORD *) buffer;
}
```

A2.2.b Measuring throughput

/** This program differs from the RTT test program by creating a thread to dequeue the data instead of calling ACS_Dequeue() in the host program. The host program then keeps enqueueing five mega bytes data into the system and records the time of enqueueing every 1024 bytes data. */

```
#include <stdio.h>
#include <iostream.h>
#include "acs.h"
#include "CWildForce4Node.h"
#include "mpi.h"

#define FIFO_SIZE          1024
#define PASS_THROUGH      2500

#define WF4_CLOCK_FREQUENCY 24.576

_SYSTEM_TYPE(CWildForce4Node);

void Node_initialize(ACS_NODE nodes[], int sites);
void Channel_initialize(ACS_CHANNEL channels[], int sites);
void Config_Nodes(ACS_SYSTEM *, int, int);
DWORD *readBitFile (const char *filename, int *numBytes);
void Dequeue_thread(void *);

char    in_buffer[FIFO_SIZE];
char    out_buffer[FIFO_SIZE];

ACS_SYSTEM *    acs_system;
ACS_CHANNEL    channels[0x10];
ACS_NODE       nodes[0x10];

int thread_quit = 0;
int quit = 0;

double pt_time[PASS_THROUGH];

void main(int argc, char ** argv)
{
    ACS_STATUS  status;
    char        version[0x10];

    // Initialize the SLAAC system.
    cout<<"ACS Initializing...";
    ACS_Initialize(&argc, &argv, &status);
    cout<<"done"<<endl;

    // Initialize log file. make sure the path is a network path so
    // each node can visit it
    cout<<"Log Initializing...";
    ACS_Initialize_Log("\\\\tuba\\home\\mpi\\");
    cout<<"done"<<endl;

    // Get the ACS world information.
    // How many nodes participated in the computation.
```

```

ACS_WORLD_INFO wf;
ACS_World_Info(&wf);
int sites = wf.site_number;

Node_initialize(nodes, sites);
Channel_initialize(channels, sites);

// Creating the ACS system...
cout<<"Creating system...";
ACS_System_Create(&acs_system, nodes, sites, channels, sites + 1, &status);
cout<<"done. node number: "<<sites<<" , linked by a ring"<<endl<<endl;

int    success = TRUE;

for (int j=0; j<FIFO_SIZE; j++)
    out_buffer[j] = (j % 256);

for (int i=0; i<sites; i++)
    Config_Nodes(acs_system, nodes[i].number, i);

printf("Sleeping...");
for (i=0; i<4; i++) {
    ACS_SLEEP(1000);
    printf(".");
}
printf("\nWake up\n");

ACS_THREAD_SPAWN(Dequeue_thread);

for (i=0; i<PASS_THROUGH; i++) {
    double start_t = MPI_Wtime();
    ACS_Enqueue(out_buffer, FIFO_SIZE, 0, acs_system, &status);
    double end_t = MPI_Wtime();
    pt_time[i] = end_t - start_t;
    printf("%d\n", i);
}

#define _SKIP 500

double total_time = 0;
int invalid = 0;

long total_data = (long)FIFO_SIZE * (PASS_THROUGH - invalid - _SKIP);
double throughput = (double)total_data / total_time;

cout<<"The average throughput is "<<throughput<<" bytes/seconds"<<endl;
cout<<"total invalid number "<<invalid<<endl;

quit = 1;
printf("wait until thread quit");
while (!thread_quit);

cout<<endl<<"Sleeping....";
for (i=0; i<4; i++) {
    ACS_SLEEP(1000);
    printf(".");
}

```

```

printf("\nWake up\n");

cout<<"Destorying system...";
ACS_System_Destroy(acs_system, &status);
cout<<"done"<<endl;

// finalize the acs world, destroy throughly.
cout<<"ACS_Finalize...";
ACS_Finalize();
cout<<"done."<<endl<<endl;

cout<<"VPI & SU, ECPE, CCM Lab. Fall 1999"<<endl;
}

void Node_initialize(ACS_NODE nodes[], int sites)
{
    for (int i=0; i<sites; i++) {
        memset((void *)&nodes[i], 0, sizeof(ACS_NODE));
        nodes[i].site = i;
    }
}

void Channel_initialize(ACS_CHANNEL channels[], int sites)
{
    for (int i=1; i<sites; i++) {
        channels[i].src_node      = i-1;
        channels[i].src_port      = WF4_Fifo_Pe4;
        channels[i].des_node      = i;
        channels[i].des_port      = WF4_Fifo_Pe1;
        channels[i].window_size  = 10;
        channels[i].dequeue_size = 1024;
        channels[i].number       = 0;
    }
    channels[0].src_node          = ACS_HOST_NUM(0);
    channels[0].src_port          = 0;
    channels[0].des_node          = 0;
    channels[0].des_port          = WF4_Fifo_Pe1;
    channels[0].window_size      = 10;
    channels[0].dequeue_size     = 1024;
    channels[0].number           = 0;

    channels[sites].src_node      = sites - 1;
    channels[sites].src_port      = WF4_Fifo_Pe4;
    channels[sites].des_node      = ACS_HOST_NUM(0);
    channels[sites].des_port      = 0;
    channels[sites].window_size   = 10;
    channels[sites].dequeue_size  = 1024;
    channels[sites].number        = 0;
}

void Config_Nodes(ACS_SYSTEM * system, int node_id, int board_id)
{
    int i;
    ACS_CONFIG config;
    ACS_CLOCK clock;
    char configFileName[256];
    ACS_STATUS status;

```

```

char          boardname[256];
int          serial_no = 0;

sprintf(config.label, "Passthrough Test");
sprintf(boardname, "board%d", board_id);

for (i = 0; i < WF4_MAX_PES; i++) {
    sprintf (configFileName, "%s\\%spe%d.x86", boardname, (i)?"": "c", i);
    printf("config File %s\n", configFileName);
    config.bitstream = readBitFile (configFileName, &(config.count));
    config.serial_no = serial_no++;

    if (config.bitstream == NULL) {
        printf ("initSlaac: NULL configuration buffer! Exiting.\n");
        exit (-1);
    }
    config.pe_mask = WF4_PE (i);
    ACS_Configure(&config, node_id, system, &status);
    delete [] config.bitstream;
}

// Start the clock
clock.frequency = WF4_CLOCK_FREQUENCY;
clock.countdown = 0;
if (ACS_Clock_Set(&clock, node_id, system, &status) != ACS_SUCCESS)
    printf ("initSlaac: ClockSet failed.\n");

// Run the board
if (ACS_Run(node_id, system, &status) != ACS_SUCCESS)
    printf ("initSlaac: Failed to start clock.\n");

// Reset the board
if (ACS_Reset(node_id, system, ACS_PE1, TRUE, &status) != ACS_SUCCESS)
    printf ("initSlaac: Failed to send reset.\n");

printf ("Clock set to %.3f MHz.\n", WF4_CLOCK_FREQUENCY);
}

DWORD *readBitFile (const char *filename, int *numBytes)
{
    FILE *inFile;
    unsigned char *buffer;
    long fileSize;

    // try to open the input file
    if ((inFile = fopen (filename, "rb")) == NULL) {
        *numBytes = 0;
        printf ("readBitFile: Couldn't open file '%s\n", filename);
        return NULL;
    }

    // figure out how long the file is
    fseek (inFile, 0, SEEK_END);
    fileSize = ftell (inFile);
    *numBytes = (int) fileSize;
    // make sure numBytes is accurate...

```

```

    if (*numBytes != fileSize) {
        printf ("readBitFile: Oh, no! File '%s' is bigger than an
int!\n", filename);
        fclose (inFile);
        return NULL;
    }

    // make the new buffer
    buffer = new unsigned char [fileSize];

    // read in the bytes
    fseek (inFile, 0, SEEK_SET);
    fread ((void *) buffer, sizeof (unsigned char), fileSize, inFile);

    return (DWORD *) buffer;
}

void Dequeue_thread(void *)
{
    long dequeue = 0;
    ACS_STATUS status;

    while (!quit) {
        dequeue += ACS_Dequeue(in_buffer, FIFO_SIZE, 0, acs_system, &status);
    }

    printf("dequeued %d bytes data\n", dequeue);

    thread_quit = 1;
}

```

A2.3 Non-blocking

/** This program illustrates how the user takes advantage of using non-blocking functions to reduce the total execution time through overlapping the computation and communication */

```
#include <stdio.h>
#include <iostream.h>
#include "mpi.h"
#include "acs.h"
#include "CWildForce4Node.h"

#define MEMORY_SIZE 1024000

_SYSTEM_TYPE(CWildForce4Node);

void Node_initialize(ACS_NODE nodes[], int sites);
void Channel_initialize(ACS_CHANNEL channels[], int sites);

char out_buffer[MEMORY_SIZE];
char in_buffer[MEMORY_SIZE];

void main(int argc, char ** argv)
{
    ACS_STATUS  status;
    char        version[0x10];

    // Get the SLAAC API version
    ACS_Version(version);
    cout<<"SLAAC APIs version "<<version<<endl;

    // Initialize the SLAAC system.
    cout<<"ACS Initializing...";
    ACS_Initialize(&argc, &argv, &status);
    cout<<"done"<<endl;

    // Get to know the ACS world information.
    // How many nodes participated in the computation.
    ACS_WORLD_INFO wf;
    ACS_World_Info(&wf);
    int sites = wf.site_number;

    // Construct the network topology
    ACS_CHANNEL        channels[0x10];
    ACS_NODE           nodes[0x10];
    Node_initialize(nodes, sites);
    //Channel_initialize(channels, sites);

    // Creating the ACS system...
    cout<<"Creating system...";
    ACS_SYSTEM  * system;
    ACS_System_Create(&system, nodes, sites, channels, 0, &status);

    ACS_REQUEST * request;
    ACS_REQUEST_STAT r_status;
```

```

ACS_ADDRESS addr;

for (int i=0; i<MEMORY_SIZE; i++) out_buffer[i] = (i % 256);

ACS_Request_Create(&request, 5);

addr.mem      = 0;
addr.offset   = 0;
addr.pe      = WF4_PE(0);

double start_t, end_t;

#define _REPEAT      50
#define _REPEAT2    5

start_t = MPI_Wtime();
for (int j=0; j<_REPEAT2; j++) {
    ACS_Write(out_buffer, MEMORY_SIZE, nodes[1].number, &addr,
              system, &status);
    ACS_Read(in_buffer, MEMORY_SIZE, nodes[1].number, &addr,
             system, &status);
}
for (j=0; j<_REPEAT; j++) {
    ACS_Write(out_buffer, MEMORY_SIZE, nodes[0].number, &addr,
              system, &status);
    ACS_Read(in_buffer, MEMORY_SIZE, nodes[0].number, &addr,
             system, &status);
}
end_t = MPI_Wtime();
printf("\nTotal time for blocking call is %f\n", (end_t - start_t));

start_t = MPI_Wtime();
for (j=0; j<_REPEAT2; j++) {
    ACS_WriteN(request, out_buffer, MEMORY_SIZE,
               nodes[1].number, &addr, system, &status);
    ACS_ReadN(request, in_buffer, MEMORY_SIZE, nodes[1].number,
              &addr, system, &status);
}
ACS_Commit(request, &r_status);
for (j=0; j<_REPEAT; j++) {
    ACS_Write(out_buffer, MEMORY_SIZE, nodes[0].number, &addr,
              system, &status);
    ACS_Read(in_buffer, MEMORY_SIZE, nodes[0].number, &addr,
             system, &status);
}
ACS_Wait(request, &r_status);
end_t = MPI_Wtime();
printf("\nTotal time for non-blocking call is %f\n", (end_t - start_t));

ACS_Request_Destroy(request);

// Destroy the acs_system, release the resource.
cout<<"Destorying system...";
ACS_System_Destroy(system, &status);
cout<<"done"<<endl;

```

```
// finalize the acs world, destroy throughly.
cout<<"ACS_Finalize...";
ACS_Finalize();
cout<<"done."<<endl<<endl;

cout<<"VPI & SU, ECPE, CCM Lab. Fall 1999"<<endl;
}

void Node_initialize(ACS_NODE nodes[], int sites)
{
    for (int i=0; i<sites; i++) {
        memset((void *)&nodes[i], 0, sizeof(ACS_NODE));
        nodes[i].site = i;
    }
}
```

A2.4 Testing ACS_ALL

/** This program is revised from the data stream test program. Instead of using an individual node id in calling the board management functions, it uses ACS_ALL. So the user only needs to call those functions once. */

```
#include <stdio.h>
#include <iostream.h>
#include "acs.h"
#include "CWildForce4Node.h"

#define WF4_CLOCK_FREQUENCY 24.576

_SYSTEM_TYPE(CWildForce4Node);

void Node_initialize(ACS_NODE nodes[], int sites);
void Channel_initialize(ACS_CHANNEL channels[], int sites);
void Config_Nodes(ACS_SYSTEM * system, int node_id, int board_id);
DWORD *readBitFile (const char *filename, int *numBytes);

void main(int argc, char ** argv)
{
    ACS_STATUS    status;
    char          version[0x10];

    // Get the SLAAC API version
    ACS_Version(version);
    cout<<"SLAAC APIs version "<<version<<endl;

    // Initialize the SLAAC system.
    cout<<"ACS Initializing...";
    ACS_Initialize(&argc, &argv, &status);
    cout<<"done"<<endl;

    // Get the ACS world information.
    // How many nodes participated in the computation.
    ACS_WORLD_INFO wf;
    ACS_World_Info(&wf);
    int sites = wf.site_number;

    // Construct the network topology
    ACS_CHANNEL          channels[0x10];
    ACS_NODE             nodes[0x10];
    Node_initialize(nodes, sites);
    Channel_initialize(channels, sites);

    // Creating the ACS system...
    cout<<"Creating system...";
    ACS_SYSTEM    * system;
    ACS_System_Create(&system, nodes, sites, channels, sites + 1, &status);
    cout<<"done. node number: "<<sites<<" , linked by a ring"<<endl<<endl;

    for (int i=0; i<sites; i++)
        Config_Nodes(system, nodes[i].number, i);
}
```

```

ACS_CLOCK    clock;
clock.frequency = WF4_CLOCK_FREQUENCY;
clock.countdown = 0;

if (ACS_Clock_Set(&clock, ACS_ALL, system, &status) != ACS_SUCCESS)
    printf ("initSlaac: ClockSet failed.\n");

if (ACS_Run(ACS_ALL, system, &status) != ACS_SUCCESS)
    printf ("initSlaac: Failed to start clock.\n");
printf ("before reset\n");

if (ACS_Reset(ACS_ALL, system, ACS_PE1, TRUE, &status) != ACS_SUCCESS)
    printf ("initSlaac: Failed to send reset.\n");

printf ("Clock set to %.3f MHz.\n", WF4_CLOCK_FREQUENCY);

// Destroy the acs_system, release the resource.
cout<<"Destorying system...";
ACS_System_Destroy(system, &status);
cout<<"done"<<endl;

// finalize the acs world, destroy throughly.
cout<<"ACS_Finalize...";
ACS_Finalize();
cout<<"done."<<endl<<endl;

cout<<"VPI & SU, ECPE, CCM Lab. Fall 1999"<<endl;
}

void Node_initialize(ACS_NODE nodes[], int sites)
{
    for (int i=0; i<sites; i++) {
        memset((void *)&nodes[i], 0, sizeof(ACS_NODE));
        nodes[i].site = i;
    }
}

void Channel_initialize(ACS_CHANNEL channels[], int sites)
{
    for (int i=1; i<sites; i++) {
        channels[i].src_node = i-1;
        channels[i].src_port = WF4_Fifo_Pe4;
        channels[i].des_node = i;
        channels[i].des_port = WF4_Fifo_Pe1;
    }
    channels[0].src_node    = ACS_HOST_NUM(0);
    channels[0].src_port    = 0;
    channels[0].des_node    = 0;
    channels[0].des_port    = WF4_Fifo_Pe1;

    channels[sites].src_node = sites - 1;
    channels[sites].src_port = WF4_Fifo_Pe4;
    channels[sites].des_node = ACS_HOST_NUM(0);
    channels[sites].des_port = 0;
}

```

```

void Config_Nodes(ACS_SYSTEM * system, int node_id, int board_id)
{
    int                i;
    ACS_CONFIG        config;
    char              configFileName[256];
    ACS_STATUS        status;
    char              boardname[256];

    int                serial_no = 0;

    sprintf(config.label, "Passthrough Test");
    sprintf(boardname, "board%d", board_id);

    for (i = 0; i < WF4_MAX_PES; i++) {
        sprintf (configFileName, "%s\\%spe%d.x86", boardname, (i)?"": "c", i);
        printf("config File %s\n", configFileName);
        config.bitstream = readBitFile (configFileName, &(config.count));
        config.serial_no = serial_no++;

        if (config.bitstream == NULL) {
            printf ("initSlaac: NULL configuration buffer! Exiting.\n");
            exit (-1);
        }

        config.pe_mask = WF4_PE (i);

        ACS_Configure(&config, node_id, system, &status);
        delete [] config.bitstream;
    }
}

```

```

DWORD *readBitFile (const char *filename, int *numBytes)
{
    FILE *inFile;
    unsigned char *buffer;
    long fileSize;

    // try to open the input file
    if ((inFile = fopen (filename, "rb")) == NULL) {
        *numBytes = 0;
        printf ("readBitFile: Couldn't open file '%s'\n", filename);
        return NULL;
    }

    // figure out how long the file is
    fseek (inFile, 0, SEEK_END);
    fileSize = ftell (inFile);
    *numBytes = (int) fileSize;
    // make sure numBytes is accurate...
    if (*numBytes != fileSize) {
        printf ("readBitFile: Oh, no! File '%s' is bigger than an
int!\n", filename);
        fclose (inFile);
        return NULL;
    }
}

```

```
// make the new buffer
buffer = new unsigned char [fileSize];

// read in the bytes
fseek (inFile, 0, SEEK_SET);
fread ((void *) buffer, sizeof (unsigned char), fileSize, inFile);

return (DWORD *) buffer;
}
```

A2.5 Cache configuration file

*/** In this test, the host program sends three different configuration files to the remote node repeatedly. Since the configuration files have been cached in the remote machine during the first time when it was configured, the system will not send any actual data to the remote machine in the later time. The time of reconfiguring the board has been significantly reduced after the first time. */*

```
#include <stdio.h>
#include <iostream.h>
#include "acs.h"
#include "acs_system.h"
#include "CWildForce4Node.h"

_SYSTEM_TYPE(CWildForce4Node);

void Node_initialize(ACS_NODE nodes[], int sites);
void Channel_initialize(ACS_CHANNEL channels[], int sites);
void Config_Nodes(ACS_SYSTEM * system, int node_id, int board_id);
DWORD *readBitFile (const char *filename, int *numBytes) ;

void main(int argc, char ** argv)
{
    ACS_STATUS   status;
    char         version[0x10];

    // Get the SLAAC API version
    ACS_Version(version);
    cout<<"SLAAC APIs version "<<version<<endl;

    // Initialize the SLAAC system.
    cout<<"ACS Initializing...";
    ACS_Initialize(&argc, &argv, &status);
    cout<<"done"<<endl;

    // Initialize log file. make sure the path is a network path so
    // each node can visit it
    cout<<"Log Initializing...";
    ACS_Initialize_Log("\\\\tuba\\home\\mpi\\");
    cout<<"done"<<endl;

    // Get to know the ACS world information.
    // How many nodes participated in the computation.
    ACS_WORLD_INFO wf;
    ACS_World_Info(&wf);
    int sites = wf.site_number;

    // Construct the network topology
    ACS_CHANNEL     channels[0x10];
    ACS_NODE        nodes[0x10];
    Node_initialize(nodes, sites);
    Channel_initialize(channels, sites);

    // Creating the ACS system...
    cout<<"Creating system...";
```

```

ACS_SYSTEM * system;
ACS_System_Create(&system, nodes, sites, channels, sites + 1, &status);
cout<<"done. node number: "<<sites<<", linked by a ring"<<endl<<endl;

double start_t, end_t;

for (int i=0; i<2; i++) {
    for (int j=0; j<sites; j++) {
        start_t = MPI_Wtime();
        Config_Nodes(system, nodes[j].number, j);
        end_t = MPI_Wtime();
        cout<<endl<<endl<<"Send configure to node "<<j<<" in
        "<<(end_t-start_t)<<" seconds"<<endl;
    }
    Sleep(2000);
}

// Destroy the acs_system, release the resource.
cout<<"Destorying system...";
ACS_System_Destroy(system, &status);
cout<<"done"<<endl;

// finalize the acs world, destroy throughly.
cout<<"ACS_Finalize...";
ACS_Finalize();
cout<<"done."<<endl<<endl;

cout<<"VPI & SU, ECPE, CCM Lab. Fall 1999"<<endl;
}

void Node_initialize(ACS_NODE nodes[], int sites)
{
    for (int i=0; i<sites; i++) {
        memset((void *)&nodes[i], 0, sizeof(ACS_NODE));
        nodes[i].site = i;
    }
}

void Channel_initialize(ACS_CHANNEL channels[], int sites)
{
    for (int i=1; i<sites; i++) {
        channels[i].src_node = i-1;
        channels[i].src_port = WF4_Fifo_Pe4;
        channels[i].des_node = i;
        channels[i].des_port = WF4_Fifo_Pe1;
    }
    channels[0].src_node = ACS_HOST_NUM(0);
    channels[0].src_port = 0;
    channels[0].des_node = 0;
    channels[0].des_port = WF4_Fifo_Pe1;

    channels[sites].src_node = sites - 1;
    channels[sites].src_port = WF4_Fifo_Pe4;
    channels[sites].des_node = ACS_HOST_NUM(0);
    channels[sites].des_port = 0;
}

```

```

void Config_Nodes(ACS_SYSTEM * system, int node_id, int board_id)
{
    int                i;
    ACS_CONFIG         config;
    ACS_CLOCK          clock;
    char               configFileName[256];
    ACS_STATUS         status;
    char               boardname[256];

    int                serial_no = 0;

    sprintf(config.label, "Passthrough Test");
    sprintf(boardname, "board%d", board_id);

    for (i = 0; i < WF4_MAX_PES; i++) {
        sprintf (configFileName, "%s\\%spe%d.x86", boardname, (i)?":": "c", i);
        printf("config File %s\n", configFileName);
        config.bitstream = readBitFile (configFileName, &(config.count));
        config.serial_no = serial_no++;

        if (config.bitstream == NULL) {
            printf ("initSlaac: NULL configuration buffer! Exiting.\n");
            exit (-1);
        }
        config.pe_mask = WF4_PE (i);
        ACS_Configure(&config, node_id, system, &status);
        delete [] config.bitstream;
    }
}

DWORD *readBitFile (const char *filename, int *numBytes)
{
    FILE *            inFile;
    unsigned char *   buffer;
    long              fileSize;

    // try to open the input file
    if ((inFile = fopen (filename, "rb")) == NULL) {
        *numBytes = 0;
        printf ("readBitFile: Couldn't open file '%s'\n", filename);
        return NULL;
    }

    // figure out how long the file is
    fseek (inFile, 0, SEEK_END);
    fileSize = ftell (inFile);
    *numBytes = (int) fileSize;
    // make sure numBytes is accurate...
    if (*numBytes != fileSize) {
        printf ("readBitFile: Oh, no! File '%s' is bigger than an
int!\n", filename);
        fclose (inFile);
        return NULL;
    }

    // make the new buffer

```

```
buffer = new unsigned char [fileSize];

// read in the bytes
fseek (inFile, 0, SEEK_SET);
fread ((void *) buffer, sizeof (unsigned char), fileSize, inFile);
return (DWORD *) buffer;
}
```