The Effects of Caching on Reconfigurable Adaptive Computing Systems

James Hugh Hendry

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 Jones, Chair
James Armstrong
Peter Athanas

December 19, 2003
Bradley Department of Electrical and Computer Engineering
Blacksburg, Virginia

keywords: Configurable Computing, FPGA, Adaptive Computing, Run-Time Reconfiguration

The Effects of Caching on Reconfigurable Adaptive Computing Systems

James H. Hendry

(ABSTRACT)

Adaptive computing systems have proven useful for implementing a wide range of algorithms. A limitation of current systems is the relatively small amount of reconfigurable hardware resources. Many algorithms require more hardware resources than are available. One solution to this problem is runtime reconfiguration (RTR). Using RTR techniques, a large algorithm is implemented as a collection of configurations for the reconfigurable hardware. These configurations are loaded onto the reconfigurable hardware as necessary to implement the algorithm. A primary limitation of RTR is that the reconfiguration process is slow. Therefore, methods of decreasing reconfiguration time are desirable.

Another method of implementing large algorithms on small hardware is to use multiple configurable computing platforms connected via a communication network. RTR techniques can be used in conjunction with this method to further increase hardware availability. In this case reconfiguration time is increased by the overhead of transmitting data across the communication network. Methods of decreasing network overhead are desirable.

This thesis discusses the use of caching techniques to decrease reconfiguration time. An architecture for caching configurations is implemented on a configurable computing system platform. The use of caching to decrease network overhead is discussed and exhibited. An example application is implemented and used to evaluate the effects of caching on reconfiguration time and algorithm performance.

Acknowledgements

I would like to thank my graduate advisor, Dr. Mark T. Jones, and the rest of my committee, Dr. Peter M. Athanas and Dr. James R. Armstrong, for their support, time, and exceptional flexibility. Special thanks goes to Scott Harper for extra guidance and proofreading services. Many thanks to William Worek, David Lehn, George Morgan, Zahi Nakad, Jason Zimmerman, and Ryan Fong, who all provided technical assistance and worked on projects related to this research. Thanks must also go to all of my friends and family.

Contents

List of Figures

List of Tables

Listings

Terms

ACS Adaptive Computing Systems

API Application Program Interface

ASIC Application Specific Integrated Circuit

CCM Configurable Computing Machine

CPU Central Processing Unit

CSRC Context Switching Reconfigurable Computer

DRAM Dynamic RAM

FIFO First-In First-Out

FPGA Field-Programmable Gate Array

FSM Finite State Machine

LRU Least Recently Used

RAM Random Access Memory

RCM Reconfigurable Computing Module

RFUOP Reconfigurable Functional Unit Operation

RTR Run-Time Reconfigurable

SLAAC Systems Level Applications of Adaptive Computing

SRAM Static RAM

VHDL VHSIC Hardware Description Language

VHSIC Very High Speed Integrated Circuits

Chapter 1

Introduction

1.1 Motivation

Field Programmable Gate Arrays (FPGAs) exhibit many of the beneficial characteristics of both Application Specific Integrated Circuits (ASICs) and software. FPGAs achieve performance similar to ASICs while also being reprogrammable. This allows different configurations to the loaded onto the FPGA to implement different algorithms. Loading a configuration onto an FPGA is analogous to loading a program on a computer. New configurations can be loaded at any time. Current reconfigurable computing platforms use FPGAs to provide reconfigurable hardware.

Reconfigurable computing systems exhibit higher performance than general purpose computing systems for a variety of problem types. A major drawback of current reconfigurable computing systems is the limited amount of configurable hardware resources. Research has shown that this limitation can be minimized through the use of runtime reconfiguration (RTR). Using RTR, an algorithm is mapped to two or more configurations. These configurations are then loaded onto the configurable hardware resource as necessary to implement the algorithm. The benefits of RTR are limited by the rate of reconfiguration. Current

general use configurable computing platforms exhibit long reconfiguration times.

An additional approach to increasing the availability of configurable hardware resources is to implement algorithms spanning multiple configurable computing platforms. Data is transferred between the platforms via a data network. Configurations must also be sent across the data network. RTR may be used in conjuntion with using multiple platforms. In this case, configurations must be sent across the data network. Typically, configurations must be fully transferred across the network before being sent to the board. Therefore, the network transfer time is summed with local configuration time to get the total reconfiguration time. Some method of minimizing transfer of configurations over the network is desirable.

1.2 Thesis Statement

This thesis presents a method of reducing configuration times for reconfigurable computing platforms by caching configuration data at multiple levels. The Adaptive Computing System Application Programming Interface (ACS API) [1] is used to facilitate evaluating multiple platform systems. This thesis contributes the following.

- It demonstrates and analyzes the effects of caching configuration data on reconfigurable computing platforms in order to minimize bus transfers.
- It demonstrates and analyzes the caching of configurations in order to minimize network transfer in multiple platform systems using the ACS API.
- It analyzes the effects of using the two levels of caching simultaneously.
- It analyzes the results of using configuration caching with a data streaming application.

1.3 Organization

This thesis is arranged in seven chapters. Chapter 2 contains background information for the work presented in this thesis. Chapter 3 contains a description of the reconfigurable computing platform and the associated software interface. The ACS API is described in Chapter 4. The implementation of caching is described in Chapter 5, along with a test application that utilizes caching capabilities. Results and analysis are presented in Chapter 6. Conclusions are drawn in Chapter 7.

Chapter 2

Background

This chapter discusses the history of reconfigurable computing, from its origins through the current state of the art. The concept of *Virtual Hardware* is explained. The effect of configuration time on the effectiveness of *Virtual Hardware* is discussed. Finally, the concept of caching configurations in order to decrease configuration time is discussed.

2.1 Reconfigurable Computing

There are two traditional approaches to implementing digital logic devices: mapping an algorithm to a general purpose processor and designing hardware that implements an algorithm. General purpose (GP) processors can implement a wide variety of tasks, but do not fully utilize the potential power of the silicon with which they are implemented. The other approach is to design custom silicon for a particular task. The custom silicon is commonly known as an Application Specific Integrated Circuit (ASIC). Hardware designed for a specific task generally exhibits much higher performance for that task than an equivalent implementation utilizing general purpose processors. A major drawback is increased application development time. Instead of writing software to run on a GP processor, hardware must be designed and

fabricated. The other drawback is the specialized nature of the hardware. Few, if any, other algorithms can be mapped to the hardware.

Programmable Logic Devices (PLDs) provide an alternative approach to implementing digital logic directly in hardware. PLDs are generic silicon that can be configured to implement arbitrary digital logic. PLDs have become popular for a number of reasons. Development time for PLDs is shorter than for ASICs, while designs generally exhibit similar performance. During the design phase, PLDs allow for easier debugging and quicker response time to design changes than ASICs. Modern PLDs can be configured to be GP processors. PLDs combine the benefits of GP processors with the benefits of ASICs.

Reconfigurable computing was first implemented at UCLA in 1960[2]. The architecture consisted of a a number of digital components such as flip-flops, counters, and shift registers, and a wiring harness that connected the components. Different wiring harnesses resulted in different interconnections between the components, thereby implementing different algorithms with the same hardware resources.

Introduced by Xilinx in 1985, Field Programmable Gate Arrays (FPGAs) are the focus of many modern reconfigurable computing systems. An FPGA consists of a series of Lookup Tables (LUTs), flip-flops, and a reconfigurable interconnection network. Each LUT is a memory. Any Boolean function can be implemented by a LUT. The number of address lines of the LUT is the maximum number of variables that can be in the Boolean function. The number of functions that can be implemented by a LUT is equal to the number of outputs of the LUT. For example, a 4-input, 1-output LUT can implement and Boolean function containing four or less variables. Arbitrary combinational and sequential circuits can be implemented by configuring the LUTs and interconnection network. Modern FPGAs are static RAM based (SRAM). Because of this they can be reprogrammed.

Splash 2[3] is a configurable computing system made up of seventeen Xilinx XC4010 FPGAs controlled by a host computer. Sixteen of the FPGAs, or processing elements (PEs), are

connected to a 16x16 crossbar switch that is controlled by the seventeenth PE. Each PE has 512KB of local memory that is also accessible by the host computer. The host computer is responsible for configuring the PEs and feeding data to the Splash 2 system. The platform lends itself to algorithms that can be pipelined and single-instruction, multiple datastream (SIMD) operations. For pipelined algorithms, each PE acts as a stage in the pipeline, passing data along to the next PE in the pipeline. For SIMD operations, all PEs contain identical configurations but differing streams of data are sent to the PEs. Other types of algorithms can be implemented but may not fully utilize the processing power for the Splash 2 processor.

The PRISM[4] architecture uses FPGAs to adapt a general purpose processor to a given algorithm. The aim is to take advantage of a processing guideline which states that 90% of execution time is typically spent in 10% of a program. This is known as the 90/10 rule[**?**, 5]. The PRISM1 implementation of the PRISM architecture consists of a general purpose processor board connected via a 16-bit bus to a board containing a number of FPGAs. Programs for this platform are written in the C programming language. Programs are run through a configuration compiler that determines the best part of each program to optimize in hardware. The compiler generates a custom configuration for the FPGAs that implements the optimized part of the program. Significant speed-ups are achieved with this method.

The Chimaera reconfigurable function unit (RFU) was designed to be integrated with a general purpose processor[6]. In addition to the standard machine language instructions of the GP processor, special instructions are provided to utilize the RFU. The RFU has direct access to the register file of the GP processor. RFU operations can use as many operands as necessary from the GP processor. The result of the operation is written back into a GP processor register. By tightly coupling the Chimaera with a host processor communication overhead is reduced. Communication between the reconfigurable hardware and the host processor is the primary bottleneck in the other configurable computing systems mentioned.

Configurable computing technology has matured to the point where commercially available

solutions are available. A typical commercial configurable computing platform is a board containing a number of FPGAs and some RAM designed to be hosted in a computer. Examples are the Osiris[7] and SLAAC-1V[8] boards from USC-ISI, and the WildForce card from Annapolis Microsystems, Inc.[9]. All of the aforementioned configurable computing platforms are designed to communicate with a host computer via a PCI bus. The host is responsible for configuring the PEs on the board and data I/O. Another commercially available system is the the DN3000K10 system from the Dini Group[10]. While the card has the capability of communicating with a host processor via a PCI bus, it is intended to be a stand alone computing system. Configurations are loaded from a removable compact flash card. The prinicple components are a number of Xilinx FPGAs, an Atmel microprocessor, and RAM.

2.2 Virtual Hardware

Reconfigurable Computing systems have a finite amount of hardware resources. In order to implement algorithms that require more hardware resources than are available, the "Virtual Hardware" concept was conceived[11]. Virtual hardware borrows from the virtual memory concept used in computer operating systems. Virtual memory enables a computer with a finite amount of memory to emulate more memory by swapping pages of memory to disk. Virtual hardware swaps configurations of the configurable hardware instead of memory. Using virtual hardware techniques, a finite amount of hardware resources can emulate an infinite hardware resource.

The limiting factor of virtual hardware techniques is the time spent swapping configurations. This time is determined by where the configurations are stored, the size of the configurations, and the rate at which the reconfigurable hardware can accept the new configurations.

Until recently, most commercial FPGAs have been configured serially. This configuration method has the limitation that an entire FPGA must be programmed every time a new

configuration is required. This makes swapping configurations slow, severely limiting the benefits of virtual hardware. A number of solutions to this limitation have been implemented.

Because FPGA configurations are stored in memory, a straightforward approach to changing configurations rapidly is to have multiple memories. One of the earliest proposals of this technique is the WASMII chip [11]. Instead of having only one RAM per internal logic block, the WASMII had n different memories per logic block. A multiplexer was used to select one memory out of the n to configure the logic block. Additionally there was a backup RAM that could be loaded into any of the non-selected RAMs. This allowed virtual hardware techniques to be used to enable a large design to be implemented on the relatively small FPGA. The combination of background loading of configuration RAMs and the use of a multiplexer to choose among the configuration RAMs provided for a reconfiguration time of one clock cycle. The primary limitation of the WASMII architecture is that data cannot be shared between configurations in the storage units such as flip-flops and registers. Swapping configuration RAMs caused all storage units to lose state.

The Reconfigurable Computing Module (RCM) is a PCI card designed by BAE Systems to be hosted in a computer[12]. The RCM consists of a Power PC 750 microprocessor, a Xilinx 4085 FPGA, two Context Switching Reconfigurable Computer (CSRC) chips, and supporting hardware. The CSRC architecture shares similarities with the WASMII chip. The CSRC contains four configuration RAMs, enabling the chip to switch among the four configurations, or contexts, in one clock cycle. The RAMs are loaded serially however, resulting in a slow operation. This latency can be hidden by loading the unused contexts in the background while another context is active. The CSRC architecture provides two benefits over the the WASMII architecture: storage units keep their state during context switches and the chip can signal itself to switch contexts. The ability of storage units to keep their state during context switches means different contexts can operate on the same data in succession.

An alternative to context-switching reconfiguration is Partial Run-time Reconfiguration (PRTR). Instead of reconfiguring an entire chip just the portions of the chip that need

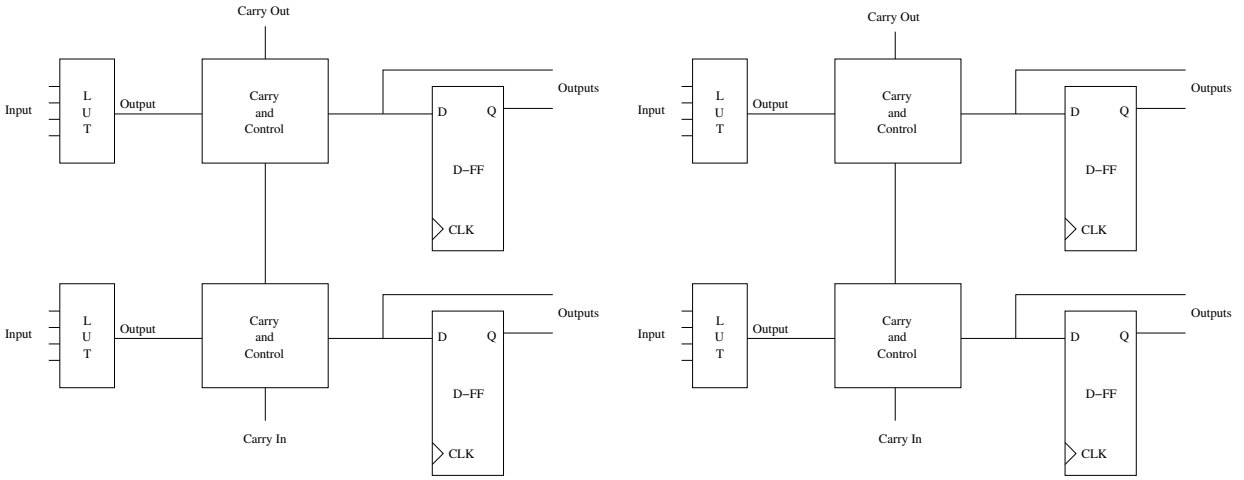

Figure 2.1: Simplified view of a Xilinx Virtex CLB

to be changed are configured. This can be done while the rest of the chip is still active. A number of commercial FPGAs support PRTR. In particular the Virtex chip from Xilinx supports PRTR[13]. The Virtex chips consists of a number of Configurable Logic Blocks (CLBs). As shown Figure 2.1, each CLB contains four LUTs, four D-FFs that can be configured as latches, fast carry logic, and SRAM control logic. The CLBs are arranged in a two-dimensional matrix in the Virtex. Each column of CLBs in this matrix is called a frame. The frame is the atomic unit of configuration. A specific frame can be reconfigured without affecting any other frame. Configuration time for the entire FPGA is similar to configuring other FPGAs of the same size, but many applications benefit from the ability to partially reconfigure. A trivial example is an application that only needs to change constant values used in a calculation. In this case just the frames containing the CLBs containing the values need to be updated.

Wormhole Run-time Reconfiguration is an alternative to the context switching and PRTR methods used by the WASMII, CSRC, and Virtex architectures. FPGAs, including WASMII and CSRC, typically have a centralized reconfiguration controller. This means that there is only one path to load configuration data into the chip. Wormhole RTR is based on the concept of streams. A stream consists of a programming header and operand data. The

programming header contains information on how to configure the different CLBs. As the stream traverses the chip, configuration information from the header is used to configure CLBs as they are reached. The CLBs forming the path of the stream perform calculations on the operand data. There can be multiple streams traversing a chip in parallel, each picking their own path through the chip based on their programming header. The ability to configure different parts of the chip concurrently decreases overall configuration time. The ability to only configure the CLBs necessary to implement an algorithm also decreases overall configuration time. The Wormhole RTR paradigm was used in the Colt CCM[14] and Stallion chip[15].

2.3 Caching

In addition to chip reconfiguration time, the time to swap configurations is increased by bus and network transfers. The ability to store, or cache, configurations on a reconfigurable computing board eliminates this additional overhead[16]. A finite number of configurations can be cached on the board. Configurations that are not currently cached on the board must be sent across the bus and possibly a network to the board, increasing reconfiguration time.

Multiple context architectures, such as WASMII and CSRC, provide a finite number of contexts. Switching to a configuration contained a context is very fast. Switching to a configuration not contained in a context results in configuration times similar to single context architectures.

Because many designs require more configurations than can be cached locally on the board or in contexts, methods of increasing the cache hit rate are desirable to cause a decrease in reconfiguration time. The Least Recently Used (LRU) cache replacement strategy is advocated for multiple context architectures for which the order for configurations is not predetermined in [17]. This technique can be extended to caching configurations on the board. With more in-depth knowledge of the order of configuration requests a much higher

cache hit rate can be achieved. If an algorithm requests configurations randomly, then a fifty percent hit rate is expected, with or without caching algorithms.

Caching of Reconfigurable Functional Unit Operations (RFUOPs) is discussed in [17]. RFUOPs present a finer level of caching granularity than entire chip configurations. Multiple RFUOPs may reside on an FPGA simultaneously. Methods of caching RFUOPs on the FPGA itself are proposed in [17]. Techniques are presented for single context, multiple context, and PRTR capable FPGAs. The focus of the single context caching strategy is to determine how to group RFUOPs into configurations to minimize the frequency of FPGA reconfigurations. The multiple context FPGA caching strategy is to determine groupings of RFUOPs that minimize the frequency of loading contexts from off-chip. The caching of RFUOPs on the FPGA differs from the method of caching entire configurations in RAM external to the FPGA used in this thesis.

Chapter 3

Platform

The hardware platform used in this thesis is the SLAAC-1V[8] reconfigurable computing platform designed by the University of Southern California's Information Sciences Institute (USC-ISI). Software designs use the SLAAC Application Programming Interface (API) to interact with the board. This chapter discusses the capabilities of the hardware and the interface provided to software applications by the SLAAC API.

3.1 Hardware

The SLAAC-1V platform is a printed circuit board designed to operate in conjunction with a host computer. Communication between the host and the board occurs via a standard PCI bus. The board has the capability of utilizing a 64-bit or 32-bit bus operating at 33MHz. Figure 3.1 gives an overview of the major components of the SLAAC-1V board.

The SLAAC-1V board is comprised of three Xilinx Virtex XCV1000 Field Programmable Gate Arrays (FPGAs)[13] labeled X0, X1 and X2. These FPGAs are referred to as Processing Elements (PEs). PEs X1 and X2 are fully user programmable. PE X0 contains logic that implements the PCI bus interface of the board so user designs have access to only a portion

of this FPGA.

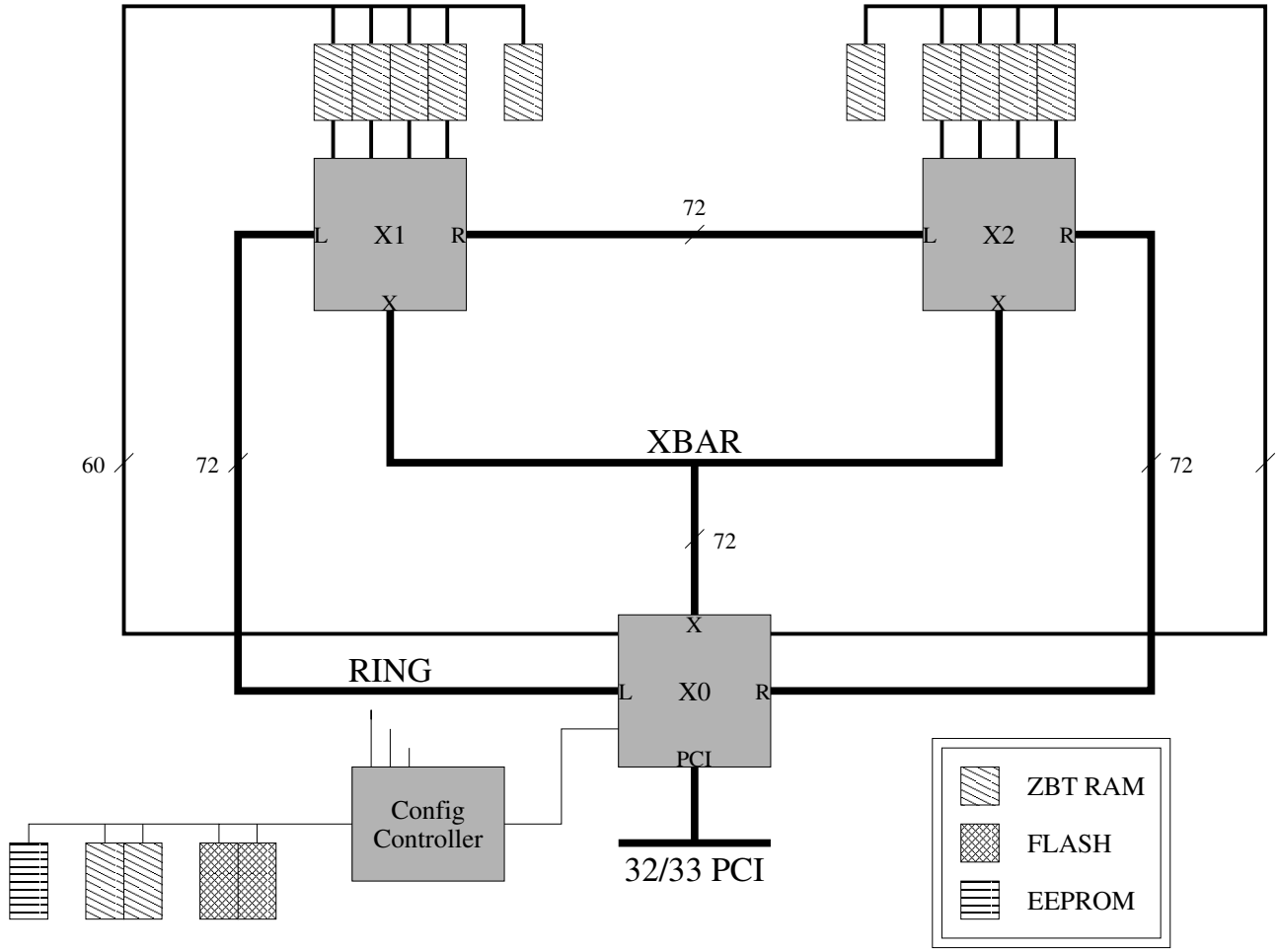

Figure 3.1: SLAAC-1V Platform

There are two clocks on the SLAAC-1V board. The MCLK is a 33MHz clock driven by the PCI bus. This clock is used to drive the PCI core in X0. The PCLK, or FPGA clock, is the main clock for user designs. The board is capable of synthesizing frequencies of 0MHz through 100MHz for the PCLK. There is also support for single-stepping of the PCLK.

The SLAAC-1V board contains two pairs of FIFOs for communication with the host. Each FIFO consists of an A and a B port. Data is enqueued to port A from the host and dequeued by the board. Data is enqueued to port B from the board and dequeued by the host. FIFO pair A0/B0 is a one-deep FIFO (mailbox FIFO) intended to be used for signalling between

the host and board. FIFO pair A1/B1 is a 256-deep FIFO (deep FIFO) intended for use as a data input/output path between the host and board. Both FIFOs are implemented in X0. From a user design perspective the FIFOs are both synchronous with the PCLK. The deep FIFO uses DMA transfers to and from the host to improve performance and decrease system resource consumption on the host machine.

There are two methods of inter-PE communication: a crossbar interconnect and a systolic array. The crossbar is 72 bits wide and connects all three PEs. It enables any PE to broadcast information to the other two PEs. Additionally the crossbar also provides access to expansion headers on the board. When the crossbar is configured to access the expansion headers broadcast functionality is disabled. The systolic array is also 72 bits wide. Because there are three PEs the array appears as a ring. Each PE has two 72-bit ports, labeled left and right, connected to the array. The left port of PE X0 is connected to the left port of PE X1. The right port of PE X0 is connected to the right port of PE X2. The right port of PE X1 is connected to the left of of PE X2. The systolic array is often used to route data through the two user PEs for processing. The results feed back into X0 at the end of the ring. This is especially convenient when used in conjunction with the FIFOs from the host because data can be streamed from the host through the ring and back to the host after processing by all of the PEs.

In addition to the FPGAs, the SLAAC-1V board contains static random access memories (SRAMs). Each of the user PEs, X1 and X2, has four independent memory units connected to it. Each memory bank is named for the PE and PE port it is connected to. Thus the memory connected to PE X2 memory port 2 is named X2M2. PE X0 has two independent memory units. Each memory has 256K 36-bit locations. Thus, each user PE has 4 * 256K * 36-bit = 1M 36-bit memory locations. In order to allow X0 to access all of the memories, the memories are not physically connected to the PEs. They are instead connected to a memory bus. It is possible to swap X0M0 with any of memories belonging to X1 and to swap X0M1 with any of memories belonging to X2. Typically when an X0 memory is swapped with

one of the user PE's memories the clock to the user PE is disabled. Because the clock is automatically disabled user designs do not need to contain extra logic to detect the swap status of memories that are accessed. This is referred to as non-preemptive memory access. Preemptive memory access is documented but not currently not supported on the SLAAC-1V. In this memory access mode, user designs have to verify they have physical access to a memory bank before using it. Generally the performance benefits of preemptive memory access are not high enough to justify the added complexity user designs require to implement it.

There are a series of registers on the SLAAC-1V that the host can manipulate to control and get feedback from the board. Additionally there are eight user registers in X0 that programs running on the host can use to interact with user applications on the board. Register read and writes are synchronous to the MCLK.

The board also contains a Xilinx Virtex 100 FPGA[13] functioning as a configuration controller. The controller has two 256Kx36-bit flash memories, two 256Kx36-bit random access memories, and an EEPROM attached to it. The EEPROM contains the configuration for the configuration controller. Upon power-up, the configuration is loaded into the Virtex 100 part from EEPROM. At that point the configuration controller initializes the rest of the board. The first 1MB of flash memory is reserved for the startup PE X0 configuration. Because the PCI interface for the board is implemented in X0, X0 must be configured before the board can communicate with its host computer. The configuration controller is responsible for configuring PE X0 on power-up from the flash memory.

The SLAAC-1V can be hosted in a computer running either Microsoft Windows NT or Linux. The results presented in this thesis were produced using a board hosted in a Debian Linux 3.0 machine with a 32-bit PCI bus.

3.2 Software

In order to use the SLAAC-1V, a user must design both an application to run on the board and software to run on the host. The host software interface is the SLAAC API . As shown in Figure 3.2, all user interaction with the board from the host takes place through this API. The API calls can be divided into two categories: data transfer functions and board control functions.

There are three methods of communicating with the SLAAC-1V board: streaming data through the FIFOs, reading and writing on-board memory, and passing data via the user registers.

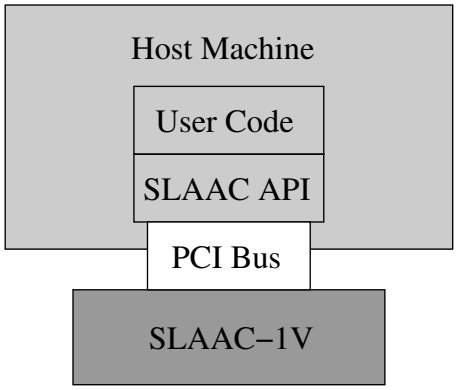

Figure 3.2: SLAAC API System Hierarchy

The FIFOs are the primary streaming data input/output path. Because the SLAAC API and the board implement the logic necessary to coordinate FIFO access, it is not necessary for user designs to contain complicated control logic to interface with the host. In order to provide maximum throughput, data is sent to and received from the deep FIFOs using DMA transfers. This enables queuing of FIFO operations in order to provide a constant stream of data to the board. Similarly, dequeue operations can be queued so that the board does not stall while waiting for the host.

Memory access functions allow applications to read from and write to the on-board memories.

Access is fast due to the fact that DMA transfers are used, though due to complexities of coordinating memory access between applications and on-board logic, most streaming applications do not use the memories to pass data to and from the board. The primary use of memories in the experimental applications discussed in this thesis is to cache configuration data. In these experiments the memory access is either under host control or, when under board control, the host does not interact with the memories once all configurations have been loaded into cache.

The user register functions provide another avenue of data transfer. There are eight user registers. Data can be sent and received using these registers, however the throughput rate will be lower than that of the FIFO and memory access functions. The user registers are useful for retrieving status information from the board and for sending short commands to the board. They can also be used for data exchange when high performance is not necessary.

SLAAC API board control functions allow applications to setup and alter operation of the SLAAC-1V board. Functions are provided that allow applications to load configuration bitstreams into the PEs, assert and deassert reset lines to the PEs, enable and disable the on-board clock, and set the clock frequencies.

In addition the control functions listed above, functions are provided to aid in debugging. Applications can single-step the FPGA clock. The API accepts a clock rate and the number of times to step the clock. The user can use this function to step the clock cycle by cycle. After each step user registers can be polled to get debugging information from the user design in X0. The more esoteric "fifostep" mode allows the application to specify how many cycles to step the FPGA clock after an enqueue to the deep FIFO. This greatly aids in debugging applications that process data from the deep FIFO. Normally such designs need to run at normal clock speeds until data is received. However, once data has arrived the user may want to step the clock. The "fifostep" functionality enables this behavior.

Chapter 4

ACS API

The Adaptive Computing Systems (ACS) API developed at the Virginia Tech Configurable Computing Machinery (CCM) Lab enables users to easily implement designs spanning multiple reconfigurable boards[1]. It currently supports the SLAAC-1V[8] and Osiris[7] boards from USC-ISI, the Reconfigurable Computer Module (RCM)[12] designed by Lockheed Martin, and the WildForce board manufactured by Annapolis Microsystems[9]. This API is designed to control systems comprised of multiple computers each hosting a reconfigurable computing board. There is no requirement that these systems be homogeneous. That is, hosts in the system may host different types of configurable computing boards. The API handles all network communication without application intervention. This chapter outlines the function of that API, and contributions made to the API.

4.1 Overview

The ACS API interfaces user programs to the board specific APIs. Applications using the ACS API consist of a *host program* and configurations for the reconfigurable computing boards. The host program may be written in C, C++, or TCL. Interfaces for other language

are easily added. The host program calls ACS API functions to setup the system, to configure boards, and to perform data transfers. The ACS API provides access to all common functionality of the supported configurable computing boards.

A possible system based upon the ACS API is shown in Figure 4.1. A user program developed with the ACS API runs on a client machine that may or may not have a reconfigurable computing board in it. Each board in the system is called a node, and every node has a unique identifier. User code is able to interact with any node in the system. All API functions that interact with a board take a parameter that specifies for which node the command is intended. Therefore, user code can control every board in the system.

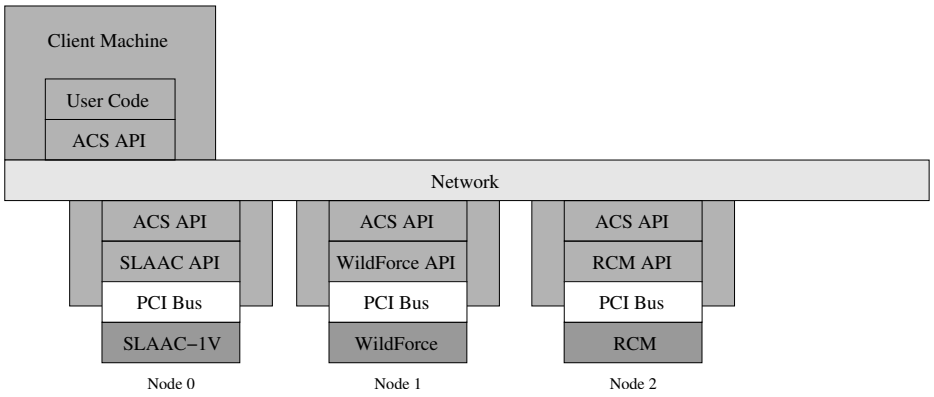

Figure 4.1: ACS API System Hierarchy

The API uses the concept of *channels* to interact with the individual board FIFOs. Channels provide a convenient interface to the various board FIFOs in a system. Applications use API calls to describe the route a channel takes through the system. The route of a channel is described as a pair of node locations and FIFO ports. Every channel must start at a data producing port; either a port on a board that produces data or the provided host port that enables applications to input data to the channel. Every channel must end at a data receiving port; either a port on a board that accepts data or the provided host port that allows applications to retrieve data from the channel. Channels allow applications to route FIFO data through whatever combination of boards they desire.

The ACS API uses the Message Passing Interface Standard[18] (MPI) to implement inter-node communication. When using MPI, an ACS *control process* is started on every node in the system. The control process receives commands from the host program via MPI and performs actions on the local board based on those commands. This communication is hidden from the host program by the API.

Not every application requires multiple boards, so the ACS API can be used with a single board without using MPI. In this mode using the ACS is similar to using the native board API. The communication latency of the MPI is avoided in this mode, so higher performance can be achieved.

4.2 Functions

The ACS API functions can be placed in three major categories: system management, board management, and data management. System management functions are used to setup and destroy the ACS system. Board management functions are used to set clock parameters and reset boards. Data management functions are used to enqueue and dequeue data from a channel.

4.2.1 System Management

System management functions are used to initialize and configure the ACS system, query the topology of the system, and shutdown the system. The system management functions are listed in Table 4.1.

MPI requires a configuration file that specifies all nodes in the system and the process to start on each node. In addition, the ACS requires host programs to specify all nodes and channels in the system. In order to avoid this error-prone and tedious task, the ACS has

Table 4.1: ACS System Management Functions

Name	**Description**
ACS_Initialize	Takes command line arguments and initializes the ACS.
ACS_System_Create	Creates the system. Parameters specify the nodes and channels.
ACS_System_Destroy	Destroys ACS system.
ACS_Finalize	Shutdown the ACS.
ACS_Channel_Add	Add a new channel to the system.
ACS_Node_Add	Add a new node to the system.
ACS_Is_Local	Is the specified node local?
ACS_Is_Remote	Is the specified node remote?
ACS_Get_Board_Info	Returns information about the board, including information on PEs and memories.

the capability to read an XML configuration file, an example of which is shown in Listing 4.1. The XML file contains a list of all board in the system and their locations. It may also contain configuration files to load to the PEs on the boards. If an application is using the onboard FIFOs, the channels can be specified in the configuration file. After processing the XML file the ACS generates the MPI configuration file if necessary, initializes the ACS, and creates the ACS system.

4.2.2 Board Management

The board management functions listed in Table 4.2 allow applications to configure board parameters. Variants of these functions are provided to simplify ACS applications. ACS_Configure configures PEs with configuration data passed to it. The complementary function ACS_Load_Configuration_ loads a PE configuration from a file. ACS_ConfigureFromFile configures a PE with configu-

Listing 4.1: XML Configuration File Example

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE config SYSTEM "./acs_conf.dtd">
<config>

<header>
<commtype>MPI</commtype> <!-- could also be NONE if MPI is not needed -->
<version>0.03</version>
<author>James Hendry</author>
<desc>Example File</desc>
</header>

<boards>
<slaac1v board_id="node0" location="local" ctrl_proc_dir="bin/">
<PE pe_num="0" prog_file="x0.bit"/>
<PE pe_num="1" prog_file="x1.bit"/>
<PE pe_num="2" prog_file="enigma.bit"/>
</slaac1v>
</boards>

<channels>
<channel>
<src><host port="HOST_OUT"/></src>
<dest><node board_id="node0" port="SLAAC_FIFO_A1"/></dest>
</channel>

<channel>
<src><node board_id="node0" port="SLAAC_FIFO_B1"/></src>
<dest><host port="HOST_IN"/></dest>
</channel>
</channels>

</config>
```

Table 4.2: ACS Board Management Functions

Name	**Description**
ACS_Configure	Configure a PE
ACS_Reset	Control reset signals
ACS_Clock_Set	Configure clock parameters
ACS_Run	Start the clock on a board
ACS_Stop	Stop the clock on a board
ACS_Set_FSM	Configure an onboard FSM

ration data from a file. ACS_Reset_Toggle toggles the state of a reset signal.

The API has a "register" interface to support board-specific functionality. Instead of having API calls that are not implemented for some platforms, all special board functionality is wrapped in register read and writes. An example is special functionality for Lockheed Martin's RCM board. The RCM board is capable of caching four configurations for each PE and instantly switching between them[12]. To enable this unique functionality for the RCM an ACS register is used. Values written to this register from user programs specify which PE to reconfigure and what cached configuration to use. The registers can also be used to extend the functionality of the ACS API in conjunction with user designs.

4.2.3 Data Management

Data management functions allow applications to send data to and receive data from the boards. The functions in Table 4.3 provide access to the onboard memories and FIFOs. All boards that the ACS supports contain memory and FIFOs.

Table 4.3: ACS Data Management Functions

Name	**Description**
ACS_Read	Retrieve data from onboard memory
ACS_Write	Write data to onboard memroy
ACS_Enqueue	Enqueue data into a channel
ACS_Dequeue	Retrieve data from a channel

4.3 Caching

Each board in a multiple board system can be placed in one of two categories: local node or remote node. A local node is a board contained in the machine running the user program. There may not be a local node in the system. Only one local node can be in the system. Remote nodes are boards in computers not running the user program. In a multiple board system there is at least one remote board.

Configurations for remote nodes must be sent over the network. The need to send the configuration over the network significantly slows the configuration process. Contemporary FPGAs can be configured a byte at a time with a 50 MHz clock[13]. This is equivalent to a 400 Mb transfer rate. This is slower than modern gigabit network transfer rates, but the network still presents a limitation because entire configurations are transferred over the network before being sent to the board. This means that any network delay is added to the reconfiguration time. For applications that require the same configurations repeatedly, a system of caching configurations to decrease network transfers is desirable.

The ACS API currently caches configurations at the *control process* level in multiple board systems. A cache of configurations is kept on all remote nodes. The host node tracks the contents of the caches on the remote nodes. When configuration of a remote node is requested by the user program, the host first determines if the remote node has the configuration

cached. If it is determined that the configuration is cached on the remote node, a command is sent to the remote node instructing it to configure from cache. Otherwise the configuration is sent over the network.

4.4 Example ACS API Program

Listing 4.2 shows an example ACS API program written in C++. The program starts by using the ACS_XMLConfig object to configure the system based on an XML configuration file specified at the command line. There are no calls to ACS_Initialize or ACS_System_Create because ACS_XMLConfig calls them. After the system is initialized and created, one kilobyte of data is read from a file. A 10 MHz clock rate is then selected. The reset signal is asserted and then the clock is enabled with a call to ACS_Run. The reset signal is then deasserted. The one kilobyte of data read from a file is now enqueued to channel zero. ACS_Dequeue is called to read the one kilobyte of data now processed by a board. This data is saved to a file. Calls to ACS_System_Destroy and ACS_Finalize shutdown the ACS API.

4.5 Contributions

A number of modifications were made to the ACS API during the course of the work presented in this thesis. Originally, the API was compiled from source using a custom Perl script. The resulting complexity of using a non-standard method of compiling code slowed the progress of new users of the API. Therefore, the build system was converted to using traditional Makefiles. This change simplifies code changes and presents a build process with which users are generally familiar.

The ACS_Load_Configuration_File and ACS_ConfigureFromFile functions were added to the API. Without these functions, every user application must contain custom, board specific

code to load configurations from disk. Because almost every application requires this capability, these functions were provided to isolate user applications from the differences in configuration file formats of the different boards. The ACS_ConfigureFromFile call provides the added benefit of only having to send a filename to remote nodes, not the entire configuration.

The ACS_Reg_Read was modified to increase the flexibility of the register interface. Originally the only input parameter was a single integer. The buffer populated by the function could not be used to send extra information. This limitation was removed. The limitation only affected register access to remote nodes. To remove the limitation, the buffer passed to the function is sent over the network to remote nodes. This improves the functionality of the register interface, and was a requirement of the application presented in Chapter 5.

Listing 4.2: ACS Application Example

```
ACS_STATUS     status;
ACS_SYSTEM     * system;
ACS_XMLConfig  config;
ACS_CLOCK      clock;
unsigned       buffer[256];

config.setFilename(argv[1]);
config.getSystem(&system);   // system created based on XML parameters

ifstream iFile(''data.in'');
iFile.read((char*)buffer, 1024);  // read 1024 bytes from data.in

clock.frequency = 10;  // 10 MHz
clock.countdown = 0;   // run forever
clock.clockId   = 1;   // user clock
ACS_Clock_Set(&clock, 0, system, &status);
ACS_Reset(0, system, 1, 1, &status); // reset node 0
ACS_Run(0, system, &status);         // start clock on node 0
ACS_Reset(0, system, 1, 0, &status); // deassert reset on node 0

ACS_Enqueue(buffer, 1024, 0, system, &status); // enqueue 1024 B
ACS_Dequeue(buffer, 1024, 1, system, &status); // dequeue 1024 B

ofstream oFile(''data.out'');
oFile.write(buffer, 1024);       // write 1024 bytes to data.out

ACS_System_Destroy(system, &status);
ACS_Finalize();
```

Chapter 5

Implementation

This chapter describes the implementation of configuration caching on the SLAAC-1V board. Section 5.1 describes the framework for caching on the SLAAC-1V. Section 5.2 discusses an application utilizing the caching framework. This application is used to obtain the results presented in Chapter 6.

5.1 Caching Architecture

Figure 5.1 shows the caching architecture implemented on the SLAAC-1V. Caching configurations on the board is implemented using three components: the XVPI_PROG module, a finite state machine, and on-board memories.

5.1.1 Operation

Configurations are cached in the Port 0 memories of PE X0 on the SLAAC-1V. Port 0 has access to local memory bank X0M0, and all four PE X1 memories: X1M0, X1M1, X1M2, and X1M3. Configurations for the Xilinx XCV1000 are approximately 766 kilobytes in length.

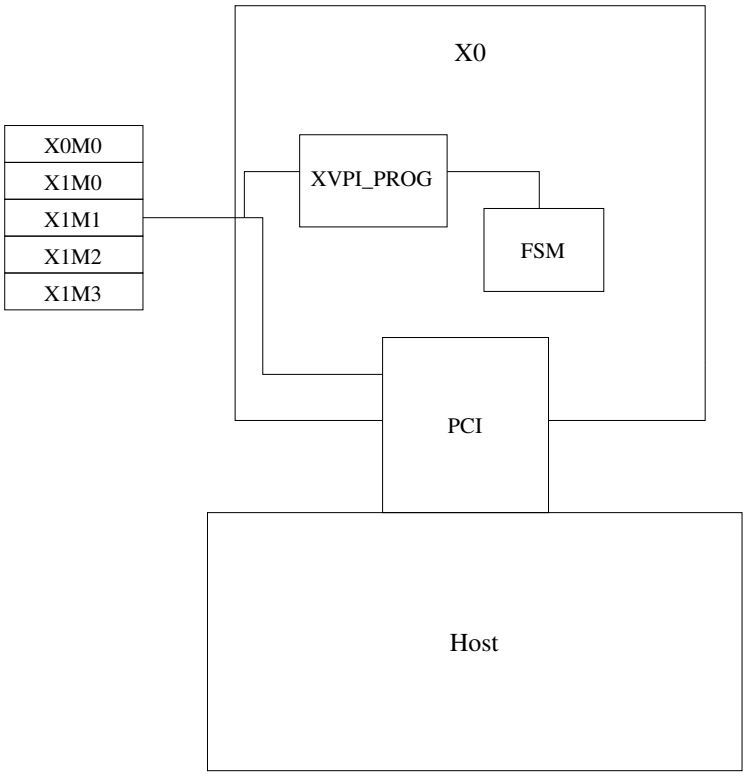

Figure 5.1: SLAAC-1V Caching Architecture

Each SLAAC-1V memory can hold one megabyte. This implies that up to six different configurations can be cached on the board. In order to simplify the hardware design it was decided that only one configuration would be stored in a memory. As a result, five memories yield five cache slots. Each Port 0 memory is defined here as a cache slot. The cache slots are labeled with the numbers zero through four inclusive. Table 5.1 shows the mapping between cache slots and memories.

The default firmware of the SLAAC-1V does not support configuring a PE from an onboard memory. Modifications to the firmware made at Virginia Tech supplied this capability[19]. The SLAAC-1V firmware provides PE X0 designs access to the configuration ports of PEs X1 and X2 through the Xilinx Virtex Portable Interface[20] (XVPI). This functionality allows X0 designs to configure X1 and X2 without intervention from the host.

The XVPI_PROG module designed at Virginia Tech contains the logic necessary to configure

Table 5.1: Cache slot memory mappings

Slot	**Memory**
0	X0M0
1	X1M0
2	X1M1
3	X1M2
4	X1M3

X1 and X2[19]. It is implemented in VHDL. The module is added to PE X0 designs. The module reconfigures PEs with configuration data supplied by the user design.

5.1.2 Control

A finite state machine (FSM) is implemented in X0 to control PE configuration. The FSM is defined by several parameters. Parameters consist of the number of states in the FSM and the number of bits in the input to the FSM. All parameters of the FSM are user configurable. State transitions and state actions are stored in internal FPGA RAM resources. The memory layout is shown in Figure 5.2. Transitions are made based on the current state and input. When a state transition occurs the action to perform is read from the memory based on the new state. The action is then performed.

Actions are specified with a four bit value. Bit three selects the device to configure, X1 or X2. Bits two downto zero select the memory to configure from: X0M0 (000), X1M0 (001), X1M1 (010), X1M2 (011), or X1M3 (100). If the bits are 111 then no configuration is performed.

User designs are responsible for configuring the FSM by modifying the FSM configuration RAM. Because state transitions and actions are stored in a RAM, the FSM may be modified at runtime.

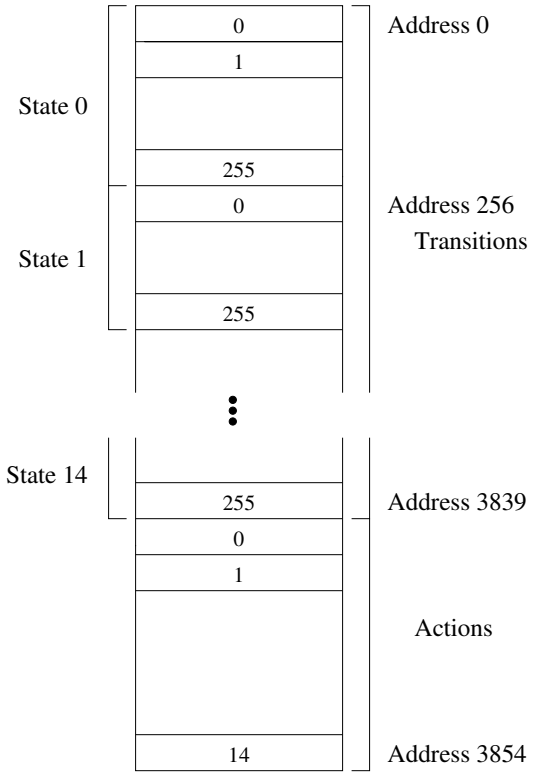

Figure 5.2: FSM Configuration Memory for 15 states with 8-bit input

5.2 Application

5.2.1 Overview

The Enigma encryption algorithm was used by the German military and cracked by England during World War II[21]. The algorithm was implemented on a machine consisting of a twenty-six key keyboard, three scrambling rotors, a reflector, and twenty-six lights representing alphabetic characters. A character was encrypted by pressing its corresponding key on the keyboard and retrieving the encrypted character from the lights. After every letter the rotors were rotated. Each rotor was a cylindrical disk with fifty-two electrical contacts on each side.

There is a one-to-one mapping between the electrical contacts on either side of the disk. This

mapping is different for every rotor. When a key on the machine is pressed exactly one of the contacts on the first rotor is connected to electrical current. This current flows through all three rotors traversing one pair of contacts on each rotor. After traversing the three rotors, the current is reflected back through the rotors. The lights are connected to twenty-six of the contacts on the first rotor. After traversing all three rotors twice, the current flows through a light thereby displaying the encrypted character. The encryption scheme relies on limited access to the encryption hardware and the secrecy of the starting positions of the rotors. Enigma is a symmetric encryption algorithm, meaning that decryption is performed by the same operation as encryption.

5.2.2 SLAAC-1V Enigma Implementation

A modified version of the enigma algorithm is implemented in this document. Rotors are simulated using FPGA hardware resources. The rotor simulations encrypt eight bits of data instead of twenty-six characters. Each rotor configuration is implemented twice: once for data travelling to the reflector and once for data travelling from the reflector. Using different rotor configurations yields multiple encryption keys. Multiple encryption engines are used in parallel to encrypt 32-bits of data at a time.

The hardware implementation consists of the enigma encryption algorithm in PE X2, X2 configurations cached onboard, the FSM in X0 controlling reconfigurations, and software running on the host. The hardware architecture is shown in Figure 5.3. Three different host programs control the SLAAC-1V board: one using the SLAAC-API, one manually controls caching using the ACS API, and one using the ACS API to automatically manage caching. The programs behave identically save for the differences in API function calls and control of the cache. The ACS API implementations can optionally use node level caching when using a remote node.

A host program is responsible for loading configurations to the onboard memories, program-

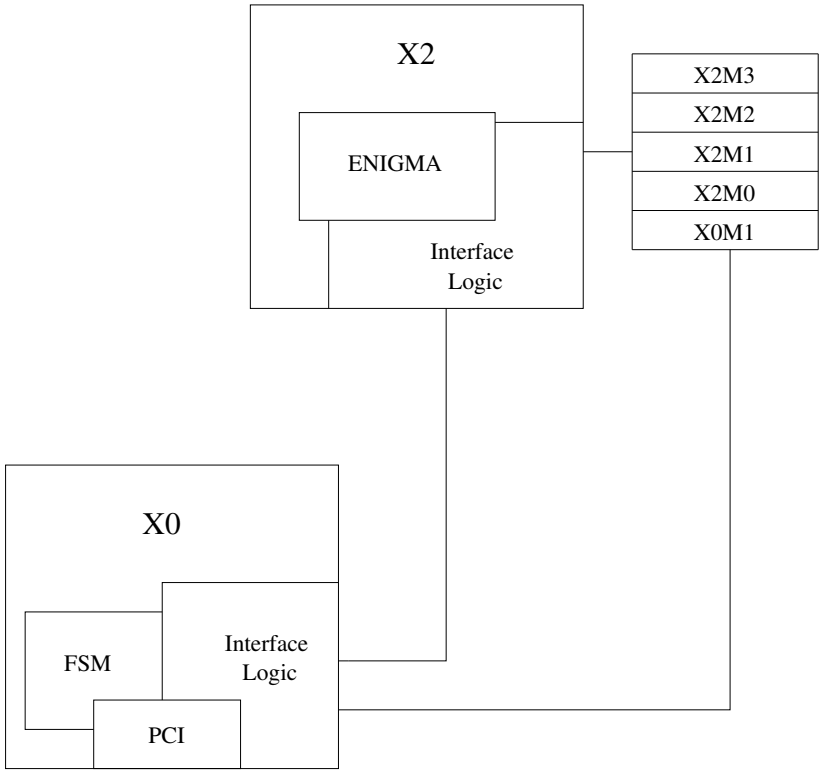

Figure 5.3: Enigma Caching Hardware Overview

ming the FSM, sending unencrypted data to the board, and retrieving encrypted data from the board. The host program encrypts files by streaming the data to the SLAAC-1V and retrieving the encrypted data. Program behavior is dictated by a configuration file, an example of which is shown in Listing 5.1. The configuration file specifies the configurations to load to the cache slots and what files to encrypt. It also contains options specifing whether caching will be used and if all files may be encrypted in parallel. The cache slot containing the desired encryption configuration is specified for each file.

If caching is not selected then all configurations are done from the host using the appropriate API call, otherwise all configuration is done from the on-board cache slots. If parallel file encryption is selected all input files are read into memory before being encrypted and all results are saved in memory before being saved to output files. This allows file access times to be removed from the timing results. If parallel file access is not selected then each file is

opened, encrypted, and saved to disk sequentially. This allows one file to be passed through multiple layers of encryption by repeatedly encrypting the result of the previous encryption step.

As discussed in Chapter 3, the SLAAC-1V has two data transfer mechanisms: FIFOs and memories. For a streaming application such as encryption, the deep FIFO seems like the logical choice for data transfer. Unfortunately the small depth of the deep FIFO limits its usefulness. The deep FIFO contains 256 64-bit locations. The smaller the packet size sent to the board the more overhead there is setting up PCI bus transfers. Using memory allows the host to send packets as large as one megabyte to the board, significantly decreasing overhead.

The board is configured to encrypt *packets* of data. A packet consists of data to be encrypted, information about the length and location of the data, and FSM inputs. Data to be encrypted is written to X2M0. Encrypted data is retrieved from X2M1. After a packet has been written to X2M0, the host notifies the board of the location and length of the packet. The packet information received from the host is buffered with a FIFO implemented in X0. This FIFO serves two purposes. The primary use is to allow the host to send packets at any time. Multiple packets may be sent before any are retrieved. Up to 1024 packets may be buffered by the FIFO. The secondary purpose of the FIFO is to transition between clock domains. The packet information is sent by the host synchronously with the PCI bus clock. User designs are tied to the USER clock. Data can be written to the FIFO on PCI clock transitions and read from the FIFO on USER clock transitions.

The FSM is configured to have fifteen states with an input width of eight bits. When X2 is ready to accept a packet, the packet information is read from the FIFO. The new FSM inputs may cause a reconfiguration, if this is the case the packet data is not sent to X2 until after reconfiguration is complete. Once X2 is ready the packet data is sent to it from X0. A signal is sent to X0 when X2 completes processing the packet. X0 and X2 communicate using the systolic array. X0 then increments a packet counter that is accessible by the host.

Table 5.2: User Register Description

Register	**Description**
0	Receives length and location of data to encrypt.
1	Receives location to put encrypted data and inputs for the FSM.
2	Program and control FSM.
3	Send number of packets encrypted.

The host uses the packet counter to determine when packets are ready to be retrieved from X2M1.

All control and signalling communication between the host and board is done via the SLAAC-1V user registers mentioned in Chapter 3. Table 5.2 shows how the enigma implementation utilizes the user registers from the perspective of the board. Figure 5.4 shows the architecure of the X0 design.

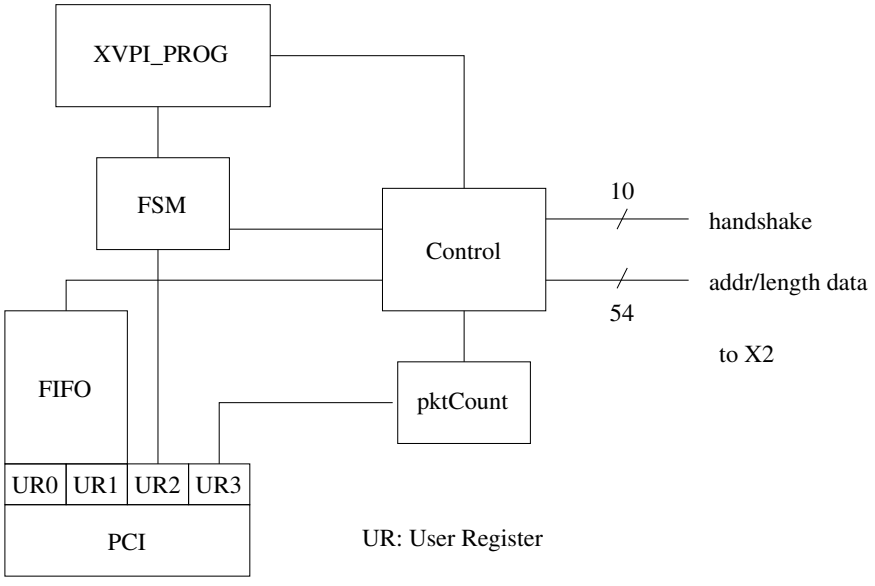

Figure 5.4: X0 Architecture

The host application is designed to get maximum throughput. Instead of sending a data

Table 5.3: FIFO Manager Interface

Name	**Description**
SendPkt	Enqueues a packet of data to be encrypted with specified cache slot.
RetrvPkt	Retrieve the first available packet of encrypted data. Also returns length of packet.
RetrvReady	Returns true if any packets are ready to be dequeued.

packet and waiting for the board to process it, the host application sends packets continuously while retrieving packets as soon as they become available. This helps ensure the enigma core does not stall waiting on data. The host application polls user register three to determine when packets are available.

5.2.3 FIFO Manager

The use of memories to stream data to and from the board is hidden from the host application by the FIFO Manager (FM). The FM is a C++ class that presents a FIFO like interface to the host application. This interface consists of the methods in Table 5.3.

The FM is implemented for both the SLAAC API and the ACS API, presenting the same interface to the host application in both cases. This simplifies the host application. The code that initializes the board at startup is API specific, but the code that streams data to and from the board is identical because it only interacts with the FM. The FM, in conjunction with the hardware design, can be used by other applications to emulate FIFOs with memories.

The FM is responsible for placing packets in X2M0 and then notifying the board that a new packet is ready. The FM must also track packet sizes so that the correct amount of data can be retrieved from X2M1 from the correct location.

5.2.4 ACS API

Modifications to the ACS API were required to enable caching functionality. Changes were limited to the SLAAC-1V specific code in the ACS. The ACS API provides a standard method to configure an FSM: ACS_Set_FSM. This function was implemented for the SLAAC-1V.

As discussed in Chapter 4, the ACS API provides a register interface to access board specific functionality. Applications may write or read a block of data of any length to ACS registers. The data format is specified by the ACS. ACS register three is used to provide access to the SLAAC-1V user registers. The ACS register uses a two word data packet. The first word specifies which SLAAC-1V user register to access. The second word is used either to specify data to write to a user register or to receive data from a user register. This functionality is used by the ACS API version of the host application.

Listing 5.1: Configuration File Format

```
# Host program configuration file
# SEQ_FILES: set false if all input files can be loaded then encrypted
# USE_CACHE: set true to use cache
# 0 = false, 1 = true
BEGIN
USE_CACHE        0
SEQ_FILES        0
END

# Configurations are listed here.  Numbering starts at 0
BEGIN
encrypt0.bit
encrypt1.bit
encrypt2.bit
encrypt3.bit
END

# Files to encrypt are listed here
# ConfigurationNumber TAB inputfilename TAB outputfilename
BEGIN
0       input1          output1
1       input2          output2
2       input3          output3
3       input4          output4
END
```

Chapter 6

Results and Analysis

This chapter presents the results of experiments run to assess the benefits of using caching on reconfigurable computing boards, namely, the SLAAC-1V board. A theoretical model is presented that calculates the effects of caching on throughput in data streaming applications. Experiments are presented that verify that both board level and node level caching do improve performance. Results are compared to the theoretical expectations of the effects of caching. Finally, the anticipated affects of caching on other configurable computing platforms is discussed.

6.1 Theory

The throughput of a streaming application is measured in megabytes per second. The baseline for throughput performance is P_0. P_0 is the system throughput with no reconfigurations. Throughput is decreased with increasing reconfigurations because while a processing element is being reconfigured it is not processing data. Therefore, throughput is affected by both the frequency of reconfigurations and the time to reconfigure. These effects are given by the expression,

$$P_r = \frac{S_{avg}}{\frac{S_{avg}}{P_0} + R_{reconfig} T_{avg}}, \tag{6.1}$$

where S_{avg} is the average packet size of data sent to the board, T_{avg} is the average time of reconfiguration, $R_{reconfig}$ is the ratio of reconfigurations to no reconfiguration, and P_r is throughput adjusted for reconfiguration. A packet of data is processed with no opportunity for reconfiguration. Reconfiguration may occur at the boundary between packets. T_{avg} is computed using the classic cache performance formula,

$$T_{avg} = T_{hit} + R_{miss} * T_{penalty}[22], \tag{6.2}$$

where T_{hit} is the reconfiguration time in the event of a cache hit, P_{miss} is the ratio of cache misses and cache hits, and $T_{penalty}$ is the reconfiguration time penalty incurred by a cache miss.

Equation 6.1 shows that throughput rate is controlled by S_{avg}, $R_{reconfig}$, and T_{avg}. Equation 6.2 shows that T_{avg} is controlled by T_{hit}, R_{miss}, and $T_{penalty}$. Therefore, overall throughput, P_r, is controlled by five variables.

Figure 6.1 shows the ratio of P_r and P_0 in relation to $R_{reconfig}$ for various values of S_{avg} with $T_m = 200$ ms. From this graph, it is evident that increasing the size of packets diminishes the influence of average configuration time and $R_{reconfig}$ on P_r. Thus, increasing packet size, S_{avg}, is an effective means of increasing throughput.

The time penalty per packet due to reconfiguration is given by the expression,

$$T_{pp} = R_{reconfig} * T_{avg}. \tag{6.3}$$

Minimizing the time penalty per packet is the alternative to increasing S_{avg} to increase P_r. Assumming T_{hit} is constant, T_{pp} can be decreased by reducing $R_{reconfig}$, R_{miss}, or $T_{penalty}$.

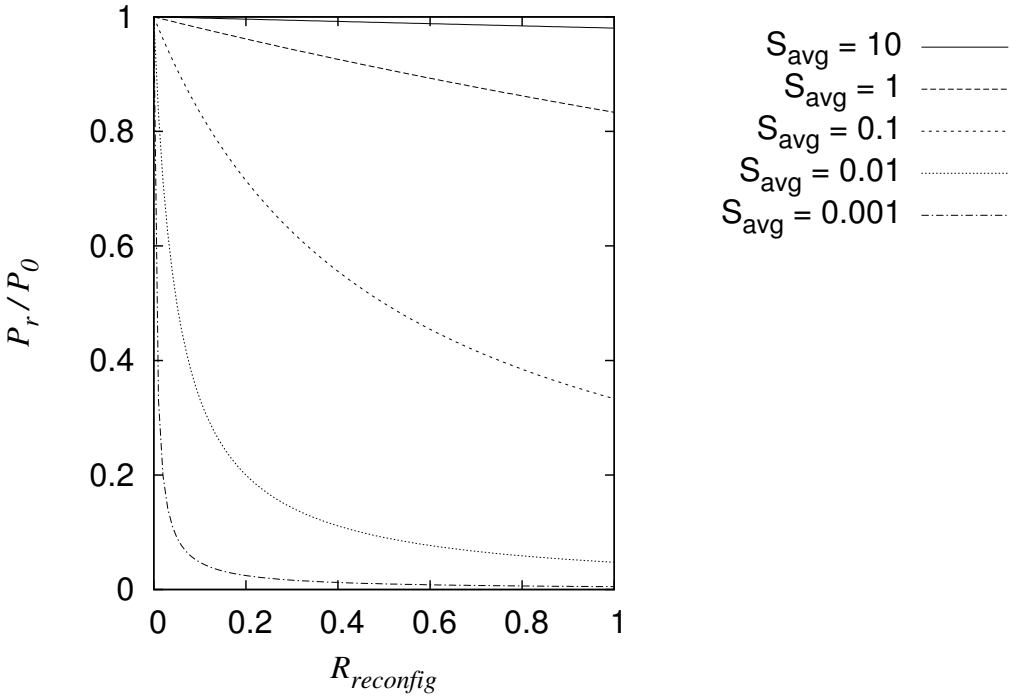

Figure 6.1: Performance versus $R_{reconfig}$ with respect to S_{avg}

Figure 6.2 shows the ratio of P_r and P_0 in relation to $R_{reconfig}$ for various values of T_{avg} with $S_{avg} = 0.1$ megabyte. T_{avg} decreases with decreasing cache miss rate or by reducing the time penalty for a cache miss.

In summary, it is evident from Equations 6.1 and 6.2 that decreasing the cache miss rate, reducing the cache miss time penalty, decreasing the ratio of reconfigurations to configurations, and increasing data packet size all increase the throughput of the system. This thesis focuses on reducing T_{avg} by minimizing the cache miss time penalty, $T_{penalty}$. Applications using the caching scheme implemented in this document can increase throughput by increasing the data packet size or by decreasing the reconfiguration ratio.

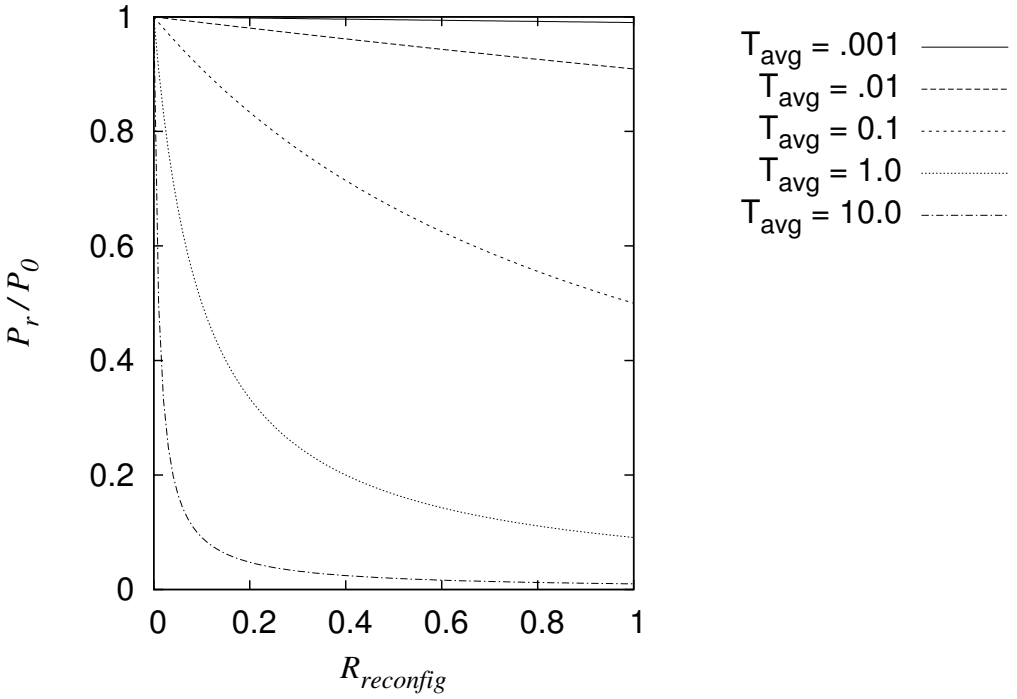

Figure 6.2: Performance versus $R_{reconfig}$ with respect to T_{avg}

6.2 Empirical Results

6.2.1 Reconfiguration Time

Empirical results are presented for a number of scenarios. Experiments are run with or without caching using both the SLAAC API and ACS API. For the ACS API, results are obtained for both local node and remote node situations. Remote node experiments are run without caching, node level caching only, and full node and board level caching. Results are also presented for using automatic cache management with the ACS.

Table 6.1 shows empirically measured values for reconfiguration times for a local board using the SLAAC API and ACS API. Very little overhead is introduced by using the ACS API with a local board. A suprising result is that reconfiguration time for a cache miss is much

Table 6.1: Local Board Reconfiguration Times

	SLAAC API	**ACS Manual**	**ACS Automatic**
No Caching	319ms	320ms	X
Cache Miss	145ms	146ms	153ms
Cache Hit	88ms	88ms	88ms

Table 6.2: Remote Board Reconfiguration Times

	ACS Manual	**ACS Automatic**
Total Cache Miss	230ms	221ms
Board Cache Hit	108ms	87ms

less than not using caching at all. This is counterintuitive because a cache miss involves two steps: (1) loading the configuration into a cache slot and (2) configuring from the cache slot. This performance gain is due to the fact that memory writes use direct memory access (DMA) transfers across the PCI bus, while the SLAAC API *Configure* function does not use DMA and transfers only one byte per PCI transfer. On average, a cache hit decreases reconfiguration time by forty percent over a cache miss.

Table 6.2 shows empirically measured values for reconfiguration times for a remote board using the ACS API. Results indicate that the automatic cache management mechanism of the ACS API decreases reconfiguration times relative to manually controlling the cache. This is expected because the automatic cache management code performs more work on the node local to the board, whereas manual cache management must execute entirely on the the root node relying on the network to communicate with the board.

Table 6.3: Throughput Rates for Local Board

	SLAAC	**ACS**
No Reconfiguration	5.49 MB/s	5.53 MB/s
No Caching	2.00 MB/s	1.99 MB/s
With Caching	3.71 MB/s	3.71 MB/s

6.2.2 Throughput

The enigma encryption application presented in Chapter 5 processes streams of data. The rate of data transfer is called throughput. In this document, throughput is measured in megabytes per second (MB/s). Performance of streaming applications is determined by throughput.

Table 6.3 shows empirically measured throughput rates for different scenarios using a local board. Results are from a 300 MHz Pentium II host with 128 MB of RAM hosting a SLAAC-1V board. The user clock on the board is set to 35 MHz. Packet size is one megabyte. The cache miss rate, R_{miss}, is 0, and the reconfiguration ration, $R_{reconfig}$, is 1 for all results.

Figure 6.3 shows the predicted throughput and actual throughput utilizing the SLAAC API without caching for different packet sizes on a local board. $R_{reconfig}$ is set to one in order clearly show the effects of caching. Using the empirical data from Tables 6.1 and 6.3, T_{avg} = 319 ms and P_0 = 5.49 MB/s. With these values, P_r can be calculated from the average packet size, S_{avg}, using Equation 6.1. Figure 6.3 shows that actual performance correlates with predicted performance.

Figure 6.4 shows the predicted throughput and actual throughput utilizing the ACS API with caching for different packet sizes on a local board. $R_{reconfig}$ is set to one in order clearly show the effects of caching. Using the empirical data from Tables 6.1 and 6.3, T_{avg} = 88 ms and P_0 = 5.53 MB/s. With these values, P_r can be calculated from the average packet size,

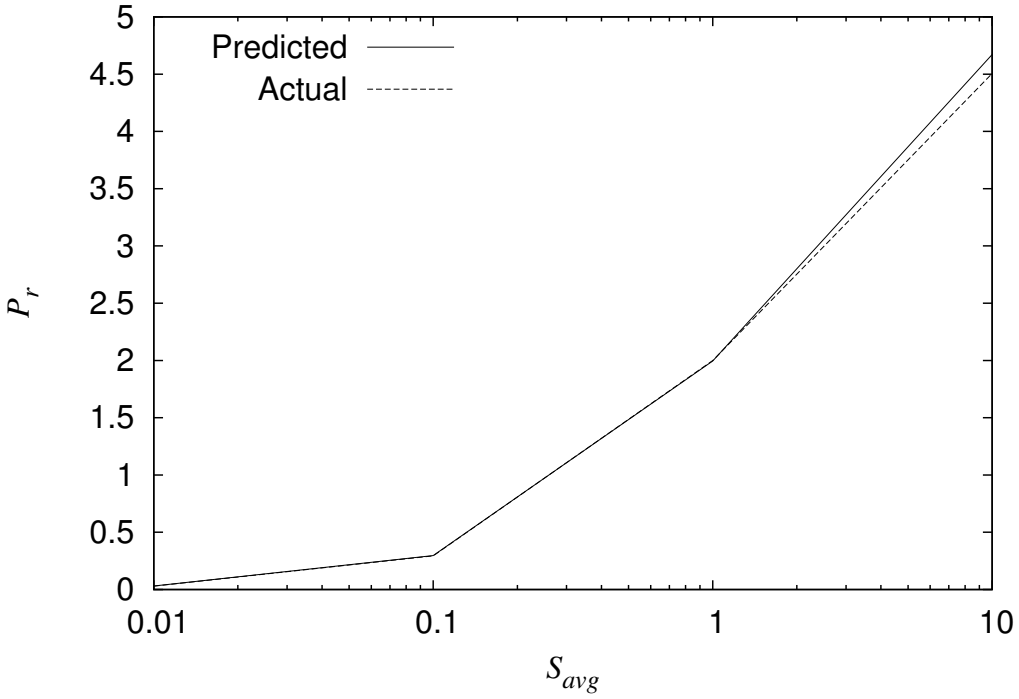

Figure 6.3: Throughput v S_{avg} for SLAAC API without caching

S_{avg}, using Equation 6.1. Figure 6.4 shows that actual performance correlates with predicted performance.

6.3 Analysis

The time to configure from cache is limited by the user clock on the SLAAC-1V board. Because of a limitation of the SLAAC-1V firmware, it takes four clock cycles per byte of configuration data to program the XCV1000 FPGA. A clock rate of 35 MHz and a configuration data size of 766 kilobytes yields an 87 ms configuration time. This is consistent with empirical results. Increasing the user clock rate will decrease the configuration time. The application used to obtain results does not operate correctly above 35 MHz. If the limitation in the SLAAC-1V firmware was removed, it would take two clock cycles per byte

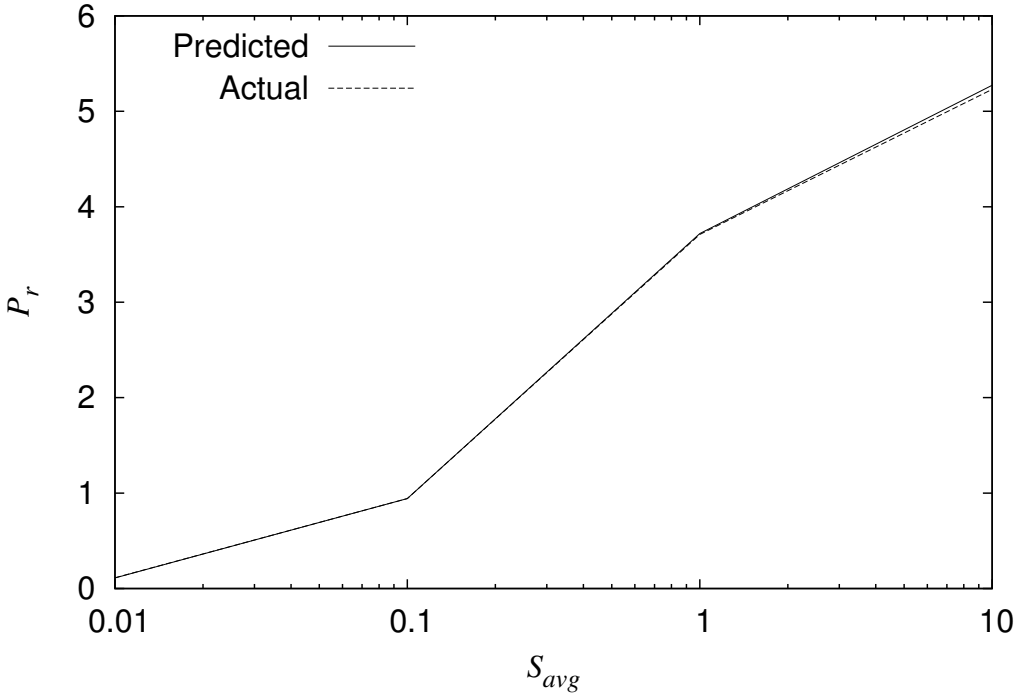

Figure 6.4: Throughput v S_{avg} for ACS API with caching

of configuration data to program an FPGA.

For the SLAAC-API, measured throughput without caching is 2.00 MB/s, and measured throughput with onboard caching is 3.71 MB/s. This is a speedup of 85 percent. The speedup benefit increases with decreasing packet size. For example, the calculated speedup for packet size = 1 kilobyte is 262 percent. As packet size increases the benefits of caching are diminished. This conclusion applies to the ACS API results as well.

One approach to improving cache performance is reducing the cache miss penalty [22]. The cache miss penalty, $T_{penalty}$, is equal to the reconfiguration time a cache miss minus the reconfiguration time of a cache hit. From Table 6.1, $T_{penalty} = 57$ ms for the SLAAC API. This implies that the memory transfer to load the cache slot takes 57 ms to complete. The only way to lower $T_{penalty}$, and thereby increase throughput, is to decrease the amount of

configuration data that must be transferred to the board. Because the configuration data for the processing elements on the SLAAC-1V has fixed length, a method of compressing the data is desirable. A number of compression algorithms for Virtex FPGA configuration data are presented in [23]. Compression ratios of less than 50% are exhibited. Decreasing the amount of data tranferred to the board on a cache miss decreases $T_{penalty}$.

Another approach to improving cache performance is reducing the cache miss rate [22]. The obvious way of reducing cache misses is to increase the size of the cache. Memory resources on the SLAAC-1V board are fixed. Therefore, a method of storing more data in the existing space is desirable. Currently only five configurations can be cached, one per onboard memory. Using the configuration compression algorithms found in [23], a configuration can be compressed by at least a factor of two. This reduces a 766 kilobyte configuration to 383 kilobytes. Each memory is one megabyte, so two 383 kilobyte configurations will fit in each memory. This doubles the number of cache slots from five to ten, thereby increasing the size of the cache. For many configurations a 40% compression ratio is achievable[23]. This ratio yields a compressed size of 306 kilobytes for a 766 kilobyte configuration. Three compressed configurations could be stored in a memory, thereby tripling the size of the cache to fifteen slots.

6.4 Other Platforms

The caching techniques analyzed in this document can be applied to other reconfigurable computing platforms. In particular, caching configurations at the network level improves the performance of all platforms.

Caching has been implemented on the RCM[12] board in [16] and [24]. Configuration data for the RCM is 9.4 kilobytes in length. Loading a context from the host takes 92ms. Loading a context from cache takes 454μs. The transfer time over a 100Mb network is at least 50μs.

The Osiris[7] board from USC-ISI has an architecture similar to that of the SLAAC-1V board. In addition to SRAM, the Osiris contains 512 MB of Dynamic RAM (DRAM). Instead of three Xilinx XVC1000 chips, it uses a single Xilinx XC2V6000 FPGA to provide reconfigurable computing resources. Configurations for the XC2V6000 are about 2.61 megabytes in length. If the DRAM is used as a configuration cache, then 196 configurations could be cached on the board. Few designs require that many configurations. The minimum theoretical time to transfer configuration data across the PCI bus is 21ms, therefore configuring from cache will be at least 21ms faster than configuring from the host. Transferring 2.61 megabytes of data across a 100Mb network takes at least 208ms. This very large delay is minimized if network level caching is employed.

The Xilinx ML300 Virtex-II Pro Development System contains a Xilinx XC2VP7 FPGA. An interesting feature of this reconfigurable computing platform is that the FPGA has an integrated general purpose processor, an IBM PowerPC 405. Configurations are 575 kilobytes in length. This results in a minimum network transfer time of 46ms. The ML300 contains 128 megabytes of SDRAM memory[25]. Up to 227 configurations could be cached onboard in this memory. The onchip general purpose processor controls reconfiguration, enabling more sophisticated caching strategies than are possible with an FSM. This board is not hosted in a computer, so all configurations are sent via a network.

The DINI DN3000K10 board[10] contains five Xilinx X2CV4000 FPGAs. Modifications being made by the Virginia Tech CCM lab will enable configuration of four of the FPGAs from onboard memory. The board contains one gigabyte of SDRAM memory. Configurations are approximately 1.87 megabytes in length. This means up to a whopping 548 configurations could be cached in onboard memory. Transfer of 1.87 megabytes across a 64-bit, 33 MHz PCI bus takes at least 8ms. Transferring 1.87 megabytes across a 100Mb network takes at least 150ms. Therefore, a cache hit can eliminate at least 158ms of configuration time.

FPGAs continue to grow in size, and therefore require ever more configuration data. The ability to cache configurations to avoid network and bus transfers becomes more useful as

configuration size grows. Eliminating a network transfer for the ML300 board cuts 46ms from configuration time, while eliminating a network transfer for the Osiris board with its larger configuration size cuts 208ms from configuration time.

6.5 Multi-Level Caching

Figure 6.5 shows the different levels of caching. Configuration time is lower for the higher levels in the figure. Capacity is higher for lower levels in the figure. The lowest level is mass storage, generally a magnetic or optical disk. Access to configurations is slow, but many can be stored. The next level is remote host memory. Configurations must be transferred across a network if found in this level. Local host memory is the level used by the ACS API to cache configurations at the network level. Each context in a multiple context architecture can be considered a cache slot.

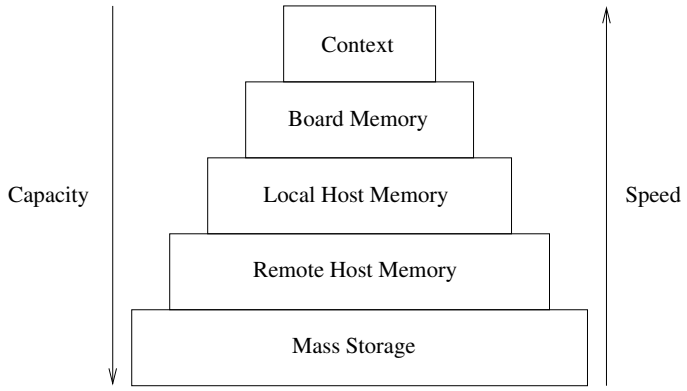

Figure 6.5: Cache Hierarchy Diagram

The higher in the cache hierarchy a configuration resides, the faster it will load. For boards with SDRAM, the board memory cache level is usually large enough to hold all configurations. The board memory cache level is also the highest cache level on all boards but the RCM[12] board. If all configurations are preloaded into the board memory cache, then $T_{avg} = T_{hit}$ because $R_{miss} = 0$ from Equation 6.2. This is the best performance available for

the single context architecture.

Relative to switching contexts, loading a configuration from board memory is slow. Figure 6.6 shows the performance of a multiple context system versus average packet size, where $R_{reconfig} = 1$ and $R_{miss} = 0$. It is evident from the graph that for $S_{avg} > 100$ there is essentially no performance penalty associated with switching contexts. It is expected that in practice R_{miss} will not be zero. Figure 6.7 shows expected performance for different rates of context cache misses versus average packet size, where R_{miss} is the rate of context cache miss and the $T_{penalty}$ refers to the time difference between configuring from onboard memory and switching contexts. It is evident from the graph that a ten percent context cache miss rate dramatically reduces performance. It is also evident that a fifty percent cache miss rate exhibits significant performance gains over having no context switching. This implies that context switching is a viable method of improving system performance by reducing the average configuration time.

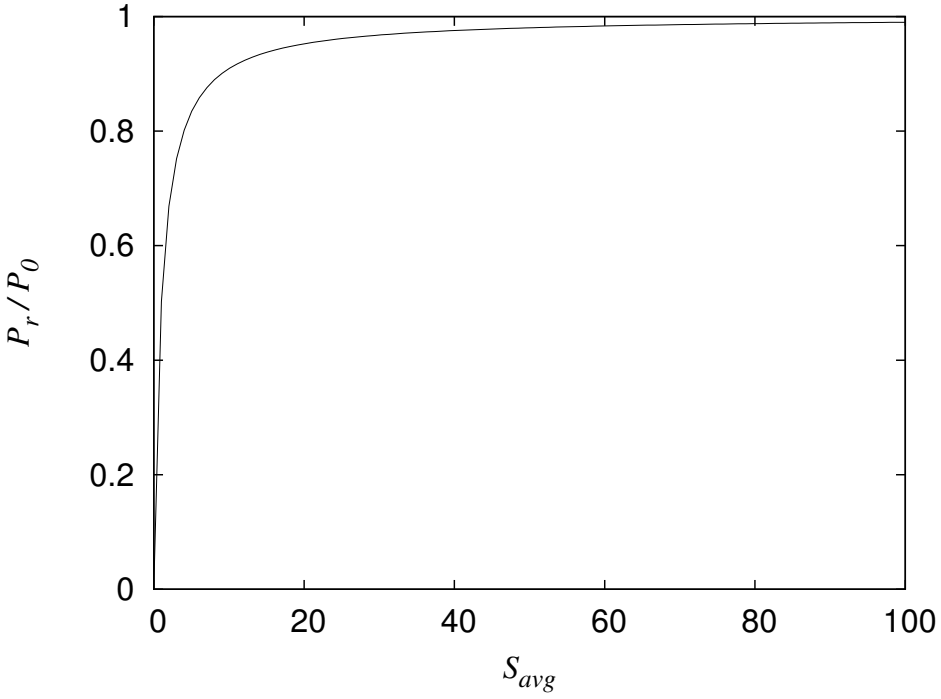

Figure 6.6: Performance versus S_{avg}

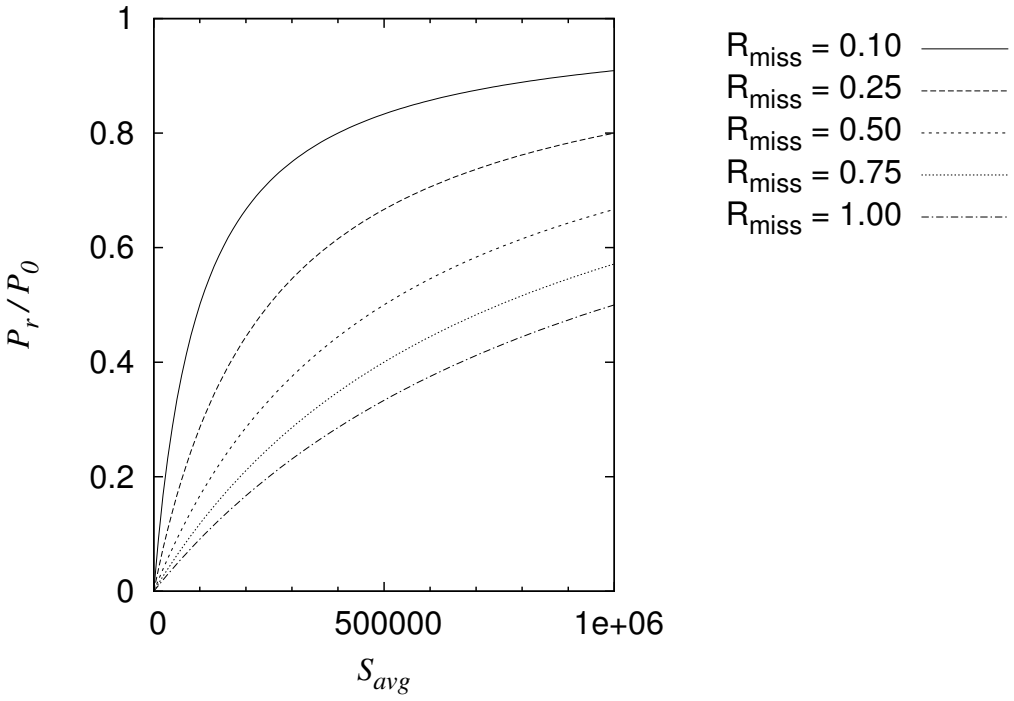

Figure 6.7: Performance versus S_{avg} with respect to R_{miss}

Chapter 7

Conclusions

7.1 Summary

This thesis presents a method of reducing configuration times for reconfigurable computing platforms. The benefits of decreasing configuration times are shown using a runtime reconfigurable streaming application.

A hardware architecture to cache configurations at the board level was implemented and benefits of the architecture were shown. The reduced average configuration time caused by the caching architecture resulted in improved performance of runtime reconfigurable applications.

Modifications made to the ACS API implemented node level caching for the SLAAC-1V board. This was shown to decrease the average reconfiguration time. It was shown that this improved the performance of runtime reconfigurable computing applications.

In summary, it was shown that decreasing average configuration time resulted in improved performance of runtime reconfigurable applications. Caching at the board level was shown to be effective method of reducing average reconfiguration time. Caching at the node level

was shown to decrease the detrimental effects of sending configuration data across a network.

7.2 Future Work

The application developed in this thesis is an example of *host-driven* RTR. This means that the host is controlling the active configuration of the board. An alternative is *data-driven* RTR, where the board controls the active configuration according to the data being processed. The presented caching architecture is capable of being used in a data-driven RTR system. Instead of having the host drive the FSM inputs, the inputs could be driven by logic on the board. Additionally, it is necessary to implement logic that allows the board to request a configuration from the host. The host would be responsible for loading requested configurations and data transfer. This approach could exhibit higher performance and more flexibility.

The finer granualarity of caching RFUOPs proposed in [17] could be combined with the configuration caching techniques used in this thesis. When caching RFUOPs, performance is penalized with every FPGA reconfiguration. Decreasing the reconfiguration time decreases the penalty. The caching technique analyzed in this thesis effectively decreases reconfiguration time, and therefore would improve the performance of systems using RFUOPs.

An exciting possibility is to take advantage of the ability of the XCV1000 FPGA to be partially reconfigured. Partial configurations can be loaded into one part of an FPGA while the rest of the chip continues to process data. Partial configurations are a fraction of the size of a whole chip configuration, leading to both decreased configuration time and an increase in the number of cache slots. Due to the fact that partial configurations may have different sizes, the onboard logic that manages the cache will increase in complexity.

Bibliography

[1] Kuan Yao, "Implementing an Application Programming Interface for Distributed Adaptive Computing Systems," M.S. thesis, Virginia Polytechnic Institute and State University, May 2000.

[2] G. Estrin, "Organization of Computer Systems - The Fixed Plus Variable Structure Computer," in *Proceedings of the Western Joint Computer Conference*, 1960, pp. 33–40.

[3] J. M. Arnold, D. A. Buell, and E. G. Davis, "Splash 2," in *Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, June 1992, pp. 316–324.

[4] Peter M. Athanas, "A Functional Reconfigurable Architecture and Compiler for Adaptive Computing," in *Twelfth Annual International Phoenix Conference on Computers and Communications*, March 1993, pp. 49–55.

[5] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.

[6] Scott Hauck, Thomas W. Fry, Matthew M. Hosler, and Jeffrey P. Kao, "The Chimaera Reconfigurable Functional Unit," in *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines*, April 1997, pp. 81–96.

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

[8] Pawel Chodowiec, Kris Gaj, Peter Bellows, and Brian Schott, "Experimental Testing of the Gigabit IPSec-Compliant Implementations of Rijndael and Triple DES Using SLAAC-1V FPGA Accelerator Board," in *Information Security, 4th International Conference, ISC 2001, Malaga, Spain, October 1-3, 2001, Proceedings*. 2001, vol. 2200 of *Lecture Notes in Computer Science*, pp. 220–234, Springer.

[9] Annapolis Micro Systems, Inc., *Wildforce Reference Manual*, Annapolis, MD, 1997.

[10] "The dini group website - dn3000k10 technical specs," http://www.dinigroup.com/products/3000k10ns.html.

[11] X.-P. Ling and H. Amano, "WASMII: a Data Driven Computer on a Virtual Hardware," in *IEEE Workshop on FPGAs for Custom Computing Machines*, Duncan A. Buell and Kenneth L. Pocek, Eds., Napa, CA, April 1993, pp. 33–42, IEEE Computer Society Press.

[12] Stephen M. Scalera and Jóse R. Vázquez, "The design and implementation of a context switching FPGA," in *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines*, April 1998, pp. 78–85.

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

[14] R. Bittner, M. Musgrove, and P. Athanas, "Colt: An Experiment in Wormhole Run-Time Reconfiguration," in *High-Speed Computing, Digital Signal Processing, and Filtering Using Reconfigurable Logic*. SPIE, November 1996, pp. 187–194.

[15] Yingchun He, "VLSI Implementation of a Run-time Configurable Computing Integrated Circuit - The Stallion Chip," M.S. thesis, Virginia Polytechnic Institute and State University, July 1998.

[16] Kiran Puttegowda, David I. Lehn, Jae H. Park, Peter Athanas, and Mark Jones, "Context Switching in a Run-Time Reconfigurable System," *The Journal of Supercomputing*, vol. 26 (3), pp. 239–257, November 2003.

[17] Scott Hauck, Zhiyuan Li, and Katherine Compton, "Configuration Caching Management Techniques for Reconfigurable Computing," in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2000, pp. 22–36.

[18] Marc Snir, Steve Otto, Steven Huss-Lederman, and David Walker end Jack Dongarra, *MPI: The Complete Reference*, The MIT Press, 1997.

[19] Scott J. Harper, *A Secure Adaptive Network Processor*, Ph.D. thesis, Virginia Polytechnic Institute and State University, April 2003.

[20] Prasanna Sundararajan and Steven A. Guccione, "XVPI: A Portable Hardware/Software Interface for Virtex," in *Reconfigurable Technology: FPGAs for Computing and Applications II, Proc. SPIE 4212*, John Schewel, Ed., Bellingham, WA, November 2000, pp. 90–95, SPIE - The International Society for Optical Engineering.

[21] Deutsches Museum, "Enigma Encryption Machine," http://www.deutsches-museum-bonn.de/ausstellungen/meisterwerke/2_3enigma/enigma_e.html.

[22] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach, 3rd Edition*, Morgan Kaufmann Publishers, Inc., 2003.

[23] Zhiyuan Li and Scott Hauck, "Configuration Compression for Virtex FPGAs," 2001, http://www.ee.washington.edu/faculty/hauck/publications/VirtexCompressJ.pdf.

[24] David I. Lehn, "Framework for a Context-Switching Run-Time Reconfigurable System," M.S. thesis, Virginia Polytechnic Institute and State University, Blacksburg, VA USA, May 2002.

[25] "ML300 User Guide," http://www.xilinx.com/products/boards/ml300/docs/ml300_ug.pdf.

Vita

James H. Hendry

James Hugh Hendry was born in South Kingstown, Rhode Island on November 14, 1975, and was raised in Catonsville, Maryland. After graduating from Cardinal Gibbons School in 1993, he entered the Engineering program at Virginia Tech. He graduated with a Bachelor of Science in Computer Engineering with a Minor in Computer Science in spring 1997. At that time he took a position with Acterna in Salem, Virginia. In fall of 1998 he took a position with SynaptiCAD, Inc. in Blacksburg, Virginia. He entered the graduate program of the Bradley Department of Electrical and Computer Engineering at Virginia Tech in fall 1999. While conducting research on reconfigurable computing systems, he took a part-time instructor position with his department in fall 2002.