UNIVERSITY OF CALIFORNIA

Los Angeles

Accelerating Applications through Cross-Layer Co-design

A dissertation submitted in partial satisfaction of the

requirements for the degree Doctor of Philosophy

in Electrical Engineering

by

Herwin Chan

2007

This dissertation of Herwin Chan is approved.

Eli Yablonovitch

William Kaiser

Glenn Reimann

Richard Wesel, Committee Co-Chair

Ingrid Verbauwhede, Committee Co-Chair

University of California, Los Angeles

2007

Table of Contents

List of Figures

List of Tables

Acknowledgments

I look back and am amazed at how I got to this point. It would have been impossible without the help of so many people. I would especially like to thank:

Ingrid Verbauwhede

Richard Wesel

Patrick Schaumont

... for their support and guidance

The members of

EMSEC (embedded security group)

CSL (communication systems laboratory)

... for their camaraderie and collaboration

My family

... for their emotional support

Kelly Tierney

... for the sunshine

VITA

1975	Born, Vancouver, Canada
1993-1999	B.A.Sc. in Electrical Engineering (Computer Option) University of British Columbia Vancouver, Canada
1999-2002	IMEC, vzw Leuven, Belgium Telecommunications Engineer
2002-2004	M.S. in Electrical Engineering University of California, Los Angeles Los Angeles, California
2002-2007	Graduate Student Researcher Electrical Engineering Department University of California, Los Angeles Los Angeles, California

PUBLICATIONS AND PRESENTATIONS

Y.K. Lee, H. Chan and I. Verbauwhede , “Design Methodology for Throughput Optimum Architectures of Hash Algorithms of the MD4-class ," Journal of VLSI Signal Processing Systems, (accepted).

Y.K. Lee, H. Chan and I. Verbauwhede, "Iteration Bound Analysis and Throughput Optimum Architecture of SHA-256 (384,512) for Hardware Implementations ," The 8th International Workshop in Information Security Applications, August 17-19, 2007.

H. Chan, M. Griot, A. Vila Casado, R. Wesel, and I. Verbauwhede, "High Speed Channel Coding Architectures for the Uncoordinated OR Channel," IEEE 17th INTERNATIONAL CONFERENCE ON Application-specific Systems, Architectures and Processors (ASAP), Steamboat Springs, Colorado, September 2006.

Y.K. Lee, H. Chan and I. Verbauwhede, "Throughput Optimized SHA-1 Architecture Using Unfolding Transformation," IEEE 17th INTERNATIONAL CONFERENCE ON Application-specific Systems, Architectures and Processors (ASAP), pp.354-359, Steamboat Springs, Colorado, September 2006.

M. Griot, A. Vila Casado, W.Y.Weng, H. Chan, J. Basak, R. Wesel, "Trellis Codes with Low Ones Density for the OR Multiple Access Channel," IEEE International Symposium on Information Theory (ISIT06), pp.1817-1821, July 2006.

H. Chan, P. Schaumont, and I. Verbauwhede, "Process Isolation for Reconfigurable Hardware," 2006 International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA06) (Distinguished Paper), pp. 164-170, June 2006.

M. Griot, A. Vila Casado, W.Y. Weng, H. Chan, J. Basak, E. Yablanovitch, I. Verbauwhede, B. Jalali, R. Wesel, "Interleaver-Division Multiple Access on the OR Channel," Information Theory and Applications Workshop (Invited Paper), San Diego, February 2006.

H. Chan, P. Schaumont, and I. Verbauwhede, "A Secure Multithreaded Coprocessor Interface," 3rd Workshop on Optimizations for DSP and Embedded Systems, March 2005.

H. Chan, A. Hodjat, J. Shi, R. Wesel, and I. Verbauwhede, "Streaming encryption for a secure wavelength and time domain hopped optical network," IEEE International Conference on Information Technology (ITCC 2004), pp. 578-582, April 2004.

H. Chan, A. Hodjat, J. Shi, R. Wesel and I. Verbauwhede, "Streaming Encryption for a Secure Wavelength and Time Domain Hopped Optical Network", *Embedded Cryptographic Hardware* (N. Nedjah and L. Mourelle, editors). Nova Science Publishers, Chapter 14, pp. 241-251, 2004.

ABSTRACT OF THE DISSERTATION

Accelerating Applications through Cross-Layer Co-design

By

Herwin Chan

Doctor of Philosophy in Electrical Engineering

University of California, Los Angeles, 2007

Professor Richard Wesel, Co-Chair

Professor Ingrid Verbauwhede, Co-Chair

In embedded system design, an application is usually broken up into independent blocks so that their implementation can be performed in parallel. The definition of the interfaces between these blocks ensure that they can be recombined together to produce a functional system. However, the presence of many interfaces incurs significant overhead that greatly reduces the performance of a system.

In this dissertation, a cross-layer co-design methodology is presented which produces high performance embedded systems by concentrating on the reduction of interface overheads. The main steps in the process include removal of intermediate interfaces, optimizing the application algorithm to minimize the use of interfaces, and finally, accelerating the performance of the interfaces themselves.

In addition to the research contribution of a design methodology, implementations of a wide range of embedded systems which vary in both size and application domains in presented. These implementations illustrate both the validity of the methodology and demonstrate how the process can be applied to systems that span physical domains, hardware/software domains, and security domains.

Chapter 1

Introduction

In embedded systems, the performance of a single application or function is often a critical factor to its success. The traditional design process, however, does not reflect the importance of this design goal. It first focuses on the implementation of a functionally correct system, then through profiling accelerates the application through acceleration of its component blocks.

To quickly implement an algorithm, the design is broken up into several different implementation blocks which allow different designers to work on them independently. The interface ensures that the different structures are able to interact well with each other and produce a functional system. Blocks on the same abstraction layer use the interface to synchronize parallel computations. Interfaces between different levels serve to abstract lower level structures so that they can be easily used by higher level blocks. These compositional properties are very useful in the building of functional systems; however, the interface between each layer of abstraction introduces significant performance overheads.

Indeed all the different layers of abstraction and different interfaces are very important in general computing systems that must be highly adaptable and perform many different functions. In a processor based embedded system, the operating system provides application software a system call interface to common system services such as reading and writing to files and outputting data to the screen. Drivers allow software to access peripherals without having to worry about the details of the underlying signaling protocol. In communication systems, the media access control (MAC) layer abstracts away the details of securing a communication channel so that a higher layer block can transmit its message. However, for embedded systems, the significant interface overheads between blocks can lead to expensive products or the sacrifice of important features.

The overhead of an interface is taken up by two main tasks: the packaging and sending of data and the synchronization between the two communicating components. In a software system running on UNIX, sockets are commonly used for two processes to communicate with each other; the synchronization time is taken up by the setup of a socket connection and communication time is taken up by the time it takes for data to travel through the established connection. In hardware, synchronization costs are often in the form of a handshaking protocol and communication speed is limited by the bit-width of the data ports. In the networking context, synchronization time includes the time it takes to gain access to the communication channel. In a TDMA system, this is determined by the time slot allocation algorithm. In an Ethernet system using the CSMA/CD protocol, this is a function of the current network load.

In this dissertation, a (holistic) cross-layer design methodology is presented which produces high performance embedded systems by concentrating on the reduction of interface overheads. In contrast to methods that reduce the time taken in the processing of a specific interface, the methodology presented also takes into consideration the removal of intermediate interfaces and the optimization of algorithms to minimize the use of the remaining interfaces.

This strategy is shown to be applicable to a wide range of embedded systems which vary in size and application domains. In addition to interfaces between different abstraction layers, this dissertation focuses on interfaces between different heterogeneous technologies, components and information types.

1.1 Examples of Interface Overhead

Interface inefficiencies account for a significant percentage of time spent on a calculation even in embedded systems. In [1], the authors explored performance optimization of the AES function in the Java cryptographic library. In this exercise, the Java API must be maintained in order to support legacy software which uses the library. Compared to the initial software only execution time of 198741 cycles, the hardware accelerated coprocessor version showed an order of magnitude improvement (19198 cycles). However, on further examination of the results, it was discovered that 18939 of those cycles were spent just passing data through the Java environment to the coprocessor.

In other words, performance improvements of *three orders of magnitude* can be achieved if interface overhead can be eliminated.

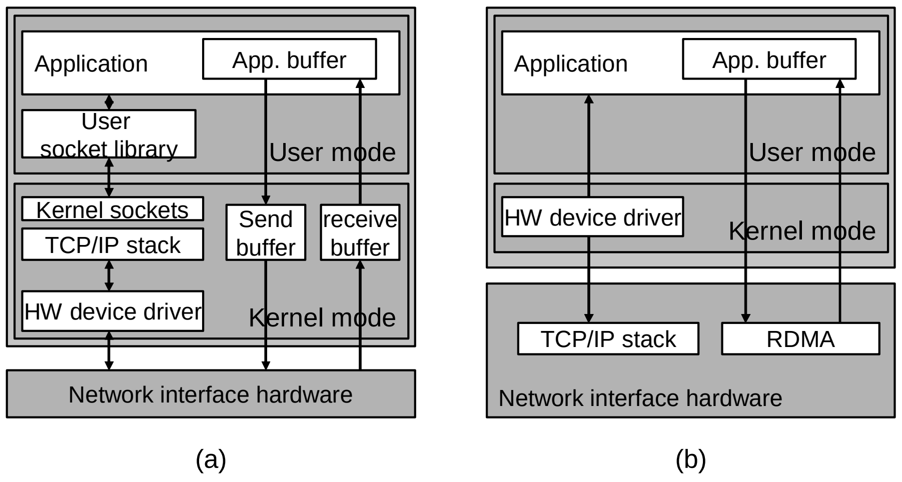

Figure 1-1 TCP protocol implementation (a) with traditional architecture (b) with optimized architecture

The implementation of TCP is an example of a widely used system where the majority of the time is lost on intermediate interfaces. In [2], a detailed performance profiling of the Linux-2.4 TCP stack is performed. The architecture of this implementation is shown in Figure 1-1(a). The results show that actual protocol processing only accounts for 10-15% of the actual execution time. Techniques such as zero-copy networking, checksum offloading and interrupt coalescing [3] are popular techniques and have been shown to double the performance of systems when used in combination.

These techniques focus on acceleration individual processing steps and do not remove intermediate interfaces which still make up the majority of execution times.

More than 50% of time is spent in the operating system performing interrupt handling, context switching, and system calls. More than 30% is spent on copying data from one memory location to another.

The profiling study implied that TCP performance can be improved by orders of magnitude if its processing can be consolidated into a single environment (such as specialized hardware) where the intermediate interfaces can be removed. Removing interfaces is difficult since the operating system is responsible for sorting and distribution of packets to the different applications; operating systems are full of internal interfaces. Remote direct memory access (RDMA) technology provided a solution to this problem in 2000 [4] by introducing a method which allows the sender to specify to the receiver the memory location to store the data. Thus TCP processing can be accelerated by 1) the removal of intermediate interfaces and 2) simplification of the application level interface through RDMA technology. The resulting implementation architecture is shown in Figure 1-1(b).

The process of TCP offloading echoes the main steps of the cross-layer design methodology proposed in this dissertation. First, the different parts of the algorithm are collected into a single implementation domain where intermediate interfaces are removed. The algorithm itself can then be optimized; optimization may even involve the design and implementation of a new algorithm that performs the same functionality more efficiently. Finally the application interface is simplified and optimized by separating the control and dataflow components.

1.2 Performance Components

To better understand how the algorithm can be optimized under the cross-layer co-design methodology, the time spend on all applications can be broken up into three generic performance components and analyzed. The three components are: control, computation and communications. Each of the design steps presented, focus on the reduction of the time spend on these components.

The control element determines the manner in which data is transferred between domains in a system. In a multitasking system, the portions of the operating system that manages the scheduling of tasks and performs context switches all reduce performance. In a software context switch, this is made up of the time it takes to save the context information of the current process and load the context information of the new processes. In the context of hardware, control overheads include the time spent performing handshaking between IP blocks, performing bus arbitration and synchronization. In a larger scale of networks such as TDMA and CDMA, control overheads include tasks such as clock synchronization, network coordination and media access control.

It is the nature of interfaces to allow data to flow between two different domains. However, each additional interface must inevitably require some performance overhead to manage this transfer of data. The removal of intermediate interfaces, therefore, also removes the control overheads due to each of the interfaces.

The computation element represents the series of operations that must be performed to produce the desired behavior in a system. The execution time of systems can be improved through modifications to the algorithms or implementation in different technologies. In data intensive applications, transformations of the algorithm for better data locality have been proven to vastly improve performance because unnecessary memory transfers are eliminated. In the context of interfaces, reducing intermediate interface boundaries exposes the algorithm to more of these optimization opportunities.

The communications element represents the time it takes to transfer data between different domains. When intermediate interfaces are removed, the communications between different domains are also removed. While this in itself can improve performance, there still exists at least one interface (to the user application) that cannot be removed. Performance can still be improved by optimizing the algorithm to require less communications or by acceleration of the communication mechanism.

1.3 The Cross-Layer Co-design Methodology

1.3.1 Minimize Interfaces

Once a task or application has been identified for acceleration, the first step is to minimize all the intermediate interfaces in the algorithm; the overhead due to interfaces are reduced if no interfaces exist. While the ideal case is for the application to be implemented without any interfaces, in most systems there exist some interfaces that are either impossible to remove or are very difficult to remove.

Interfaces that are impossible to remove include the physical interface, the application interface, and security interface. All these interfaces must exist since they are defined by the application itself. In an optical communications system, electronic data in the system must eventually be transferred in the optical domain. If a task or function is shared among several applications, then an interface must exist for the applications to access the accelerated services. In contrast, if data must not be shared among applications, a security interface must exist to prevent data from being transferred between two domains.

The interface between proprietary IP blocks in a system is a common example of an interface that is very difficult to remove. Companies depend on the implementation details of their design to survive and will not part with this information easily. An interface can also keep two complex blocks separated. Removing this interface can lead to a system that is too complex to analyze and optimize.

1.3.2 Optimize Algorithm

Once interfaces have been removed from the design, optimization to the algorithm itself is performed. Optimization techniques are very domain dependent. Classical software optimization strategies include transforming the algorithm to reduce the amount of executed instructions, memory access, and branch instructions. In the hardware domain, the algorithm can be accelerated by taking advantage of fast circuit technologies, and application specific architectures that take advantage of specialized functional units, pipelining, and parallel execution.

With a view towards reduction of interfaces, the optimization of an algorithm includes the possibility of a using a new algorithm that performs the same functions as the initial one but more efficiently. A solution obtained through this method, if available, is inherently easier to accelerate.

1.3.3 Optimize Interfaces

After the previous step, the remaining interfaces in the system are either impossible to remove or very difficult to remove. They can, however, still be accelerated to improve application performance.

The acceleration of the interface can be accomplished through implementation of new technologies. For example, DMA improves the interfaces between memory and a coprocessor in a SOC system. Widening the bus width of the system also improves interface performance by allowing larger amounts of data to be transferred each clock cycle.

A complimentary method of accelerating the interface processing is through the redefinition of the interface itself. For example, in an interface with a network protocol stack, header information is usually required by the interface for each packet to be sent. An improvement would be to only require header information to be given once per connection (which may span many packets of data).

1.4 Dissertation Overview

Chapter 2 describes the process used to design an optical network and illustrates how the methodology is used in large systems that span multiple physical domains. The development of an uncoordinated optical network was motivated by the observation that coordinated networks incur significant interface overheads in trying to maintain synchronization. An efficient implementation of such a system can be built by 1) removal of intermediate interfaces and 2) consolidating the channel code processing to a single domain. This strategy produced very low overheads between the physical domains and also enabled the development of fast and efficient channel codes in the hardware domain.

In the SOC context, hardware implementation of algorithms in coprocessors is a popular and easy way to improve performance by accelerating the computation element. Thus it is a good strategy to consolidate as much of the operation in hardware as possible to reduce intermediate interface overheads. Even with extensive consolidation, there remains, however, the final interface to the user. In Chapter 3, common processor - coprocessor interface options were studied to determine their relative merits. The overhead of the interface is from the combination of the control and communications elements. In Chapter 4, a methodology to separate these two elements and optimize them separately is presented.

Regular interfaces are designed to facilitate the efficient transfer of data between different domains. In contrast, security interfaces are designed to regulate and restrict the transfer of data. While these two types of interfaces represent conflicting purposes, Chapter 5 demonstrates how the design methodology can be used to build an efficient coprocessor that provides isolation services to the application processes that use it. The strategy of consolidating isolation services in hardware then providing a simple but restricted interface to user processes is also used to develop a new secure processor architecture in Chapter 6.

Chapter 2

Interface Between Physical Domains

Optical networks provide the backbone of a large percentage of the internet. Their main benefit is high throughput and low signal degradation. However, even with these attractive properties and the consumer's insatiable demand for more bandwidth and higher throughputs, the technology that makes up the internet is not used in local area networks (LAN).

The main reason for this is complexity and inefficiency of traditional optical networking technologies. WDMA and TDMA systems are able to transmit at near 100% optical capacities but require many resources to maintain coordination among all the nodes in the system. In TDMA, all nodes in the system must be synchronized to a base time and a central control distributes time slots to the nodes in the system.

Such a system incurs considerable costs; even before data is transmitted, a node must allocate resources to be connected and synchronized with the network. The overhead includes functions to synchronize all the nodes in the system as well as the implementation of the central control which monitors the network and arbitrates all network accesses. For the nodes in the network, overheads for transmitting a message include requesting of network resources so that messages can be exchanged, waiting for resource allocation and the partitioning of messages into multiple packets to fit time slot requirements.

Such complexities increase the interface inefficiency and have contributed to the unpopularity of optical networks in the local area network domain. Traditional Ethernet is the protocol of choice in this domain due to its simple interface and uncoordinated control scheme. Nodes on the network can be added and removed from the network dynamically without disruption of network communications. In addition, there is no explicit coordination between the nodes. When a certain node wants to transmit data, it merely starts to transmit and the protocol determines what to do when a collision occurs. The price that is paid for the simple interface design is that collisions start to dominate in highly loaded networks and bandwidth becomes highly degraded as a result [5][6].

It should be noted that though Ethernet technology has been adapted to be used in high speed and optical networks (standards are being developed up to 100Gbps), the uncoordinated feature has been removed along the way. The protocol has given up the CSMA/CD property which is responsible for detecting and dealing with collisions.

The task, therefore, is to develop an uncoordinated optical network scheme which is able to combine the easy interface properties of traditional Ethernet with the high bandwidth/throughput properties of optical networks in the LAN context. Such a system involves system co-design between three separate physical domains (the optical, electrical, and logical); the design of the two physical interfaces at the boundary of these domains must be carefully considered.

To build an efficient interface, major processing functions are first consolidated to a single domain. Because of the complexities of the algorithms required for uncoordinated access, processing was consolidated to the logical domain; thus, the electrical and optical systems and the interface between them can be kept simple.

In this chapter, the uncoordinated multiple access problem is explained in detail. To implement an efficient system, the protocol processing is consolidated in the logical domain. Once the problem is bound to a single domain, the algorithms were designed taking into consideration the design parameters of high optical throughput and architectural limitations. Validation of the methodology and design was performed by implementing the algorithms on FPGAs and interfacing it to an optical network.

2.1. Optical Networks

Optical communications have been becoming more and more important with the ever-increasing demands for bandwidth. Fiber optic technology has been demonstrated for speeds up to hundreds of gigabits per second because of its low loss and low dispersion over extended bandwidths. These properties of optical technology have been well utilized to form the backbone of global networks such as the internet and telephone networks. Communications among the optical transmission stations approach 100% efficiency (requiring no redundancy for error correction) by means of WDMA and

TDMA techniques. This has been possible at the cost of maintaining coordination between the different transmitting nodes.

In the local area network domain, optical networks have had limited success. Though the optical token ring (FDDI) network promises higher bandwidth [7], Ethernet networks are significantly more popular. This success is due to the ease in which a network can be set up. Nodes on the network can be added and removed from the network dynamically without disruption of network communications. In addition, there is no coordination between the nodes. When a certain node wants to transmit data, it merely starts to transmit and the protocol determines what to do when a collision occurs.

This need not be the case. In the CANbus network [8], collisions between data from several transmitters are used to determine the priority of the messages. By monitoring the aggregate signal of all the transmitters, a transmitter can determine whether there is a transmitter of higher priority. If this is the case, it would abort its own transmission. High priority transmissions have higher number of dominant bits (bits whose value cannot be over-written) in its header; if a dominant bit is detected by a transmitter that is transmitting a non-dominant bit, then it knows that it is colliding with a high priority transmission. Though CANbus only allows collisions in the header of a data transmission to determine priority, it illustrates that data can be transmitted in the OR channel even when collisions occur.

An OR channel is a channel that behaves like an N-input OR gate, where N is the number of nodes transmitting simultaneously. Assuming on-off keying, if any node transmits a '1' data bit, all the receivers will see a '1' bit in the channel. If all nodes transmit a '0' bit, then the receivers will see a '0' in the channel. In the CANbus network, the '1' bit is called the dominant bit since its value hides the presence of any '0' bits. A passive optical star network can also be used as an OR channel. Physically, the dominant '1' bit is represented by the presence of light and the '0' bit is presented by the absence of light.

Though there are efforts to implement Ethernet on optical networks, aggregate throughput performance is fundamentally limited by collision of data [9]. The desirable properties of Ethernet are demonstrated by marrying the high-bandwidth properties of optical networks with the flexibility of Ethernet. Collisions are avoided by careful design of channel codes. The uncoordinated multiple access properties will be provided by a set of novel channel codes, which guarantee that data can be decoded at optical bit error rates i.e. BER<10e-9. This bit error rate performance will be maintained even in the presence of other transmissions (interference). 30% efficiency was achieved by treating the interference users as noise. Theoretically, up to 70% efficiency can be achieved in such a channel. In [10] nonlinear turbo codes which provide a BER of 1e-7 at 60% efficiency has been proposed. However, these codes require block-lengths in the order of thousands, which increases latency, and require an iterative decoding is much more costly to implement and cannot achieve optical data rate using current technology.

2.1.1 Demonstrator Design

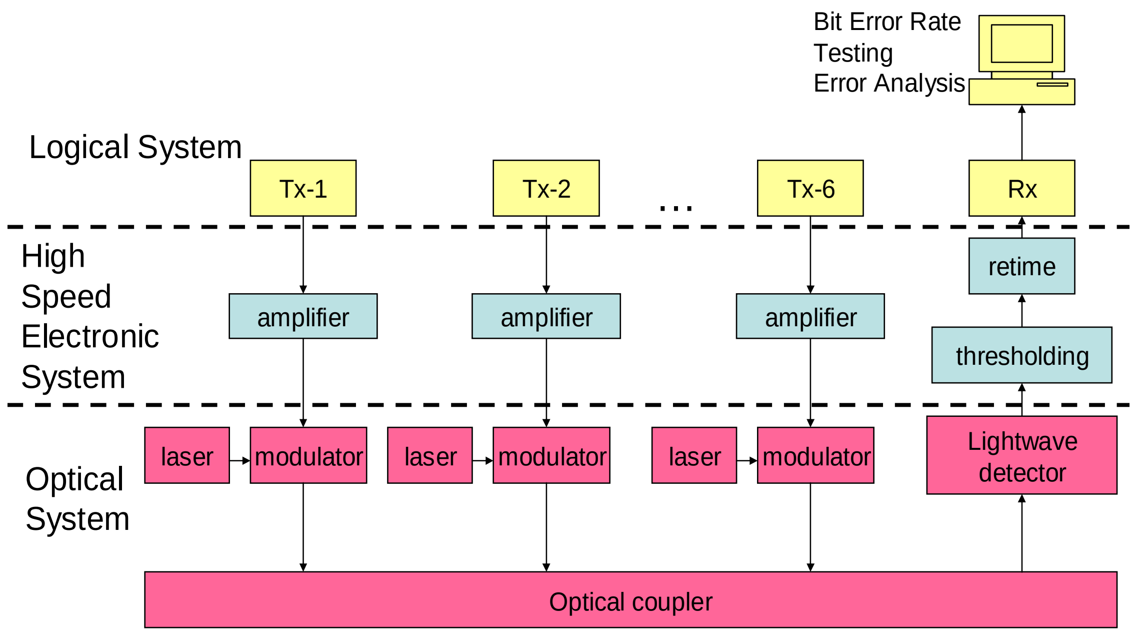

Figure 2-1 High level view of demonstration system

The process of building a high performance system through multiple technological domains is described in this chapter for the case of a novel optical network. The result is a working demonstrator that meets the system requirements and validates the design approach. To demonstrate uncoordinated multiple access optical networking, the system comprises of six nodes simultaneously transmitting into an optical channel. A receiver node takes the aggregate signal and decodes a single user. Figure 2-1 shows the system block diagram of the system. Each transmitting node is comprised of an FPGA, which codes the data; a laser, which provides the carrier for the coded data; and a modulator, which combines the two together and puts the data on the channel.

Successful transmission and decoding of data from six users on a single optically coupled network verifies the correct functionality of the design. The channel bit rate of 2 Gbps is divided among the 6 nodes using novel channel coding techniques. This will give an un-coded useful data bandwidth of 93 Mbps for each node. The channel bandwidth is guaranteed for each node; in Ethernet technologies, useful bandwidth experienced by each node are not guaranteed and depends on the traffic characteristics of the other nodes in the system.

Though the system was built with commercial off the shelf components, the design involves co-design between several design domains. The decision to implement the channel codes on a Virtex II Pro FPGA platform allows hardware speeds to be achieved while maintaining a programmable platform. In addition to this, this platform allows us to accurately predict the performance and cost of such a network in the future when ASICs are used. The choice of platform in the logical domain, simplifies the design of the interfaces in both the electrical and optical domains.

The Virtex II Pro FPGA board contains several high speed serial transceivers. They were independently measured to be able to support a channel rate of 2 Gbps. The electrical/optical system was built in order to support this rate.

The main design goal for channel coding is to find the highest rate code that is able to provide <10e-9 BER. Experience has shown to us that for complex FPGA designs, useful work take between 5ns to 10ns to compute; this translates to a 100 MHz to 200 MHz clock. In order to process the 2 Gbps, the coding algorithm will need to support a parallelization factor of at least 10 to 20. Also because of the large throughput requirements, powerful iterative coding algorithms may not be used.

2.2. Uncoordinated Multiple Access Code Design

In this section, the optical channel model used in this work and the high-level system design techniques used to provide uncoordinated multiple access to optical channels is briefly described. For further details on the theoretical aspects of the approach presented in this chapter, the reader should look at [11].

A simple communications model that can describe the multiple-user optical channel with non-coherent combining is the OR multiple access channel (OR-MAC). In this channel, if all users transmit a zero, then the channel output is a zero. However, if even one user transmits a one, then the channel output is a one.

Information theory tells us that the maximum sum-rate (the sum of the rates of all the transmitters in the system) of the OR channel is 1 information bit per received data bit. For uncoordinated multiple-access, Interleaver-Division Multiple-Access (IDMA) [12][13] is a promising approach which has been successfully applied to general MACs. With IDMA, every user has the same channel code, but each user's code bits are permuted using a unique randomly drawn interleaver. The receiver is assumed to know the interleaver of the desired users, and performs joint iterative decoding of all the users data. However, under current technology, this decoding technique produces prohibitively large designs for optical speeds with today’s technology.

Hence, for a simple uncoordinated access decoder, other users must be treated as noise. From a single-user perspective, this transforms the OR channel into the Z-Channel shown in Figure 2-2. In this channel, when a particular user transmits a 1, a 1 is received. When this user transmits a 0, a 1 can be received with probability equal to the probability that any of the users transmits a 1.

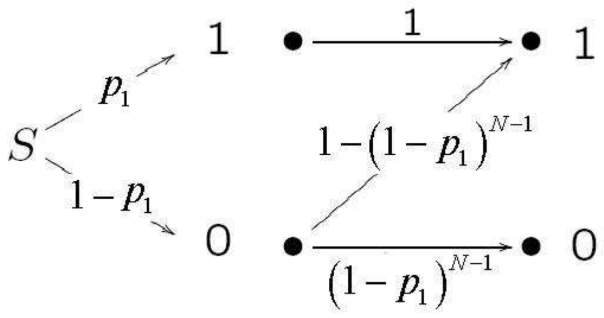

Figure 2-2 The Z-Channel

Treating other users as noise, a channel sum-rate of ln(2)~70% can theoretically be achieved for any number of users. Thus, while dramatically decreasing the decoding complexity, only 30% of the channel sum-rate would be lost with the use of capacity achieving codes.

For the IDMA-based architecture presented above, what is left is to design appropriate channel codes for the Z-Channel. In order to achieve the maximum symmetric sum-rate where each user sees a Z-channel, the channel code must produce in its output a particular average density of ones p_1 which depends on the number of users N as: $p_1^{opt} \approx \frac{\ln(2)}{N}$.

Linear codes produce an average ones density of 1/2, which would lead to an unacceptable sum-rate. For example, for 6 users the maximum achievable sum-rate using linear codes is less than 10%, and for 10 users it is less than 1%. Hence, non-linear codes that produce the proper ones density are required for this application.

The channel code used in this work is a Non-Linear Trellis Code (NL-TC) code. This novel code provides the appropriate information rate and density of ones. A Viterbi decoder allows a simple and fast decoding of NL-TC. A brief description of the design of these codes is presented in the following subsections.

2.2.1. Directional Hamming Distance

Regular convolutional codes are designed so that the Hamming distance between codewords is maximized. Hamming distance is the number of bits that differ between the codewords. This distance is directly associated with the number of errors such a code can decode. In the Z-Channel, a transmitted 1 will always induce a received 1. Thus, to make a decoding error, the decoder must see ones in all the bit positions where the incorrect codeword has ones. This implies that a new definition of distance is required. Let us define the **directional Hamming distance** $d_D(c_1,c_2)$ the number of positions at which the codeword c_1 has a 0 and the codeword c_2 has a 1. Note that $d_D(c_1,c_2)$ is not necessarily equal to $d_D(c_2,c_1)$.

Given that the purpose of the design is to maximize this directional distance, the safest definition of distance between branches would be

$$d_{i,j} = min[d_D(c_i,c_j), d_D(c_j,c_i)] ,$$

which is the ‘greedy’ branch-wise metric that will be maximized in the design. By taking the minimum between the two directional distances as the metric to maximize, we seek to maximize the minimum directional distance d_{min} between all codewords, albeit in a greedy fashion.

With this branch-wise metric, codewords with equal Hamming weights produce a larger d_{min} than codewords with different Hamming weights, so output values are assigned to the trellis branches with as similar Hamming weight as possible, preferably equal.

2.2.2. Non-Linear Trellis Code Design

A conventional feed-forward trellis encoder is used in order to determine the branches of the trellis, as shown in Figure 2-3. It is a rate-1/n, 2^v-state trellis code, with one input bit per trellis branch. However, instead of using generator polynomials to compute the output of each branch as is typically done, a non-linear table-lookup directly assigns the output values.

The trellis code design consists of assigning output values to the branches of the trellis code. Those outputs have to maintain the desired average density of ones p_1. The goal is to maximize the minimum directional distance d_{min} using the greedy pair wise metric.

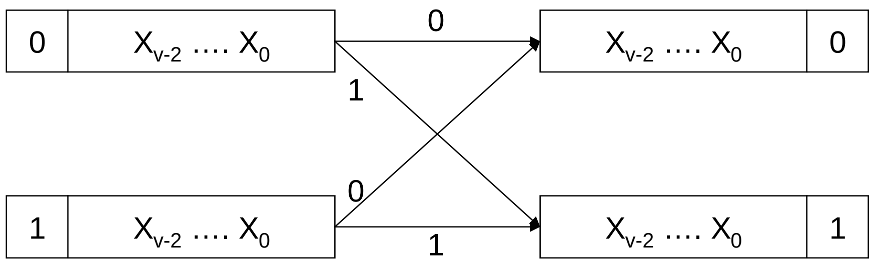

Figure 2-3 Basic sub-graph of the trellis diagram

The first step in the design is to assign the Hamming weight of each branch (number of ones), so that the optimal average ones density is satisfied as closely as possible. What is left is to assign the position of those ones in each of the outputs.

An extension of Ungerboeck's rules [10][11] in the context of the pair wise metric can be applied. Ungerboeck's rule [14] is based on the fact that every incorrect codeword, in its trellis representation, departs from the correct state (split) at some trellis section and returns to the correct state (merge) at least once. Ungerboeck's rule consists on maximizing the distance between branches splitting from a state (splits) and branches merging to a same state (merges).

One can extend Ungerboeck's rule more deeply into the trellis, and maximize not only the distance between splits, and the distance between merges, but the distance between the 4 branches coming from a split in the previous trellis section, or the 8 branches coming from a split two sections before, and so on (see Figure 2-4). One can do the same with the merges moving backwards in the trellis. Notice that by maximizing the distance between the 8 branches coming from a split two sections before, the distance between all 4 branches coming from a split a trellis section before and all splits are maximized as well. The same design strategy is used to maximize the distance between merges.

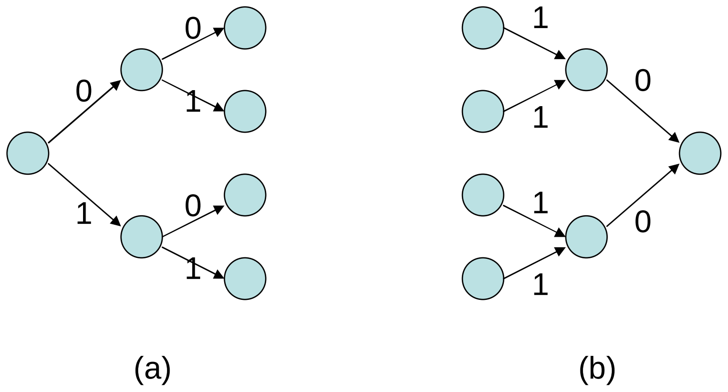

Figure 2-4 a) Four paths that start on the same state in two trellis sections. b) Four paths that arrive to the same state in two trellis sections. Branches are labeled with the input bits that induce traversal of the branch

Using the above design strategy, three candidate codes were design with coding rates of 1/17, 1/18, and 1/20. Figure 2-5 shows the candidate codes in a Matlab simulation of BER versus the number of simultaneous users in the system. The achieved BER is in the order 1e-5 which is considerably above the target BER 1e-9. However, this can be solved by using a Reed-Solomon code as an outer-code as will be explained in Section 2.2.3. The 1/20 code was chosen for the system for practical reasons. Though the 1/17 and 1/18 codes may achieve the required BER, the high speed serial transceiver has a 20 bit interface and is, therefore, easier to connect with a 1/20 code. The other codes require extra interface hardware to be built and may add to the complexity of the design and decrease throughput.

2.2.3. Block Code with NL-TC

Optical systems typically deliver a very low BER. In order to maintain this BER, the rate of the NL-TC channel code would have to be very low. A better solution is found taking into account the distribution of the erred bits in a transmitted stream after the NL-TC decoding. Thus, a high rate block code that can correct few symbol errors can be attached as an outer code, dramatically lowering the BER.

A concatenation of the rate-1/20 NL-TCM code with a (255 bytes, 237 bytes) Reed-Solomon code has been tested for the 6-user OR-MAC scenario. The rate of this code is $(237/255)\times(1/20) = 0.0465$. The simulated BER is 2.5e-10. For six users, the sum-rate is $6\times0.0465 = 0.279$.

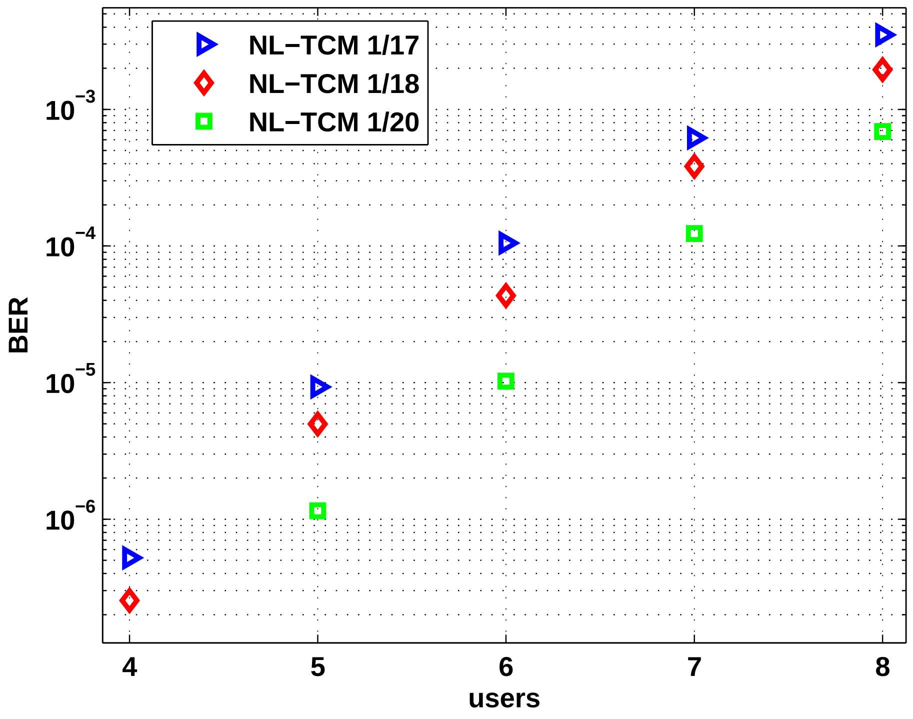

Figure 2-5 Bit error rate of NL-TC codes versus the number of users

2.3. Architectural Design

New and novel codes were designed to meet the high throughput specifications by the adaptation of simple non-iterative codes. It is, however, the implementation of these codes in hardware that ultimately determines the actual throughputs achievable by the system. The coding algorithms were implemented on the Xilinx Virtex II-Pro FPGA [15]. In particular, a rate-1/20 64-state NL-TC was implemented, intended for 6-user multiple access to the OR channel. The implementation dataflow block diagram is shown in Figure 2-6. Data to be transmitted is first encoded with a Reed Solomon block code. The output bits of this block are encoded with the trellis code and then passed on to the interleaver. Finally, the resulting bits are sent to the high speed serial transceiver (Rocket

I/O) to be sent off-chip. The chosen rate 1/20 trellis code together with IDMA provide the uncoordinated access properties of the system and are able to bring the bit error rate (BER) to about 1e-5 for six users. The outer Reed Solomon block code further decreases the BER to below 1e-9.

All the nodes of the system use the same code, but the IDMA interleavers are used to ensure that the coded bit patterns do not look the same in the optical channel. Finally, the Reed-Solomon block code is used to further reduce the BER to less than 1e-9. In addition to these blocks, synchronization blocks ensure that the received bits are aligned properly so that decoding can be performed correctly.

During code design, the interference signal was assumed to have a random

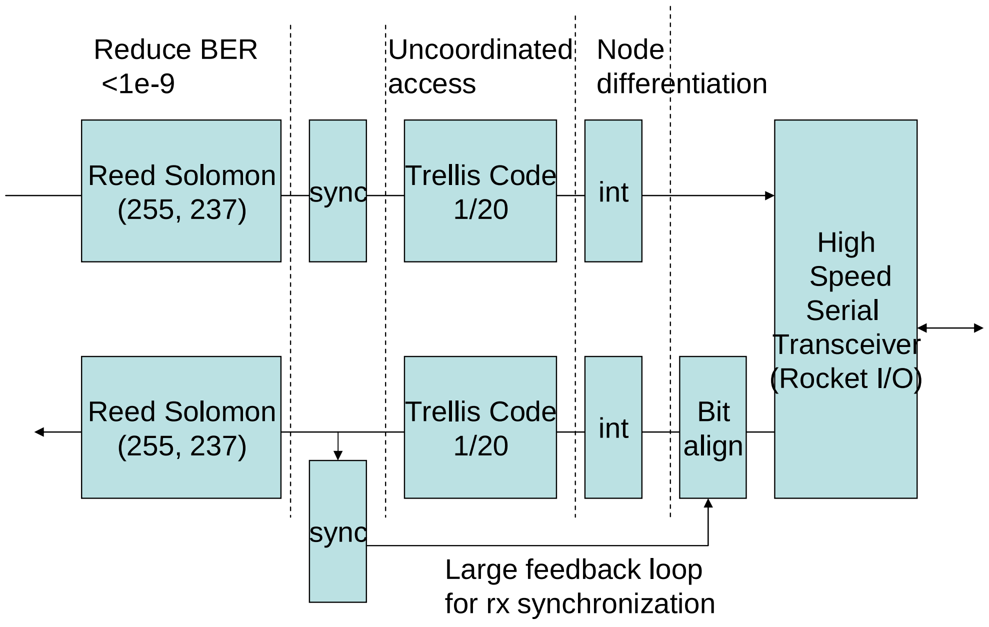

Figure 2-6 Coding implementation dataflow

uniform distribution; therefore interleavers are used after channel coding to randomize the position of the code bits. This combination allows us to recover data at a BER of 1e-5. A Reed Solomon block code is added at the back end to reduce the BER further to 1e-9. Since the target physical layer is the optical channel, data throughput is the main design criterion. The Viterbi decoder and interleaver blocks have been identified as the bottlenecks of the system and novel architectures are developed to mitigate their effects.

2.3.1. Trellis Encoder

To protect data in the OR channel, the NL-TCM code uses 20-bit codewords and contains 64 states. Figure 2-7 shows the architecture of the trellis encoder. The design of the encoder consists of a 5 bit shift register used to address memories that outputs two of the 128 possible codewords. The latest input bit is used to select the desired codeword. Each clock cycle a new data bit is shifted into the register and a new 20 bit codeword is produced. Unlike common binary encoders, the designed trellis code has a relatively low ones density, much less than the usual 50%.

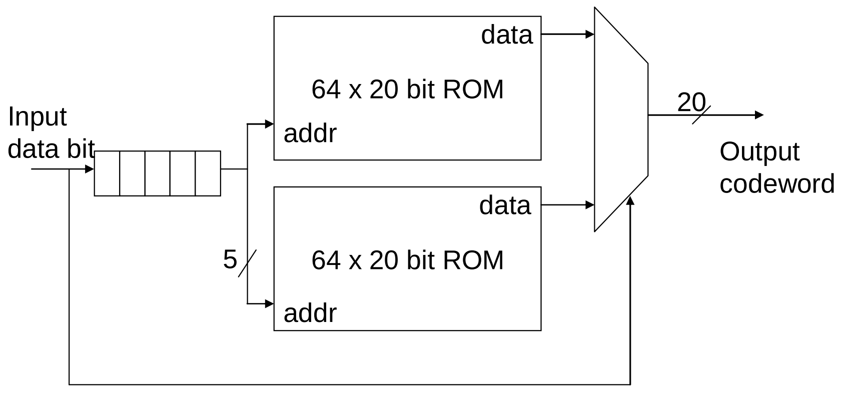

Figure 2-7 Trellis encoder architecture

2.3.2. Viterbi Decoder

In DSP implementations, the Viterbi decoder focused on the acceleration of a single branch metric calculation and careful memory management for storing the results. This means for decoding a single code word, several clock cycles (depending on the number of states in the trellis code) are needed. Hardware architectures such as those proposed by Zhu and Benaissa [16] and Guo et al. [17] have focused on area efficient architectures. For common wireless applications, such as 802.11b and 802.16a, Abdul Shakoor et al. [18] describe a fast parallel hardware implementation that decodes at 160Mbps on a FPGA.

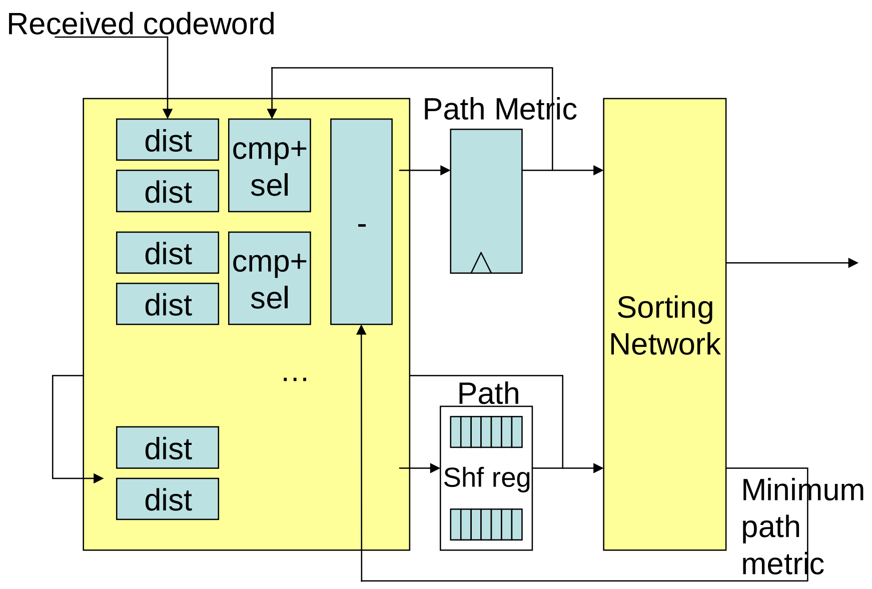

Figure 2-8 Viterbi decoder architecture

For the non-linear trellis code that uses 20-bit codewords and contains 64 states, a trace back length of 35 is used in the Viterbi decoder. The technique used to design the decoder is to parallelize and pipeline all operations as much as possible. Care was taken to find structures where feedback paths are as short as possible. The overall architecture of the Viterbi decoder is shown in Figure 2-8.

The Viterbi decoder can be divided into several different stages, each of these stages will be discussed individually in detail:

- calculation of metric
- accumulation and selection of metric
- finding of minimum path
- subtraction of accumulated result

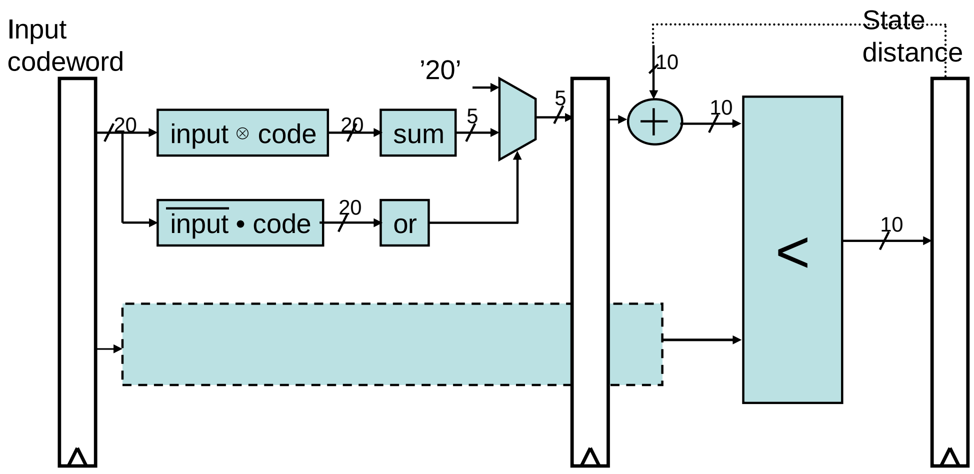

Figure 2-9 Calculation of path metric

Because the Viterbi decoder is being designed for the OR channel, the branch metric used is different from traditional designs. In an OR channel, it is impossible to receive a 0 bit when a 1 bit has been transmitted by any of the nodes. Because of this, in the comparison between the received codeword and branch codeword, if any of the received bits is 0 when a 1 is expected the branch metric is set to a maximum value of 20. Errors in which a 1 is received when a 0 is expected are summed together to give the branch metric in the normal case. The logic used to implement this function is shown in Figure 2-9. In the 64 state codes, 128 branch metrics are calculated in parallel; the logic used to calculate this function constitutes one stage in the decoder pipeline.

There are two possible branches that lead to each of the 64 state nodes. The path with the smallest path metric (which is an accumulation of past branch metrics) is chosen as the most likely path that was taken to reach the node. Path metric calculation is performed by adding the path metric of the source nodes to their respective branch metrics. The two sums are then compared and the path with the lowest metric is selected. Sixty-four of these calculations are performed in parallel and constitutes a single stage in the pipeline. Further pipelining of this stage is impossible since the calculation of the path metric involves a feedback path from previous path metric calculations. Figure 2-9 shows the implementation of this function.

The most likely bit that was transmitted is the bit at the head of the path with the lowest path metric. At each cycle, 64 path metrics are calculated and their respective paths are accumulated. A sorting network is used to select the path with the smallest accumulated metric. A minimum time sorting network based on Batcher’s odd-even merging algorithm [19] is used. This is a recursive algorithm that sorts a group of unordered numbers (Figure 2-10) and contains the following three steps:

- Divide the numbers in to two groups
- Sort the two groups of numbers separately
- odd-even merge the two groups of numbers.

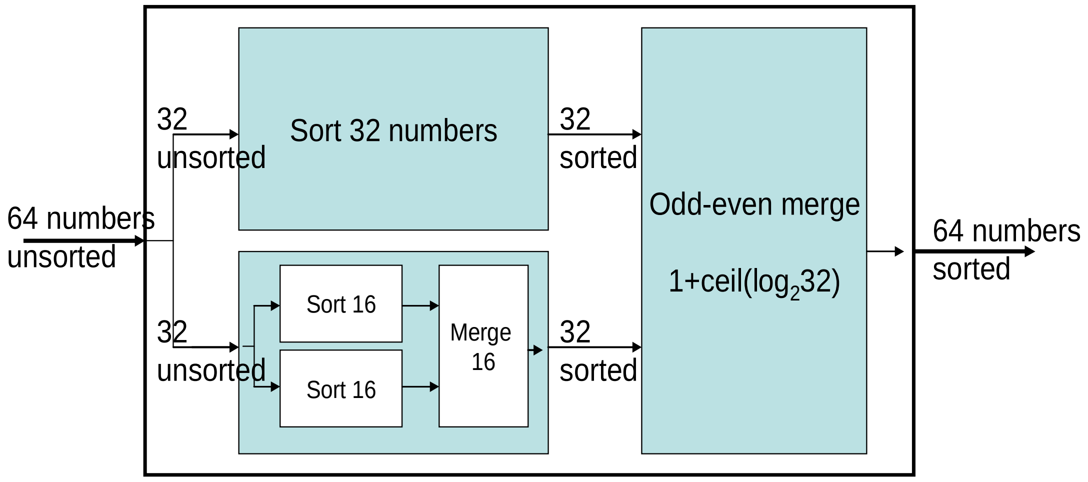

Figure 2-10 Block diagram of sorting network

Since it is a recursive algorithm, the basic operation is a sorting of two numbers. This is implemented with a two input comparator. The odd-even merge procedure which combines two sets of sorted numbers into a single set is also recursive and based on the use of two input comparators.

For the sorting of n numbers, the number of comparators grows in $O(n\log n^2)$. The delay through the network is $\begin{pmatrix} 1+\log n \\ 2 \end{pmatrix}$. For the designed system of 64 states, this translates to 543 comparators with a delay of 21 comparators. However, since there are no feedback paths in the sorting algorithm, the architecture can be fully pipelined to achieve very fast throughputs.

The minimum path metric is fed back to the Viterbi decoder and subtracted from all 64 accumulated path metrics. This is to ensure that the register values do not overflow. The sorting network used to find the minimum path is heavily pipelined, so the value used is several cycles behind the values that are currently calculated. This delay in the results translates to larger possible accumulated path metric values which may necessitate the use of larger operators (like adders); this increases the delay of the calculation. Therefore, care was taken to pipeline the sorting network only to the degree that is necessary to avoid unnecessary increases in hardware and possible increases in critical path delays. The sorting network in the design is pipelined to have 6 cycles of latency.

2.3.3. Interleaver

Interleavers, which permute the order of data bits, are commonly used to randomize the data stream and improve the performance of error correcting codes. However, in the designed system, each transmitter uses a unique interleaver pattern. This pattern is chosen from a set of patterns determined at design time to have good cross correlation properties. The role of the interleaver in the system is similar to its role in an IDMA system described by Ping et al. [13]. In that system, interleavers are used to distinguish nodes in a wireless CDMA system and increase channel capacity. The interleaver design, therefore, must be flexible enough to accommodate a family of permutation sequences that work well together. Interleaver design for IDMA has been examined by Pupeza et al. [20]; however, since the focus of that work has been on performance efficiency rather than high-speed implementation the results are not applicable to the design high data rates.

A de-interleaver is used at the receiver to recover the initial sequence. Its architecture is the same as the interleaver architecture; the permutation sequences, however, are run in reverse order to recover the original uninterleaved signal.

In theory, the ideal interleaver architecture is one that allows an input data block of size N to be permuted to any of its N! possible permutations. Conventional interleaver architectures process the data serially i.e. a single bit at a time. This scheme becomes increasingly difficult to implement as data rates increase e.g. a 10Mbps channel only allows 100ps to process each bit. The architecture design, therefore, focuses on parallel processing to achieve the desired rate. Care was taken, however, to ensure that the architecture can support enough permutations so that a good set of interleaver patterns can be found.

One possible method of implementing the interleaver is to consider the input as 20 bit words. The output of the interleaver will be a random ordering of the 20 bit words. The implementation of this interleaver is both fast and has low complexity. However, simulations show that this does not provide enough randomness for the channel codes.

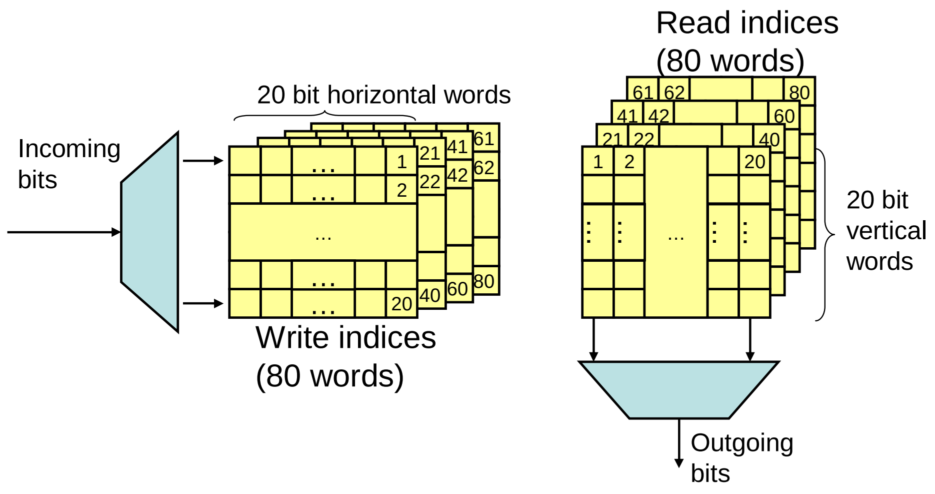

Figure 2-11 Indexed write-by-row, read-by-column interleaver

To increase randomness without sacrificing speed and complexity, a randomized write-by-row, read-by-column scheme for the 1600 bit interleaver was adopted. As seen in Figure 2-11, data can be broken into square blocks of 400 bits. Each of the 20 rows and columns are indexed. Groups of 20 incoming bits are written to a randomly indexed row. When the data block is filled, the bits are read out of the block one column at a time in a random order.

The 400-bit-square block forms the basic unit of the interleaver design. In order to produce the necessary randomness, four of such blocks were used in the final implementation. Like the indexing within the blocks, the inputs and outputs of the four blocks are accessed independently and randomly.

This scheme provides us with enough randomness to operate on the optical channel. In the interleaver design, 4 square blocks of 400 bits are used, giving us a total of 80 indexed locations. This corresponds to a design space of $(80!)^2 > 1e+237$ possible permutation sequences to choose from. For the desired channel rate of 2 Gbps, using the 20-bit word size of the trellis code, the target operating frequency for the interleaver is 100MHz.

Two such memory blocks are used to allow the desired throughput to be maintained. While the first block is being written to, the second block is being read out. When the memory block is filled/emptied, the function of the memory blocks is reversed. This ping-pong arrangement doubles the area of the interleaver.

2.3.4. Reed Solomon Code

When the Trellis decoder block makes an error, the errors usually come in a burst of a few bits at a time. A Reed Solomon (RS) code is a block code that operates on bytes at a time. This makes it a very good choice to correct the residual errors and bring the final BER to below 1e-9. A standard (255,237) RS code was selected.

Since timing is not critical in this block, a standard open source architecture design from Han [21] was used. The syndromes of the input data block are first calculated. The results are then used to calculate the error locator polynomial using Berlekamp's algorithm. The Chien algorithm is used to find the roots of the error locator polynomial and these roots provide the location of the errors. Finally, the magnitudes of the errors are captured.

The data rate at the output of the NL-TCM decoder is 100Mbps. Since the Reed Solomon code operates on data blocks of 255 bytes (2040 bits), the time budget for the

RS decoder is 20.4us. The module runs on a 50MHz clock, and at the worse case the decoding operation takes 856 cycles (17.1us) to complete.

2.3.5. Implementation Results

The system blocks were implemented on the Virtex II-Pro FPGA from Xilinx. Table 2-1 summarizes the size various blocks in the design. The critical period is given for the transmitter and receiver.

Table 2-1 Size and speed of transmitter and receiver blocks

	Area (slices)	Critical period (ns)
Transmitter		
Reed Solomon encode	189	5.3
NL-TCM encode	34	3.4
Interleaver	3387	7.7
Receiver		
Reed Solomon decode	3686	9.0
NL-TCM decode (Viterbi)	10504	10.3
Interleaver	3387	7.7

The transmitter is implemented on the Virtex II Pro XC2VP20 FPGA which contains 9,230 slices of logic. Each transmitter design occupies 40% of the available area. The receiver is a significantly larger design and is implemented on the XC2VP50 which has a capacity of 23,616 slices. The receiver design occupies 70% of the available area.

2.4. Electrical / Optical Interfaces

Optical systems implementing wavelength division multiplexing (WDM) and Ultradense WDM with wavelength spacing as small as 0.05 nm (6.25GHz) have been demonstrated to give a high level of multiplexing [22]. However, such systems require co-ordination between the different users to make sure that no two users transmit at the same wavelength. The multiple access scheme proposed in this chapter, however, requires that there be no coordination. In addition, this scheme is independent of the center wavelength used for the optical transmission, unlike the requirement of specifically designed multiplexers / demultiplexers for WDM systems.

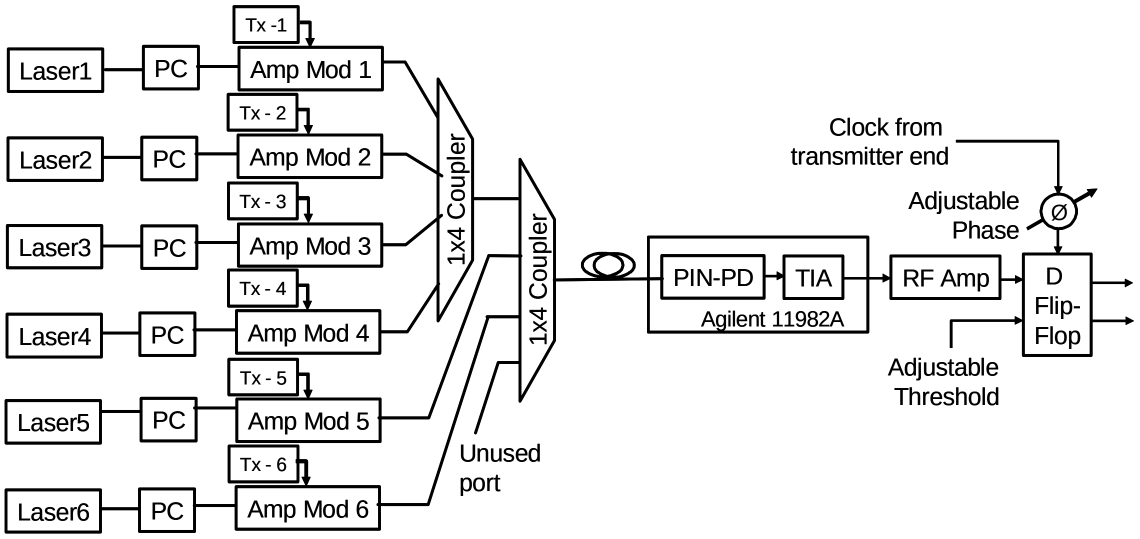

Figure 2-12 Electrical / optical system architecture

The nonlinear trellis codes described and designed here are based on the assumption that there is incoherent addition of the data from the six channels. In other words, the transmission of a ‘1’ from any two users cannot result in destructive interference and will always result in a ‘1’. In contrast, interference from two coherent sources may result in an output of ‘0’. In the implementation of the system, different laser sources were used for each channel, with wavelengths determined independently of each other. Hence, coupling of any two laser outputs can only give a ‘1’ by constructive interference and never a ‘0’.

The electrical/optical system design for the demonstration is shown in Figure 2-12. Six independent continuous wave lasers, centered at 1550 nm are independently modulated with data from each user. They are then all coupled together using two optical couplers and transmitted on a single fiber. On the receiving end, the combined signals are detected by a photodetector. Since the photodetector detects intensity and hence effectively acts as a ‘mixer’ of the different signals, care must be taken about the wavelengths being used for transmitting the data. If two lasers with very closely spaced wavelengths are used, the output of the photodetector would have components of phase noise within the bandwidth of the optical receiver. This led us to choose lasers with wavelength separation of >0.08 nm (10 GHz) since we were using a receiver with a bandwidth of 10 GHz.

In a practical implementation, an optical phase locked loop (OPLL) can be used to minimize the phase noise in the system [23] to a level below other noise sources such as the laser Relative Intensity Noise (RIN) noise [24] or the photodetector shot noise. Under the conditions of the experiments, the system is limited by the shot noise, the spectral density of which is given by $2qP_{total}\eta B$ dBm/Hz where q is the electron charge, P_{total} is the total average optical power reaching the detector, η and B are the responsivity and the bandwidth of the detector. The limitations of shot noise are explained in detail by Saleh and Teich [25]. Efforts have been made to minimize the shot noise level and will be dealt with in the following section.

2.4.1. Optical Transmitter

The first 3 transmitting lasers come from the 3 channels of a Santec External Cavity laser (ECL) while DFB lasers from JDSU and Fujitsu are used for the other 3 channels. The current of the DFB lasers is controlled using an ILX Lightwave Controller. The operating optical powers are to the order of 2-5 mW and at this power, the system is limited by shot noise. The noise at the "0" level can be minimized by allowing a negligible amount of light through the system. This can be obtained by appropriately biasing the optical modulators.

$LiNbO_3$ Mach-Zehnder modulators (MZM) are used as the intensity modulators which modulate the transmitted light with the electrical signal. These modulators work on the principle of the electro-optic effect and have been studied in detail by Wooten et al. [26]. An MZM modulator consists of two identical arms of optical waveguides made of an electro-optic material such as $LiNbO_3$. The refractive index of this material changes proportional to the electric field across it. When the signal applied across the electrodes placed close to the optical waveguides of the modulator changes, the corresponding refractive index variation causes a change in the phase of the optical signal through it. The change in phase is converted to intensity modulation by the interference of the optical signals from the two arms of the MZM. The refractive index change of the electro-optic material is dependent on the polarization of the light. Hence polarization controllers (PC) are placed in the optical path before being input into them. Correct polarization is ensured by adjusting it to give maximum optical power at the output of the MZM modulators.

Two 4 x 1 optical couplers are used to combine the 6 optical channels into a single channel for transmission. Due to the reversible nature of the couplers, each optical channel sees a loss of 6 dB every time it goes through a coupler. All the transmitters are asynchronous with each other.

As mentioned above, the shot noise level of the system is proportional to the average optical power. There may or may not be direct control of the optical output power from the laser itself. Provided the output powers of the lasers are at a minimum, the output power of the MZM can be adjusted by the DC bias applied across the modulator electrodes. In order to minimize the noise, the preferable DC bias should be set close to the minimum output power of the MZM such that the shot noise level is below the thermal noise level of the system. The trade-off at this bias point is that the signal is largely distorted due to the nonlinear characteristics of the transfer function of the MZM at lower DC bias levels [27]. A DC bias is finally set at a point which strikes a balance between the noise level (signal-to-noise ratio) and the nonlinearity and is found by optimizing the bias level until the best BER for any given user is achieved.

2.4.2. Optical Receiver

Two 4x1 couplers combine the optical signals of the six users together. The combined channels are transmitted through a single optical fiber to the receiver end. An HP 11982A lightwave converter, which consist of a p-i-n photodetector (PIN-PD) followed by a Transimpedance Amplifier (TIA), is used to convert the light into an electrical RF signal. The detected RF signal is a result of the data from all the 6 users added together i.e. the sum of the optical powers transmitted by each user. However, since this output is a sum of incoherent data, it follows the properties of an OR channel and transmits a “0” only when all the users transmit a 0. Since the HP 11982A has no limiting characteristics, the amplitude of the output is proportional to the number of users transmitting a “1”.

A D flip-flop following the lightwave converter is used to convert this multilevel signal into a binary signal. The D flip-flop samples the input data at every positive edge of the clock fed into it. The clock is followed by an adjustable RF phase delay line which changes the relative phase between the clock and the signal. This allows the receiver to be synchronized with any desired user. An adjustable threshold is provided to the D flip-flop and the multi-level photodetected output is converted into a binary signal depending on its value relative to the threshold. Thus, the D flip-flop performs the function of retiming and regeneration. The output binary signal is designed to have voltage levels that are recognizable by the FPGA receiver.

2.5. Results

Pictures of the demonstration setup are shown in the following figures. Figure 2-13 shows the laser sources; a mixture of ECL and DFB type lasers is used. Figure 2-14 shows how two of the FPGA transmitters are connected to the optical network. The light travels from left to right and passes through the polarization controllers (A) to the optical modulators (B). The modulation signal is provided by the FPGA (C). Figure 2-15 shows the computer used to display the bit error rate and the oscilloscope to look at the raw received waveform.

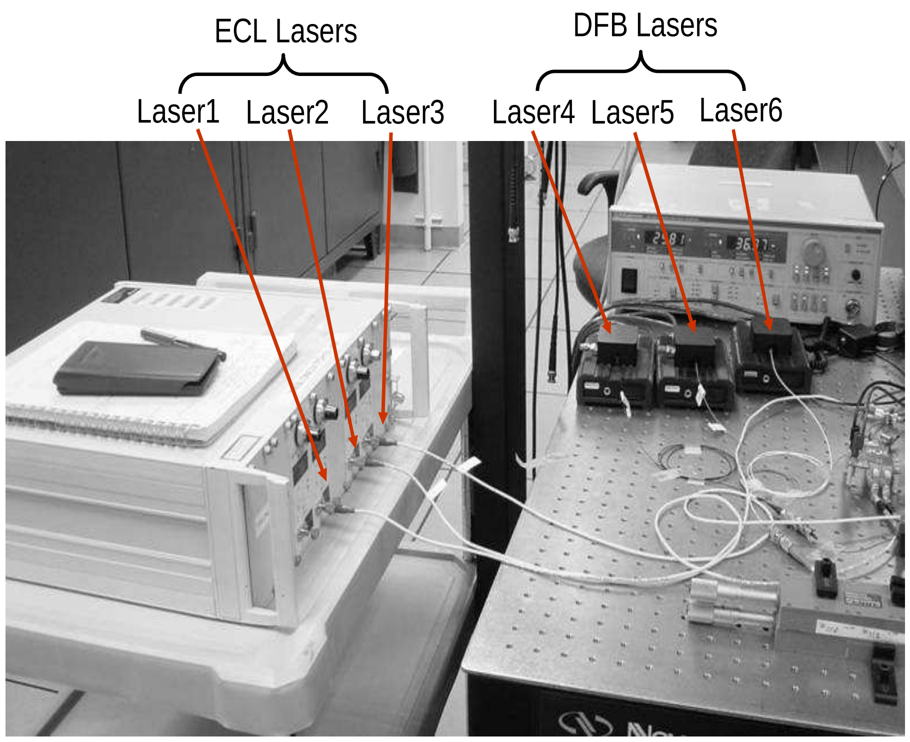

Figure 2-13 Demonstration setup: laser sources

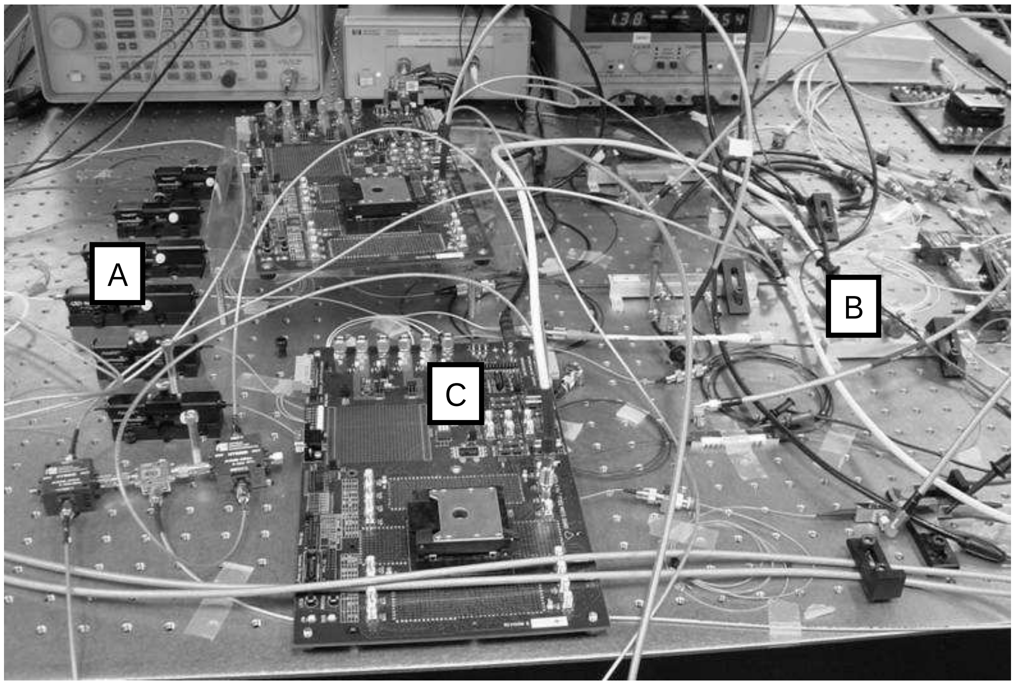

Figure 2-14 Demonstration setup: FPGA's and laser modulators

Figure 2-16(a) shows the raw received waveform for the case of four simultaneous transmitters. After the thresholder and D flip flop circuit, the result is shown in Figure 2-16(b). This is the waveform that is given to the receiver FPGA to decode.

Testing of the system proceeded in the following manner. The desired user transmits a constant pattern in which the receiver FPGA is able to detect. This channel is activated first, and the threshold and sampling moment is adjusted to the correct point. Each of the other FPGA transmitters is set to transmit random coded data. This interference is added to the optical channel one at a time so that the threshold and sampling time may be manually adjusted. This proceeds until all six transmitters are simultaneously transmitting on the optical channel. The result of this is presented in Table 2-2.

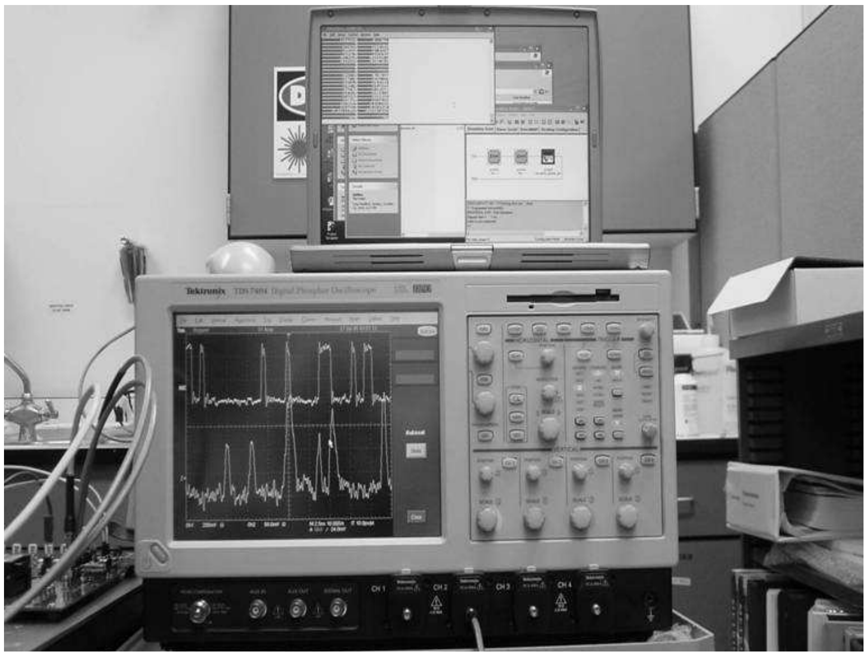

Figure 2-15 Demonstration setup: system monitor and measurement

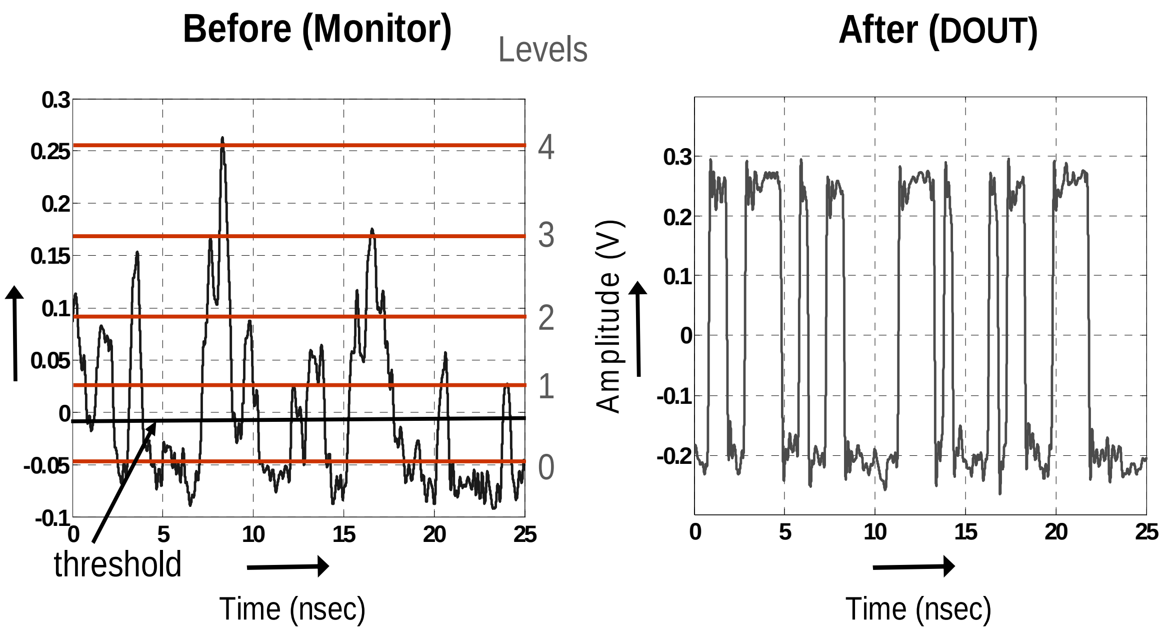

Figure 2-16 Four user case of receive signal threshold and retiming

Thresholding should also be performed automatically in a real system. In this case, the receiving node can adjust the threshold by measurement of the ones density of the received signal. When all nodes are transmitting, a ones density of 0.5 is expected and a feedback loop can be designed to track this.

Table 2-2 System results

Channel rate	1.2 Gbps * 6 = 7.2 Gbps		
Data rate	60 Mbps * 6 = 3.6 Gbps		
Users	Bits tested	Errors found	Measured BER
1	1.5e11	0	< 6.4e-12
2	4.6e10	0	< 2.2e-11
3	1.2e9	0	< 8.3e-10

Due to interfacing problems with the Rocket I/O transceiver, only a system with 3 users at a channel bit rate of 1.2 Gbps was demonstrated. This performance degradation can be attributed to two main factors:

1. the noise of the lasers used
2. the clock and data recovery circuit in the FPGA

As more users are added into the network, the noise floor begins to rise. This decreases the signal to noise ratio of the desired user and contributes to the higher bit error rate.

The high speed serial interface of the Virtex II Pro is a hard IP placed on the FPGA programmable fabric and is therefore not itself programmable. It is designed to receive data with a high degree of transitions. Since the received signal (aggregate of all transmitters) cannot be guaranteed to conform to this specification, there are instances where errors are caused by failure of the clock and data recovery circuit of the high speed serial interface. We are currently working to bypass this circuit on the FPGA so that the incoming data can be clocked with an externally supplied clock

2.6. Conclusions

In this chapter, a high performance design was achieved by first simplifying the interfaces between the physical domains. Processing functions were, therefore, concentrated in the logical domain and novel algorithms (channels codes) were developed and implemented to ensure an uncoordinated system with guaranteed bandwidth is achieved. Using commercial off the shelf components, an operational optical network with high data bandwidth was demonstrated where each user is able to transmit at a channel rate of 1.2 Gbps with a BER of less than 1e-9. To accomplish this goal, the channel codes where co-designed together with the architectural implementation. The optical system was specially designed to be able to interface with the digital hardware. It is only with the close cooperation of these three parts throughout the design and implementation steps that the system is able to function.

This design process was validated through the building of an optical network demonstrator that is able to support 3 simultaneous users transmitting at a guaranteed 1.2 Gbps on a single wavelength.

Chapter 3

Hardware / Software Interface Exploration

The previous chapter discusses how to design efficient interfaces between different physical domains. While they can both be considered part of the same logical domain (their functionality is fundamentally bounded by the properties of logic gates), there is also much research on the interface between hardware and software.

The logical domains of hardware and software have different properties. The main benefit of software is its flexibility and portability; it can be developed remotely and mistakes can be corrected afterwards. Its performance is, however, limited by the processor's instruction set architecture and serial execution. Hardware has much higher performance because specialized circuits can be developed to perform specific operations and parallel computation is possible. The drawback is extra design effort and the increased cost of manufacturing.

To produce a design that meets performance and cost requirements, a standard design process has emerged in industry. First, the software is executed on the target processor and profiled. The sections of code which are shown to be the major bottleneck are then selected to be implemented in hardware as a coprocessor. These accelerators are often connected to the processor through a peripheral bus.

The design focus, therefore, is for performance to be improved through the acceleration of individual computations; a cross-layer approach that considers interfaces has largely been ignored. However, it has been shown that for many applications, the time it takes to pass through the interface takes up a greater amount of time than the calculation itself. Therefore, significant additional performance gains can be achieved by the selections of the appropriate interface technology and optimizing algorithms for it.

In this chapter, a novel system used to analyze and explore the effects of different interface options is presented. The choice of interface impacts both the software (through the software drivers need to communicate with the coprocessor) and the hardware (through the extra logic needed to communicate with the processor). The novel contribution of the system described is that it is able to *jointly* analyze the cost of both these components, so that an efficient interface can be selected.

3.1. Interface Overhead

System performance can be improved by implementing in hardware the operation taking the most time to calculate. Hardware coprocessors enable multiple operations to be computed in parallel and can also speed up those operations with specialized logic. However, while system performance improves significantly over pure software systems, in some applications, the time spent in using the coprocessor accelerator is taken up by the interface; the actual computation on the coprocessor is minimal. Though the addition of a coprocessor provides better performance, additional system performances can be obtained by using the appropriate interface technology to communicate with the coprocessor. In addition, while performance is important in the design of a system, it is not the only factor that is considered in real world design decisions. The choice of interface technology is decided not only on the final system performance but also on factors such as portability of design, application requirements, and development tools support.

GEZEL is a powerful design tool used for both simulation and implementation of complex heterogeneous computing systems. Its main features include the ability to interconnect and simulate IP cores from disparate sources and also the ability to describe in FSMD form its own coprocessor logics. This has led to many successful designs that range in scale from large NOC and SOC systems to small RFID systems.

Interface in this context refers to the way in which a processor is able to communicate with either a specialized coprocessor or another processor in the system. In traditional design, the processors' instruction set simulator (ISS) are connected to other processing elements within a simulation environment. The GEZEL [28] system supports both clock cycle accurate simulation and also has RTL code generation facilities.

In this chapter, the way in which GEZEL is used to support different interface options is presented. In particular, the way in which it can be used to build and simulate the special functional unit interface which enables the building of ASIPs is described. A comparison of the design process for each type of interface is illustrated using the example of the coprocessor for thread management.

3.1.1. Related Work

SystemC [29] and SpecC [30] are system level modeling (SLM) languages that also support interconnection of heterogeneous processing elements. While they support descriptions of custom logic that can generate RTL, the same language is also used to describe behaviors at higher levels of abstraction (these descriptions are not synthesizable). Because of this, the interconnection of ISS and custom logic is also not straightforward and design teams manually translate the SLM description system into Verilog/VHDL. GEZEL provides an IP block construct to connect the ISS to the GEZEL development environment and all behaviors written in GEZEL are able to be automatically converted to synthesizable RTL by default.

The SLM languages are designed for high level design exploration while GEZEL is focused on design and exploration of systems at the implementation level. The Tensilica design environment [31][32] works at a similar level by providing a platform where the designer can modify processor hardware in order to accelerate an application. While this provides the designer with a full set of tools to specify new instructions and functional units, the system only allows customization of its own proprietary Xtensa processor and assumes a special functional unit interface. In contrast, GEZEL allows for easy exploration of different interface options and can be easily adapted to be used with any ISS through its simple IP block interface. Because of this, GEZEL provides a uniform environment that can be used to design small systems, such as those used in smartcards, as well as complicated SOC systems.

3.2. Interface Options

The three interface options that are examined are shown in Figure 3-1. They vary on the degree of integration with the processor core. The following sections examine in more detail the properties of each of the interfaces and how they have been used.

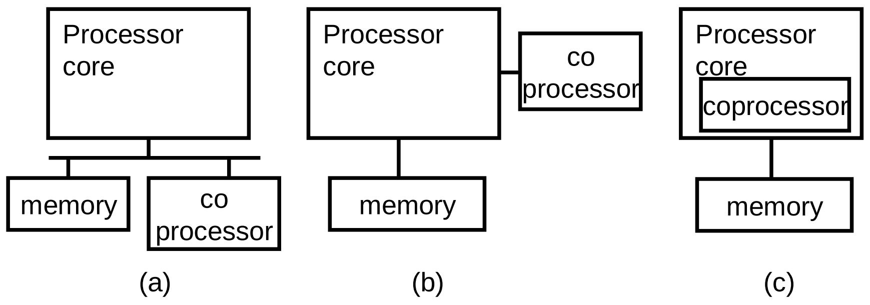

Figure 3-1 Coprocessor interface options (a) memory mapped, (b) coprocessor port, (c) special functional unit

3.2.1. Memory Mapped Interface

The memory mapped interface is the simplest and most flexible interface. Using this interface, software can access external systems by writing and reading data from specific memory locations. The architecture is shown in Figure 3-1(a). For example, in a thermostat application, software will read a value from a dedicated memory location to obtain the current temperature and write to another memory location to program the heater setting.

Much more complicated systems can be built using this interface. One example is the building of a NOC system [33] which connects several independent processors together. The software running on each of the processors communicate with one another through simple network routers described in GEZEL and interfaced through specific memory addresses.

The main benefit of the memory mapped interface is its ease of use and portability. Since most processors have memory systems, software written need only be written once to be able to run on most processors. Also, since reads and writes are treated as memory accesses, data dependency and hazard handling remains unchanged across platforms. Conversely, it is also the properties inherited from memory access that gives the biggest disadvantage; it has the same latency as memory access and performance further degrades when there is contention on the bus. In the ARM processor, the access time has been measured to be 12 clock cycles. For applications that require complex interactions or large data transfers, this option may prove to be too inefficient.

3.2.2. Coprocessor Port Interface

In the coprocessor port interface, coprocessor access is removed from the memory bus and connected directly to the processor (See Figure 3-1(b)). Communication is more efficient in this case since there are no addresses to decode and no wait cycles necessary to wait for slow memory response times.

The GEZEL system can easily interface with these special coprocessor ports. In the ARM core, two such ports are available for use. Their use was demonstrated in the design of crypto-accelerators in [34] and communication delays have been measured to be 7 clock cycles in this system. This special type of port is not only reserved for high performance processors; some microcontrollers also provide this feature. The 8-bit 8051 processor is very popular in small systems such as smart cards and RFID. The use of this port in [35] accelerated the 83-bit HECC algorithm by more than a factor of 200.

Though communications is much faster, there are some costs in terms of both software code portability and flexibility. Access to each of these ports differs according to processor platform; they are accessed through special instructions and protocols so software must be rewritten in order to run on different systems. In addition, the number of these ports for a given processor is limited, therefore, only a limited number of coprocessors can be connected this way.

3.2.3. Special Functional Unit Interface

The decoupled nature of the processor-coprocessor dynamic in the above interface options implicitly requires communication overheads. In ASIP design, the datapath of the processor itself is modified to support increased functionality (See Figure 3-1(c)). Since the functional unit has direct access to the processor register file, communication costs can be minimized.

GEZEL is able to support ASIP design through the use of the special functional unit (SFU) interface. Through this interface, GEZEL can be used to describe the datapath of special instructions that are added to the processor.

In addition to faster communications, system performance can also be greatly increased with specialized processing. The traditional method of acceleration is to use special hardware to speed up complex calculations on a data set. If communications is not too great, memory mapped interfaces and coprocessor port interfaces can be used to obtain acceptable performance. The SFU interface is also able to perform the same tasks but with increased speed due to reduced communications. However, the SFU interface also allows the integration of processing that can affect the control flow of software. By having access to the program counter in the processor core, SFU has the additional ability to jump to any memory location – this is a functionality that is impossible with the other interface options.

While there are many potential benefits in using the interface, software written on such a platform is not portable. ASIP's contain very specialized instructions that operate on specific registers; therefore, special simulators and compilers must be used for the specific development platform.

3.3. Interface Mechanics

Interface between the ISS and the development environment is accomplished through the IP block modules. IP blocks are written in C++ to allow for the description on behaviors that cannot be described by the GEZEL language. In the context of a SOC, they are used to describe the different interfaces between the ISS and the special datapath logic.

For the memory mapped and coprocessor interfaces, there is already kernel support in the design environment. Memory mapped IP blocks intercept memory reads and writes and directs them to the GEZEL environment where logic can be described to operate on the data. Coprocessor IP blocks simulate a buffered coprocessor interface. Reads and writes to the coprocessor port are buffered in the queues of these blocks and made accessible to the design environment.

For ASIP design, minor modifications to the design environment must be made. In the design, Simit-ARM [36] is used as the ISS, and a special IP block is described to interface with the special functional unit port of the ISS. When special machine instructions are called, the IP block is triggered and passes the appropriate register values to the design environment where the logic of the functional unit is described. The results are passed through to the same IP block so that it can be stored in the processor's register file.

Because the logic of the SFU may take several cycles, the ISS itself must be modified to take this into account. In most simulators, this will involve modifications to the sections of code that are responsible for instruction scheduling and resource reservations. The ISS must know exactly when an operation is complete so that subsequent dependent instructions do not get scheduled before the results are available. In addition, certain resources (e.g. registers) may be used by the functional unit and cannot be used by subsequent instructions; the ISS must be modified to prevent these data hazards from occurring.

The ASIP design is done using the Simit-ARM cycle true simulator. The simulator itself is described in a processor architecture language called MADL [59] which allows designers to easily specify the resources used up by each instruction at each stage of the pipeline. Once the architecture is described, the description can be used to automatically generate an ISS that is able to handle interactions with the new instructions and their corresponding functional units.

3.4. ASIP Case Study

To demonstrate the performance and design process of the three different interface options described, the design of a special logic block for thread management is explored. The purpose of this coprocessor is to enable acceleration of the context switch operations in embedded multitasking systems.

The thread manager is responsible for keeping track of all the running processes in the system. It does this by maintaining a queue of all the processes in the system. Processes in the system are uniquely identified by the values in the stack pointer (SP) and program counter (PC) registers.

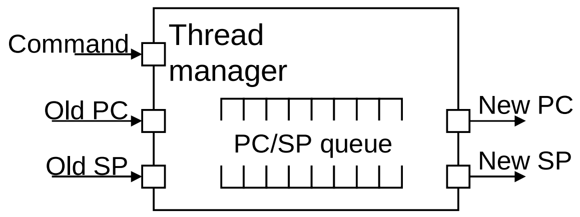

Figure 3-2 Thread manager architecture

Three thread management commands are used to access the thread manager and form the primitives needed for context switch.

- The **create** command takes the current register values of SP and PC and stores them in the queue of the thread manager.
- The **yield** instruction takes the SP and PC register values and replaces them with new ones.
- The **retire** instruction invokes a context switch by replacing the SP and PC with new values from the next process.

In the experiments, the thread manager is interfaced with the ARM processor using the three interface options. The Simit-ARM simulator was used in conjunction with GEZEL to provide cycle accurate simulation. In addition to comparing the final results, the design process is illustrated in detail so that methodology complexity can be compared.

3.4.1. Memory Mapped Interface

The thread manager acts like a special storage element in the system; it stores the PC and SP of all the processes and returns the values for the next process during a context switch. However, under the memory mapped interface, it cannot be responsible for the context switch operation itself. It is the responsibility of the software to supply the current values of PC and SP to the coprocessor to store and to also ensure that the new values of PC and SP are properly stored into the corresponding registers. Though both performance and software size is reduced by the addition of the memory mapped thread manager, these interface tasks add overhead to the actual thread management tasks.

```
  stmfd sp!, {r0-r12, r14}      1
  mov r0, #0                    2
. . . (clear all registers)     3
  mov r12, #0                   4
  mov r14, #0                   5
  mov  r0, #-2147483648         6
  str pc, [r0]                  7
  str sp, [r0,#4]               8
  mov  r1, #1                   9
  str r1, [r0, #8]             10
  str r1, [r0,#20]             11
.LOOP:                         12
  ldr r1, [r0,#24]             13
  cmp r1, #0                   14
  beq .LOOP                    15
  ldr  sp, [r0,#16]            16
  ldr  pc, [r0,#12]            17
  mov  r1, #0                  18
  str r1, [r0,#20]             19
  ldmfd sp!, {r0-r12, r14}     20
```

Figure 3-3 ARM assembly code for the memory mapped yield operation

Figure 3-3 shows the software interface code used to implement the yield instruction and illustrates the software overhead associated with the memory mapped interface. Line 6 moves the base address of the coprocessor to r0. After the PC and SP are loaded to the coprocessor, Lines 9-15 perform the handshaking protocol. The new PC and SP values are not loaded into the registers until lines 16 and 17. The write operation on line 11 signals to the thread manager that a command is available to be processed. It is paired with the read in line 13 which signals that the command is complete. These two operations represent the overhead due to handshaking.

The memory mapped interface also introduces extra hardware in the coprocessor. To facilitate the handshaking protocol, additional ports must be added to signal the availability of a new command and also the completion of a command. Table 3-1 shows the ports used by the thread manager coprocessor. Each of these ports must be buffered by registers. The need for additional registers and a more complex state machine to implement the handshaking protocol contributes to the hardware overhead associated with the memory mapped interface.

3.4.2. Coprocessor Interface

The Microblaze FSL interface on the ARM processor is used to illustrate the thread manager performance in a coprocessor port. The FSL is a dedicated coprocessor interface driven by a simple one way handshaking protocol. The link is buffered by dedicated queues which allow concurrent execution of both the ARM processor and the thread manager.

From the processor side, a write port is available for sending data to the coprocessor. A status port indicates whether data is available on the read port. Compared to the memory mapped interface, data is written to a single coprocessor port; no explicit write command is necessary to signal the coprocessor that a new command is available. However, there remains a ‘acknowledge’ read port that is polled to determine when the thread management task is completed. This one way handshaking reduces the overheads experience in the memory mapped interface.

Table 3-1 Thread manager I/O ports

Port	**Port Type**	**Access address**
Old PC	Input	0x80000000
Old SP	Input	0x80000004
Command	Input	0x80000008
Request	Input	0x8000000C
New PC	Output	0x80000010
New SP	Output	0x80000014
Acknowledge	Output	0x80000018

Figure 3-4 shows the assembly code used to implement the yield instruction. Line 6 defines the location of the write port. Lines 7-10 load the instruction, PC, and SP to the processor; no extra `str` instruction is necessary to signal the coprocessor of a new command. Lines 11-14 poll the status port for the new values. When data is available, the values of SP and PC are taken from a single read port and stored in their respective registers.

```
  stmfd sp!, {r0-r12, r14}
  mov r0, #0
. . . (clear all registers)
  mov r12, #0
  mov r14, #0
  mov  r0, #-2147483648
  mov  r1, #1
  str r1, [r0]
  str sp, [r0]
  str pc, [r0]
.LOOP:
  ldr r1, [r0,#8]
  cmp r1, #0
  beq .LOOP
  ldr  sp, [r0,#4]
  ldr  pc, [r0,#4]
  ldmfd sp!, {r0-r12, r14}
```

Figure 3-4 ARM assembly code for the coprocessor port yield operation

The FSL coprocessor interface adds only minor hardware overhead. Extra logic is necessary to implement the handshaking protocol. No extra hardware, however, is necessary for buffering the incoming commands and parameters. This is because the buffered coprocessor interface already provides this service and is included in the size of the processor.

3.4.3. Special Functional Unit Interface

The special SFU instruction in the ARM processor is used to access the thread manager. It access values directly from the processors register file; it reads values from three registers (SP, PC, Rn) and writes to two registers (SP, PC). All instructions are completed in a single cycle.

Since the ISA is modified to take advantage of the new functional unit, the additional tasks in the processor pipeline must also be specified. The MADL code fragment in Figure 3-5 shows how the new instructions are specified in the processor architecture.

```
// BUFFER STAGE                                1
e_ex_bf_sfu: {                                 2
   // reg 13 corresponds to SP                 3
   // reg 15 corresponds to PC                 4
   v_rm=*mRF[13], v_rs=*mRF[15],               5
   v_rn=*mRF[rn],                              6
    *mSF3x2[0]= (v_rm, v_rs, v_rn),            7
    dst2=mRF[13], dst3=mRF[15]                 8
};                                             9
                                              10
// WRITE BACK STAGE                           11
e_bf_wb_sfu:  {                               12
   (tmp1, tmp2, tmp3) = *mSF3x2[0],           13
    *dst2 = tmp1, *dst3 = tmp2,               14
    !dst2, !dst3,                             15
    !dst_buffer, !dst2_buffer                 16
};                                            17
```

Figure 3-5 MADL code to describe resource usage of new instructions

The code specifies the resources needed for the new instructions in the buffer (BF) and write back (WB) stages. In the BF stage, the registers that are needed for the operation are specified (lines 5-6) and then sent to the external functional unit (line 7). The result registers are also specified here (line 8) so that data hazards can be avoided. In the WB state, values are gathered from the external functional unit (line 13) and assigned to the destination registers (line 14). Note that the destination registers are the same as

```
stmfd sp!, {r0-r12, r14}          1
mov r0, #0                        2
. . . (clear all registers)       3
mov r12, #0                       4
mov r14, #0                       5
yield                             6
ldmfd sp!, {r0-r12, r14}          7
```

Figure 3-6 MADL code to describe resource usage of new instructions

those reserved in the previous pipeline stage.

The modified ISS is connected to the GEZEL development environment through a specially programmed IP block which is triggered only when the SFU instruction is invoked. Through the IP block, the thread manager, which is described in GEZEL, is able to operate directly on the values of the register file. Because of this, there is no interface overhead; performance increases and the software size is reduced. Figure 3-6 shows the assembly coded used to implement the yield instruction. After the register file is stored in the stack and the values are cleared (lines 1-5), the SFU yield instruction is called (line 6). SP and PC values are replaced directly by the thread manager in a single cycle and the register values of the new thread are then popped from the stack in line 7. There is no overhead taken up by the reading and writing of data and parameters to ports or memory locations nor is there any overhead taken by handshaking. Because these operations are no longer necessary, context switches are completed more quickly and require less software.

Using the SFU interface, the thread manager is now part of the processor's pipelines. Because of this, the unit must buffer the input data (i.e. the PC and SP values) in registers. While no extra logic is added for handshaking, the registers do add extra hardware that is not associated with the actual data processing operation and can be considered a form of overhead.

3.4.4. Results

Table 3-2 shows the performance results of the case study when compared with an unmodified ARM processor. In the multitasking system, **yield** is used for the actual context switch and is therefore invoked most often. Based on the performance of the **yield** instruction, the memory mapped interface solution is 2.4 times faster than the equivalent software routine. In contrast, the ASIP solution is 3 times faster (25% faster than the memory mapped interface version).

Table 3-2 Comparison of execution time for thread management operations

Module	**SW (cycles)**	**Mem (cycles)**	**Copr (cycles)**	**ASIP (cycles)**
Create	743	463	459	448
Retire	348	66	61	48
Yield	191	80	71	61

In all the above examples, the thread manager is able to execute the instructions in a single clock cycle. The difference in performance is due to the time taken for handshaking, communications of data and parameters, and the moving of values to the SP and PC registers. The memory mapped interface is characterized by slow handshaking and slow communications due to the use of the peripheral bus. The coprocessor interface benefits from faster handshaking and fast communications due to direct connection to the processor. Both the memory mapped interface and coprocessor interface incur overheads due to the moving of values to the SP and PC.

The SFU interface is not affected by any of these factors as the thread manager is located directly in the processor datapath and therefore has the highest performance. There are no handshaking or communication costs since the PC and SP are taken directly from the register file. The output of the thread manager is also written directly to the register file.

Table 3-3 Comparison of speed and area of interface options

Module	**Size (Kgates)**	**Critical path (ns)**
N-ARM7TM	50	17
ASIP	1.7	7.6
Coprocessor	1.0	7.5
Memory mapped	3.3	7.6

To examine the relative size of the thread manager, the GEZEL description was converted into VHDL. This code was then synthesized using Synopsis. The TSMC 0.18um CMOS standard cell library with conservative wire load model was used. Table 3-3 shows the results. The results were compared to the synthesized N_ARM7TM design from [37]. The additional memory needed for the memory depends on the maximum number threads that the system is designed to support; each additional thread requires 64 bits of storage. Not including the two port memory of the thread queue, the thread management unit only takes 2% of the total ARM processor area. When the different interface options are compared, the memory mapped interface has the largest size due to the extra registers necessary in both the input and output ports. The implementation using the coprocessor port is the smallest since there is no extra registers required within the coprocessor as buffering is included in the interface port.

Table 3-4 Comparison of code size for thread management operations

Implementation	**Size (bytes)**
Quickthreads (SW)	1330
ASIP	252

Coprocessor	352
Memory mapped	404

In the implementations explored, the thread management functionality implementation is moved from software to hardware. Table 3-4 shows the relative software size for the different interface options when compared with the Quickthreads [57], a simple and portable threads toolkit. When compared with a purely software implementation, all the coprocessor options reduced the software size by more than 3x. The remaining software overheads reflect the instructions necessary to transfer data to and from the coprocessor and perform handshaking.

3.5. Conclusions

The GEZEL design environment provides an environment that allows easy exploration of a wide variety of interface options. The cost of each interface option can be evaluated in both software and hardware. While the memory mapped interface and coprocessor port interface has been well used in previous designs, they may still under perform in certain applications. The special functional unit interface allows the designer access to the core of a processor so that coprocessor functions can be directly integrated into the processor datapath. Hardware integrated context switching is one such application that benefits from the special functional unit interface. In this chapter, the detailed design of how the SFU interface is used in the GEZEL environment is described. The resulting architecture improves context switching performance by a factor of 3 while only increasing processor core area by 2% when compared with software solutions.

Chapter 4

Accelerating Control and Communications

In the previous chapter, we showed that the SFU interface outperforms other popular interfaces in system on chip (SOC) design. This is because by integrating specialized processing directly into the processor datapath, the time it takes to transmit control and data to the specialized hardware is greatly reduced.

Though very effective, in many embedded system designs, such intimate access to a processor is not available and ASIP designs are not possible. Thus improvements to traditional interface options are restricted to the use of architectural components external to the main processor.

From chapter 3, it is noted that the interface overhead is composed of the time it takes to transmit control information and the time it takes to transmit the actual operands. The handling of each of these interface elements have different requirements so interfaces can be made more efficient if the elements are separated and their functions accelerated individually. In this chapter, the process of how this can be achieved is described and illustrated with implementation case studies.

4.1. Common Acceleration Techniques

Over the last decade, many important techniques have been developed to improve the performance of embedded systems. Chief among them is the DTSE software technique [38], acceleration of functions in hardware, and direct memory access (DMA) techniques [39]. However, there have not yet been studies that explore how all these techniques can be quickly explored and implemented in a design.

Data intensive calculations are often communications bound. That is, the time it takes to transport operands to a processing unit and store the results back to memory takes a significant percentage of the total time. Thus to improve the performance of embedded systems, techniques to reduce the amount of communications is as important as techniques to accelerate the actual computation.

The software strategy involves the transformation of an algorithm to minimize the time it takes to run. This can be done through minimizing the number and complexity of instructions (computation) needed to perform a calculation and also through minimizing the memory accesses (communications). The latter technique is well studied by [40] and has been proven to be very successful.

The traditional hardware approach accelerates the slow part of the algorithm in hardware by designing a coprocessor. Specialized hardware has the ability to perform complex operations quickly through the design of specialized circuits. In addition, multiple calculations can be performed in parallel, further reducing the time it takes to perform a computation. Compared to software techniques, this strategy is only able to accelerate the computation part of the algorithm.

The two above design steps have become industry standard. However, further performance gains can be achieved through more efficient allocation of data streams to the coprocessor. The communications part of the algorithm can be accelerated through DMA techniques. These are implemented by special hardware components in the computer architecture which transfers data directly from the memory to the hardware coprocessor. While the technology is not new, its integration into the design of accelerators in SOCs has not been explored.

Each of the above techniques can individually significantly increase system performance. However, enhanced performance increases above the sum individual improvements can be observed when several techniques are employed together.

In this chapter, a development environment which allows the exploration of the interactions among these three techniques is developed. We find that while memory access minimization techniques greatly enhance software performance, it also enables efficient data transfers in hardware. Thus significant performance improvements above the sum of the individual improvements can be obtained when the techniques are used together. Using the case studies of GCD calculation and complex matrix multiplication, we show that performance exploration can be performed quickly and lead to designs that increase performance by over two orders of magnitude over base system.

4.2. System Setup

4.2.1. System Architecture

To explore the effects of the various hardware and software optimization techniques on real SOC implementations, we required a hardware programmable platform. The XUP Virtex-II FPGA development platform [41] from Xilinx was chosen due to its wide selection of IP and mature design environment. At the heart of the board is the XC2VP30 FPGA [15] which contains hard IP's such as PowerPC processor, Block RAMs, and 18x18 multipliers. These hard IP can be interconnected through configuration of the FPGA fabric. The easy reconfiguration properties of the FPGA make it a good platform to study different processing architectures and their effect on performance.

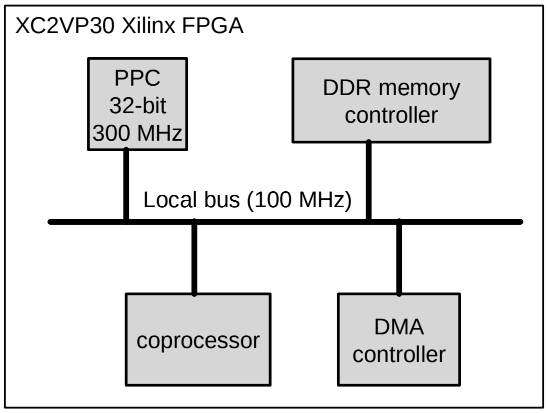

Figure 4-1 Base SOC architecture

Figure 4-1 shows the base architecture of our processing system. In the base architecture, the algorithms are implemented on the 32-bit PowerPC 405 processor core running a 300MHz. Software is stored on off-chip DDR RAM through a 64-bit interface. Due to the double data rate operation, the RAM can provide the FPGA with 128 bits of data per FPGA clock cycle.

The PowerPC is interfaced with the external memory and the general FPGA fabric through the processor local bus (a 64 bit bus running at 100 MHz). It is through this bus that hardware accelerators and memory access accelerators can be interfaced to the main processor.

4.2.2. Design Environment

A series of design tools are used to enable quick exploration of design alternatives and easy implementation of the designs onto the FPGA platform. Figure 4-2 shows an overview of the proposed design methodology.

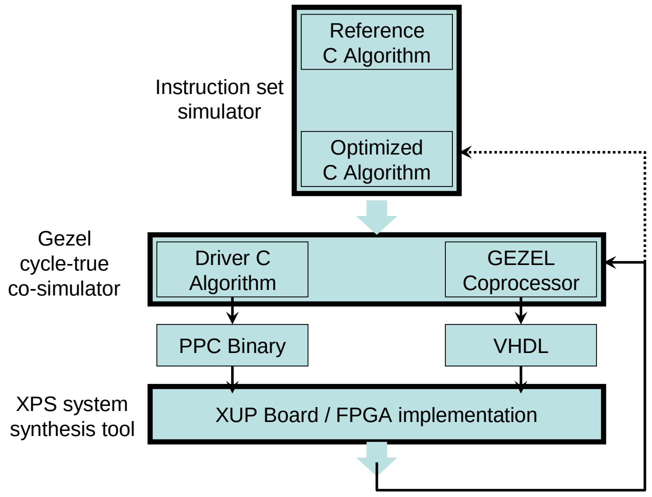
Reference C Algorithm
Instruction set simulator
Optimized C Algorithm
Gezel cycle-true co-simulator
Driver C Algorithm
GEZEL Coprocessor
PPC Binary
VHDL
XPS system synthesis tool
XUP Board / FPGA implementation

Figure 4-2 Design methodology

The design exploration is performed using the Gezel design environment [28] which enables cycle true hardware/software co-simulation. As with its use in the previous chapter, the software and hardware components can be quickly co-simulated and verified without register transfer level simulations required. Accurate system performance estimations can be used at this level to evaluate different design options. In contrast with the use of Gezel for simulation, in this chapter, it is also used to provide a easy path to implementation. Both the software and hardware descriptions that runs in the simulation environment can be directly inserted into the base XPS computing platform.

Implementation of the final design is performed using the Xilinx XPS design environment [42] which is able to generate the generic system architecture. This includes components such as system buses and peripherals which are connected to the PowerPC processor on-chip. The custom hardware designed in the Gezel environment is automatically converted to VHDL, enclosed in hardware wrappers, and integrated into the system architecture. The resulting SOC (including the custom hardware) can then be synthesized and written onto the FPGA.

The software portion of the design is implemented in standard C and can therefore be easily cross-compiled to the PowerPC processor. The interface with the custom hardware is similar to the one provided by the Gezel environment so the driver code need not be radically modified.

In addition to custom hardware accelerators which accelerate computation time, hardware blocks can also be used to accelerate the dataflow element of the system. Direct memory access (DMA) blocks are an example of such specialized hardware that can be added to the design at this stage to further accelerate the system performance.

4.2.3. Optimizations

Each of the optimizations discussed in this subsection infers a particular control and dataflow architecture. The relevant architectures are shown in Figure 4-3.

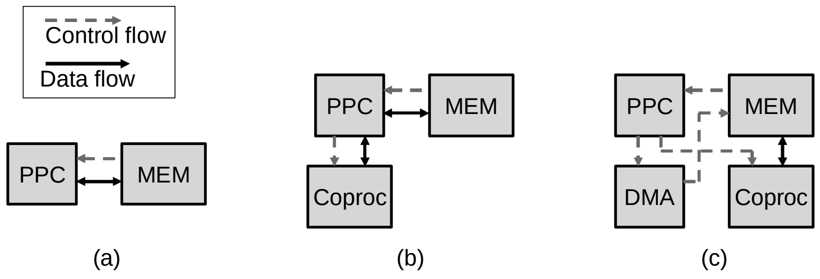

Figure 4-3 Control and dataflow architectures for (a) base architecture (b) traditional coprocessor architecture (c) datapath accelerated coprocessor architecture

Software optimizations imply a basic computer architecture (Figure 4-3(a)) where the processor is connected to external memory through a bus. Both dataflow and control information is transported along this single bus. Improvements in this architecture can focus on accelerating the algorithm or reducing the amount of data that travels between the processor and external memory.

In addition, data access patterns can be optimized to take advantage of the properties of the local bus. Data transfers between the processor and external memory can be optimized by changing the software data access pattern. In our platform, the time it takes to access 128 bits of data is the same as the time it takes to access 32 or 64 bits. Therefore, software written for this platform should try to access memory in 128-bit blocks. In addition, linear accesses provide the greatest throughput to the DDR RAM. Each DDR access requires a 1 cycle latency for the command, 2-3 cycles to access the location, followed by the actual data transfer. By utilizing sequential memory locations, the data can be transferred in bursts and therefore minimize the command and access latencies.

Custom hardware can be used to accelerate calculations to a much higher degree than software techniques. This is because hardware can be designed especially for complex calculations and can also be easily parallelized. The technique is well used and has become an industry standard for embedded systems. Though the computation task is greatly accelerated in the resulting architecture (Figure 4-3(b)), the time it takes to transport data is increased. This is because data between the memory and coprocessor must pass through the PPC; compared to a software only solution, the amount of communications is doubled.

To accelerate the communications part of a computation, a DMA block can be used (Figure 4-3(c)). This block allows data to flow directly between the coprocessor and external memory. Thus the dataflow time is improved by both the reduction and acceleration of data transfers

Direct memory access (DMA) is a hardware solution to accelerate data transfers. In software data transfers, memory accesses to a hardware coprocessor must travel through the PowerPC processor. A DMA unit accelerates the process by allowing blocks of data to be transferred directly to the coprocessor in a burst. Thus performance improves due to two main factors. One, instead of each data item hitting the bus twice, each item is only transferred once. Two, the DMA controller is able to take advantage of both bus and DDR RAM burst access.

Our design environment allows easy exploration of all these design alternatives. The integration of DMA technology to the evaluation process greatly enhances the design choices and enables additional performance gains. In the following case studies, we show that the use of DMA technology not only improves performance when used alone but its benefits are further enhanced by software optimization techniques.

4.3. Case Studies

To compare the benefits of the proposed methodology on different types of design we study the design to two case studies. The case studies illustrate the degree of improvements that is achievable for specific algorithms.

4.3.1. Greatest Common Divisor

Euclid's binary greatest common divisor (GCD) algorithm is a well used algorithm known to be simple and fast to implement. Its software description is shown in Figure 4-4. The function takes in two numbers and results in a single number which is the GCD of the two operands.

In the reference software implementation of this algorithm, a while loop successively reduces the two operands until one reaches zero. In each cycle of the loop, the least significant bit (LSB) of the operands is examined and shift and/or subtraction operations are performed depending on the result.

```
unsigned int gcd(unsigned int u, unsigned int v)
{
  int shift;

  if (u == 0 || v == 0)
    return u | v;

  for (shift = 0; ((u | v) & 1) == 0; ++shift) {
    u >>= 1;
    v >>= 1;
  }

  while ((u & 1) == 0)
    u >>= 1;

  do {
    while ((v & 1) == 0)  /* Loop X */
      v >>= 1;

    if (u <= v) {
      v -= u;
    } else {
      int diff = u - v;
      u = v;
      v = diff;
    }
    v >>= 1;
  } while (v != 0);

  return u << shift;
}
```

Figure 4-4 Reference GCD algorithm

This is a very sequential algorithm and software implementation is limited by control instructions (i.e. compare and branch instructions). At the algorithmic level, there has been some work to reduce the asymptotic complexity of the algorithm [43]. More practically, software optimizations focusing on loop unrolling techniques have show performance improvements of more than a factor of two [44].

```
fsm euclid_ctl(euclid) {
  initial s0;
  state s1, s2;

  @s0 (init) -> s1;

  @s1 if (done) then (complete)          -> s2;
      else if (m[0]&n[0])  then (reduce, outidle) -> s1;
      else if (m[0]&~n[0]) then (shiftn, outidle) -> s1;
      else if (~m[0]&n[0]) then (shiftm, outidle) -> s1;
      else (shiftn, shiftm, shiftf, outidle)   -> s1;

  @s2 (outidle) -> s2;
}
```

Figure 4-5 Finite state machine for GCD coprocessor

A hardware implementation of the binary GCD algorithm is an easy way to improve performance because complex control operations can be performed very efficiently in hardware. Figure 4-5 shows the state machine description of the GCD coprocessor. Compared to the software version where several compare and branch operations are required in each loop iteration, the hardware implementation is able to perform each iteration in a single clock cycle.

Once the control overhead is reduced through the use of a coprocessor, the time it takes to transfer operands from memory and store the results back to memory becomes the major bottleneck. A DMA unit is added to the architecture to accelerate memory accesses. The execution times of the systems produced at each step of the design is shown in Table 4-1 for operands of various sizes. Table 4-2 shows the speed up in each case.

Table 4-1 Execution time for GCD implementation options

microsec	**32**	**64**	**128**	**256**

SW	503.7	1136.08	2104.46	3615.08
FIFO	83.72	160.26	312.88	618.62
DMA	46.3	83.18	157.14	304.84

Table 4-2 Performance speedup for GCD implementation options

speedup	**32**	**64**	**128**	**256**
SW	1.00	1.00	1.00	1.00
FIFO	6.02	7.09	6.73	5.84
DMA	10.88	13.66	13.39	11.86

On average, the hardware implementation of the algorithm shows a speedup of a factor of 6. In the software implementation, each loop iteration has at minimum, 4 comparison instructions and 2 branch instructions; this suggests that most of the improvements are due to the efficient control structures in hardware. Addition of the DMA unit further improves performance by a factor of two. This number is directly related to the number of times that data is transferred in the bus. Since this operation does not involve the transfer of large amounts of data, burst mode has not been activated and larger speedups are not realized.

4.3.2. Complex Matrix Multiplication

In the second case study, an algorithm that involves more complex calculations and larger data transfers is selected. The complex matrix multiplication operation is chosen and the software code is shown in Figure 4-6. The algorithm multiplies two NxN matrices to produce the answer matrix. The software reference algorithm is comprised of three nested loops to access the matrix array elements. Note that the multiplication operation is complex and is actually composed of 4 multiplication operations and 2 addition operations.

```
void mmmKernel(Number* A, Number* B, Number* C, int N)
{
  int i, j, k;
  for (j = 0; j < N; j++)
    for (i = 0; i < N; i++)
      for (k = 0; k < N; k++)
        C[i*N+j] = C[i*N+j] + A[i*N+k]*B[k*N+j] ;
}
```

Figure 4-6 Reference code for matrix multiplication

This case study promises large potential for performance improvements. In the calculation portion, the complex multiplication operation can be computed efficiently in hardware. In addition, the computation of elements of the answer matrix is inherently independent of each other so a high degree of parallelization is possible. Large data sizes, especially in cases where N is large, can be greatly accelerated through DMA since burst mode can be taken advantage of.

To make use of efficient memory access, a matrix multiplication algorithm was devised to linearly scan through the RAM and perform 32 complex multiplications per cycle. The algorithm begins by sending eight blocks of A values (a block is 4 elements) to the coprocessor. The eight blocks are from subsequent rows in A. Next, a block from B is sent to the coprocessor. The four incoming B elements are multiplied by the first element in each row of the A, and each of the 32 partial C products is accumulated. After a full row of B has been processed, the next active elements in A are the eight elements in the subsequent columns. The process of sliding over B's rows, and A's columns is continued until the entire B matrix has been traversed. At this point, the complete product for the first 8xN elements of C has been computed and can be written back to RAM. The algorithm then repeats itself computing eight rows of C at a time until all of the C results have been calculated.

With the plan to perform linear scans through the RAM, 4 elements at a time (128-bits), the design also attempted to optimize the use of the FPGA's hard IP resources. The Virtex-II XC2VP30 provides 136, 18-bit x 18-bit multipliers in the FPGA fabric, along with 136, 18Kb SelectRAM blocks. To utilize these fast discrete resources, the design performed 32 complex multiplies and accumulates per cycle. Since a single complex multiplication requires four multiplications and 2 addition/subtractions, a total of 128 multiplications, and 64 addition/subtractions were performed per cycle. In addition, one accumulate was needed for each of the 32 real and imaginary components,

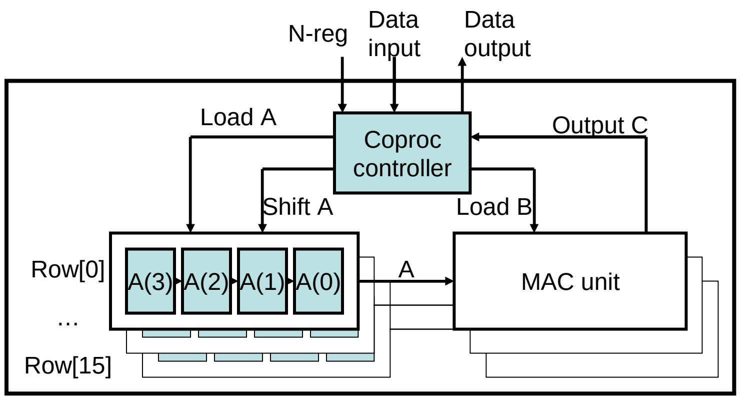

Figure 4-7 Scalable coprocessor architecture

resulting in another 64 addition / subtractions each cycle. Therefore, the final design utilizes 128 out of the 136 multipliers and performs 128 addition/ subtractions per cycle.

An interesting characteristic of our software algorithm is that the algorithm leads to a loosely coupled and scalable coprocessor architecture. The coprocessor has an input FIFO, output FIFO and single register called N-Reg. The controlling processor writes the dimension of the matrices into N-Reg. For a given coprocessor configuration, N-Reg is the only control parameter that needs to be set at runtime. Based off the single N-Reg parameter, the coprocessor's entire control FSM is determined. Therefore, in addition to a simple runtime configuration, the coprocessor's control logic is captured by a low-overhead FSM. The coprocessor's architecture also allows the designer to configure parameters optimal for their specific system. For example, Figure 4-7 displays a coprocessor with a single MAC row. To implement a coprocessor that computes eight rows in parallel, a single design parameter is set to add the additional computational capacity. In addition to computational capacity, the internal datapath width to the "Aunit" and "MAC Unit" is configurable as well. In fact, a design was tested and verified with a configuration of (Aunit = 8x4, MAC = 8x4) and (Aunit = 16x4, MAC = 16x2) within minutes of each other. Therefore, the coprocessor architecture can scale to utilize a system's available resources.

Following the above design process, three different systems were implemented. The first is a software only system that improves performances by optimizing memory access patterns. The resulting transformed algorithm is then implemented as a coprocessor. Further gains are achieved when a DMA controller is used to directly supply the data elements from memory. The execution times of the scenarios are shown in Table 4-3. The corresponding speedup compared to the reference software is shown in Table 4-4.

Table 4-3 Execution time for matrix multiplication implementation options

microsec	**128**	**256**	**512**	**1024**
SW	250736	3015457	45354321	362205433
OPT SW	169596	1414189	14411677	114796740
COPROC	30450	212524	1574459	12060870
DMA	5975	40248	165489	210370

Table 4-4 Performance speedup for matrix multiplication implementation options

speedup	**128**	**256**	**512**	**1024**
SW	1	1	1	1
OPT SW	1.5	2.1	3.1	3.2
COPROC	8.2	14.2	28.8	30.0
DMA	42.0	74.9	274.1	1721.8

From the results, software optimizations to reduce memory accesses provided modest speedup to the system. For N=1024, the software technique is able to improve performance by 3x; this suggests that data transfers are the dominant performance bottleneck as N gets large. For small matrices, the coprocessor implementation added an additional 3x improvement and the DMA implementation added an additional 3x improvement above that. In contrast, for large matrix sizes, the measured improvements is 10x and 50x respectively. This result shows that though software optimization techniques may bring only modest speedup, its combination with hardware techniques can realize performance increases of more than three orders of magnitude.

4.4. Related Work

The methodology presented in this chapter shows how high performance systems can be produced in hardware / software co-design techniques are also used to optimize control and dataflow components of a computation. This idea has already been taken advantage of in the family of stream processors [46].

The graphics processing unit (GPU) is the best know type of stream processor and is a good example of the performances that can be achieved when the control and dataflow elements are separated and optimized separately. The generic GPU architecture consists of a deep pipeline designed to process an input data stream. Software written for the GPU runs on a specialized processor core designed to supply parameters to each of the pipeline stages. Dataflow is optimized because data passes directly from one processing unit to the next. Control is optimized by the use of a processor core specially design to only provide parameters to the processing units.

The Cell processor [47] is an example of a more programmable design that explicitly optimized control and datapath elements separately. The processor consists of one PPE connected to 8 SPUs through an internal high speed bus. The SPU processors are optimized for arithmetic calculations and programmed by the PPE. Each of the 8 SPE also contains a Memory Flow Controller that allows the data to be passed directly from one SPU to another. This enables the PPE to specify how data flows between the SPUs.

In recent benchmarks [48], the Cell processor was able to achieve 60x speedup on the 256x256 matrix multiplication operation when the reference code is optimized for computation and communications for a single SPU. The SPU is considerably more powerful than the PPC405 processor used in our case studies, so it is no surprise that it also outperforms our design. However, when all SPU cores are used, the benchmarks show a 8x performance increase. This shows that through good dataflow designs, communication overheads can be minimized.

The resulting performance demonstrated is impressive and validates the efficacy of this approach. However, besides the high cost, there are several reasons why the cell processor is not suitable for embedded systems in general.

1. The SPU while optimized for arithmetic functions is still restricted by the ISA and sequential processing paradigm of processors.
2. The heterogeneous cores in the Cell processor create complications in the porting of algorithms. Taking advantage of control and dataflow optimizations for the architecture often requires a complete redesign of algorithms and software.

From a certain point of view, to realize high performance gains, the algorithm must be molded to fit the architecture.

In contrast to the design approach of stream processors, we present a design methodology which allows explicit optimization of the control and datapath of an embedded system. In contrast to designing a system based on the Cell processor, our process is based on the system-on-chip (SOC) design approach. Thus, like the design of the GPU, a system can be built to fit the specific application. Unlike the GPU design, however, the degree of programmability in the control and datapath elements may be specified. The GPU has a fixed dataflow and control limited to the specification of computational parameters. The Cell processor enables both the dataflow and control element to be programmable, though SPUs restrict the granularity of optimizations.

Our methodology provides a unified tool and environment where all these elements can be explored. The design flow also enables a quick path to implementation which is important to embedded systems where design time is an important factor.

4.5. Conclusions

It is clear that there are many techniques both in the hardware and software domains that can be used to improve the performance of embedded systems. Three of the most popular are: reduction of memory accesses in software, acceleration of functions using coprocessors, and acceleration of memory accesses direct memory access (DMA) units. They are popular in industry because they follow a simple design methodology while producing large performance improvements.

However, there has been little study on the effects of these methods when used in combination. In this chapter, we introduce a design environment which is able to explore the interactions of these design techniques (which extend through hardware / software boundaries). Through the use of two case studies, we show that for some applications, the performance gains from the combination of these techniques are more than linear. The GCD case study shows that an order of magnitude performance increase can be had by accelerating a computation in coprocessor hardware and then accelerating the communications with a DMA module. However, in the matrix multiplier case study we show that if the algorithm is first transformed to minimize data transfers, the performance gains achieved by implementation of the new algorithm is greatly multiplied. Results show performance improvement of up to three orders of magnitude in certain cases.

The design environment also flows easily into an implementation. The smooth design flow can be attributed to our design environment which is able to guide the design of the system architecture so that it is optimized for the application. This is in contrast to stream processors where the application must be modified to fit the architecture. Faster design times are a significant advantage for an embedded system design since they are very sensitive to time to market. A system that is able to realize high performance while minimizing design time is attractive. Our design environment enables the easy exploration of proven techniques to achieve high performance and a framework to quickly realize the resulting system.

In traditional design, the computing architecture is first defined and software is then developed to be optimized for it. Our design environment, in contrast, uses hardware / software co-design techniques to encourage the design of a computing architecture that is optimized for the specific application.

Chapter 5

Localized Security Interface

In this chapter, the interface between logical entities (processes) which share a common processor is examined. Multitasking computing systems have become main stream even in the field of embedded systems. With the ability to run multiple applications on a single platform, comes the real concern that some of these applications may be malicious.

The interface between different tasks in a system is a purely logical one but is also different in purpose than the interfaces discussed so far in this dissertation. The interfaces discussed so far divide up the system horizontally and are designed so that data can travel easily through. In contrast, security boundaries divide up the system vertically and are designed so that data cannot travel through them. The boundary is vertical because, though there is much sharing of resources between the different applications running on a single system, data from one application must not be accessible by another process. To prevent loss or corruption of data and software, the interface between processes must be strengthened.

Though the purpose of the two types of interfaces may, at first glance, appear to be conflicting, the cross-layer co-design methodology can still be used. In this chapter, the way in which this can be accomplished is presented through the design of a multi-threaded coprocessor architecture. In the design, a shared security function is first identified. The isolation and access control layers are then collapsed into a single hardware domain where they are optimized together with the crypto-algorithm. The resulting designs exhibit both high performance and high security.

5.1. Functional Isolation

Security for multitasking systems focus on the implementation of two main goals: resource access control and resource isolation. Resource access control is the assignment of usage permissions for system resources or functions to processes in the system. Classic security systems design focus mainly on this task; this is usually done by explicit assignment of access rights to each running process and controlling the interaction between them and various system resources (such as files, IPC, and the network stack). Resource isolation tries to ensure that data from one process is not able to leak to another process. Historically, this feature is the responsibility of the operating system and has been implemented without assuming the presence of a malicious process in the system.

In a perfect system, a malicious process can be perfectly identified and its access to system resources appropriately restricted. In such a scenario, the task of isolation is not important since the malicious process does not even have access to the resource. The contributors of the Trusted Computing Group [64] focus on this idea by building processors supported by a complex trust infrastructure to ensure that only trusted software is allowed to be installed or executed in the system. Though well promoted by industry, the effectiveness of such an approach is debatable [51][52].

In a real system, it is very easy for a malicious program to gain access [65]. In such a situation, it is important that the process is not able to crash the system or, more importantly, steal and/or use information from other resources in the system. Therefore, in the absence of perfect access control, this chapter focuses on the implementation of a system with stronger process isolation properties; malicious programs may run, but they cannot interfere with other processes.

Process isolation is not a new idea. One of the earliest implementations is the Unix process which was designed to prevent several users of a mainframe computer from interfering with each other. Since it was not designed with security in mind, malicious users have exploited the weak isolation. Though there are many OS security patches released, new security holes are discovered daily. It has been argued that this is due to the monolithic nature of traditional OS's and the large amount of code that is allowed to run in kernel mode [55].

5.1.1 Related Work

From its early days, process isolation has been thought of as a software problem in the domain of operating systems. In [55], Tanenbaum argues convincingly that a secure operating system should be exclusively composed of a set of interactive processes (microkernels). This method forces a certain degree of isolation between the different processes responsible for the various operating system functions. Due to this isolation, a security breach in one of the processes does not necessarily mean a compromise to the whole system. In addition, partitioning of the OS functions allows the designer to minimize the amount of code that is allowed to run in kernel mode.

A popular software solution to isolation is the virtual machine. Virtual machines [50] allow several operating systems to run on a single processor; isolation is enforced by a virtual machine monitor which restricts communication between the different operating systems (in fact, one operating system is not even aware of the existence of another).

Mixed hardware/software solutions include processor architectural features that try to protect against common known attacks. The ARM Trustzone [61] processors have an additional security mode which allows trusted software to access additional secure registers and memory management units. In the age of inexpensive silicon, SOC designs are able to dedicate a whole processor to run untrusted software [62]. In the CELL processor [63], each of the eight Synergistic Processor Elements (SPE) can be configured as a secure processing vault which ensures that the application running on it is isolated.

5.1.2 Functional Process Isolation

Guaranteeing process isolation to a whole system is a daunting task. However, in most systems certain functions or resources can be identified as needing more protection than others. In general computing, the keyboard or the disk drive may be a major source of information leak. In embedded computing, isolation is particularly important in the calculation of security primitives and protocols. The French company, Trusted Logic, in its TL Security Model Architecture [53] for mobiles has used this idea to build a small and secure security kernel. Isolation properties are very strong within the kernel, while other functions are left to a standard operating system.

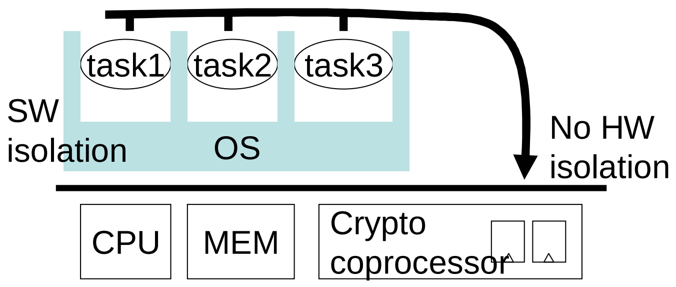

Figure 5-1 Lack of hardware isolation in current architectures

Software systems are also vulnerable to cache attacks as describe by [69] and successfully demonstrated by [67] on the AES algorithm and [68] on the RSA algorithm. The attacks involve a malicious process observing the pattern of cache misses in the system in order to obtain the secret key. While there have been some cache designs [66] that claim to be resistant to such attacks, such solutions are not widely available to common processors. Coprocessors are a better solution since they can be easily added to the peripheral bus of most processors and access to them does not pass through the memory subsystem. However, such coprocessors must still be carefully designed to ensure that information does not leak between independent processes through the coprocessor. As Figure 5-1 shows, even though a secure operating system may provide isolation services, the coprocessor still can leak information between independent processes through the coprocessor registers.

In this chapter, the use of coprocessors for process isolation is motivated. Compared with traditional isolation techniques that are defined in terms of fixed processors and fixed architectures, the major difference with the proposed coprocessor centric approach is that isolation can be provided to arbitrary architectures. In addition to higher performance, a well designed interface can guarantee isolation during computation.

5.2. Hardware Isolation Motivation

In order for a hardware based approach to be valid, the proposed system must realize properties that significantly improve current process isolation schemes. The main benefits of hardware are the potential of increased security and better performance. We propose a secure coprocessor design that will provide strong process isolation properties. Like the TL Security Model Architecture, the coprocessor will provide a specific function or service for the various processes in the system. The following examples of a coprocessor that provides an encryption service illustrate how coprocessor architectures are currently not capable of providing these properties. In the examples, a processor is making use of a coprocessor that provides encryption and decryption services to multiple processes.

5.2.1 Security Examples

One possible attack can occur at the moment that a process has just used the coprocessor to encrypt a command to a remote server. A malicious process may then use the same coprocessor without reprogramming the settings to send its own data to the remote server, essentially spoofing the identity for the first process.

In a related attack, the malicious process can partially reprogram the coprocessor by just changing modes from encryption to decryption. The previous encoded output can then be reinserted into the coprocessor to reveal the unencrypted message.

With current coprocessors, this can be prevented by having the tasks reset the coprocessor after each operation. However, this solution imposes additional overhead, because it increases the period within which the coprocessor interface will remain locked by a single task. Moreover, it leaves the responsibility of security on the design of the operating system. Operating systems themselves are very complicated software structures.

5.2.2 Performance Examples

Assume that a single coprocessor encrypts two channels of streaming data. Each channel has a different mode of operation and different keys. The streaming data is time sensitive and must have its delay bounded by a certain value. For a traditional interface, this would mean that the operating system would have to manage the context switching between these two tasks. The overhead due to software context switching and repeated interactions with the coprocessor is a limitation on the total throughput of the system.

A system with many processes having bursty data illustrates an extreme example of context switching. This situation is not unusual in a VPN server, which handles large number of secure interactive sessions.

The coprocessor can be designed to store the contexts (process specific data such as the key and mode of operation) of the processes in the coprocessor. This means that for streaming applications, the bandwidth of the encryption core is shared among the active tasks with no time lost to context switching. For extremely bursty traffic, the context information is already stored in the coprocessor; therefore, overheads associated with processor-coprocessor interactions are minimized.

5.3. Agent Based Coprocessor

To show how process isolation can be implemented, a coprocessor based scheme is proposed that is able to provide significant benefits in security and performance. In the following subsections, a general system architecture where the coprocessor exists is first defined. Then a protocol for communication with the coprocessor is defined before the actual architecture of the coprocessor itself is presented.

5.3.1 System Architecture

Figure 5-2 shows three possible system architectures for a coprocessor and its communications with a CPU. The dashed lines represent the flow of data and the solid lines represent the flow of control signals. In Figure 5-2 (a), the coprocessor directly connects to the microprocessor. This means that both data and control signals must pass through the microprocessor, making this the main bottleneck. In the second architecture (Figure 5-2(b)), the coprocessor connects directly to a streaming interface that will supply the data. With this architecture, the microprocessor only deals with control of the coprocessor and interactions proceed on a block-by-block basis rather than a word-by-word basis.

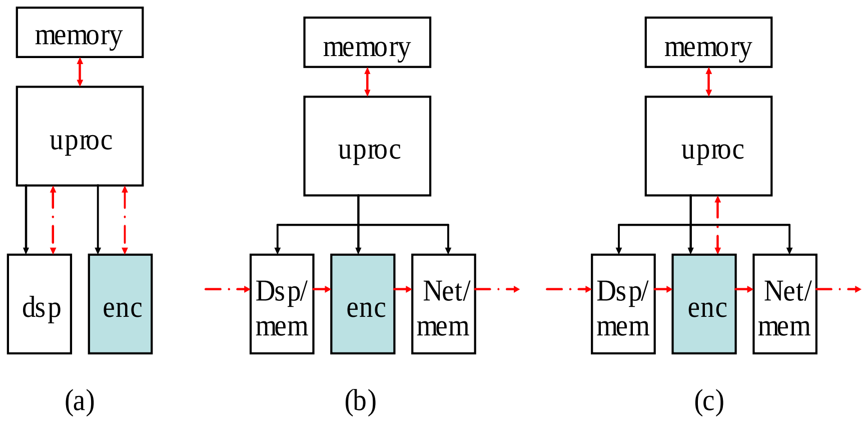

Figure 5-2 Different system architectures: a) simple coprocessor interface, b) streaming coprocessor interface, and c) hybrid coprocessor interface

The most significant source of communications overhead is the transfer of context information. In an encryption application, context is the process specific data such as the control state, the secret key, the mode of operation, and initial values. The transfer of this information for each process not only reduces system efficiency, but also increases the risk of data interception.

Figure 5-2(a) may seem to incur a large overhead but has the benefit of a more traditional architecture and is more appropriate for interactive type applications such as a telnet session. Figure 5-2(b) is more appropriate for processing of continuous streaming data or large data blocks. The proposed design must to be compatible with either of these common architectures. An example of how this is accomplished in shown in Figure 5-2(c).

5.3.2 Interface Protocol

The security example shows how information can leak between two processes in the system in ways that are not obvious. To ensure process isolation, a method to physically keep all context information between the running processes separate must be derived. In addition, a mechanism is needed to bind the software process with the hardware context such that it can not be accessed by unauthorized processes. To accomplish these two tasks, an interface protocol was developed.

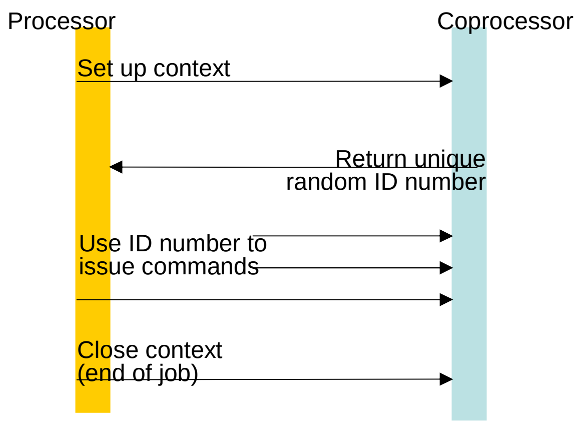
Processor
Coprocessor
Set up context
Return unique random ID number
Use ID number to issue commands
Close context (end of job)

Figure 5-3 Processor - coprocessor interface protocol

The coprocessor interface communicates with the main processor and distributes work to the agents. Using the interface protocol, a software process is able to set up a job by sending parameters to the coprocessor. This context is stored in the coprocessor until the software process terminates the job.

Figure 5-3 shows the basic interface between the coprocessor and the processor. After sending the parameters necessary for a particular operation, the coprocessor will confirm receipt by issuing a unique random number. This number will be used by the software application in the future to issue commands to the coprocessor for the specific channel. This unique ID number makes it difficult for other software processes (in charge of a different data stream) from accessing another's job parameters or internal registers.

From the point of view of a software process running on the main processor, the use of the ID numbers is invisible and is managed by the API drivers. When a process is created, a memory location is allocated to store the ID number. The API setup function will store the ID number at this location when a channel is created. In subsequent API function calls, the number will be used to communicate with the coprocessor. Using this API, the software process does not have access to the ID number and cannot give this number to another process; this means that the link between a software process and its hardware agent is tightly coupled. Table 5-1 shows the five types of commands that may be available.

Table 5-1 Commands accepted by coprocessor

Command	Description
Agent setup	Reserve/configure an agent for a calculation
Agent check	Check to see if previous calculation has been completed
Agent release	Clear a certain context from the coprocessor
Agent single	Perform a single preset calculation
Agent continuous	Perform a series of preset calculations taking data directly from memory
Immediate	Perform a single calculation

A malicious process can try to access a context by trying to guess a valid ID number. The processor interface mitigates the effectiveness of such an attack by not revealing any information about whether an ID number is valid or not. When the agent check, agent release and agent continuous commands is given with an non-existing ID number, the coprocessor will always respond with DONE. For the Agent single and Immediate commands, a random number is returned. This gives the attacker no indication of the validity of the guessed ID number.

5.3.3 Distributed Architecture

While the interface protocol ensures the correct binding between software processes and the hardware tasks, the coprocessor architecture ensures that the context information is kept physically separated. In addition, the architecture has a large influence on the speed in which the functions are performed.

The architecture of the proposed coprocessor is shown in Figure 5-4 and is composed of the following main parts:

- processing unit
- agents
- data interface
- processor interface

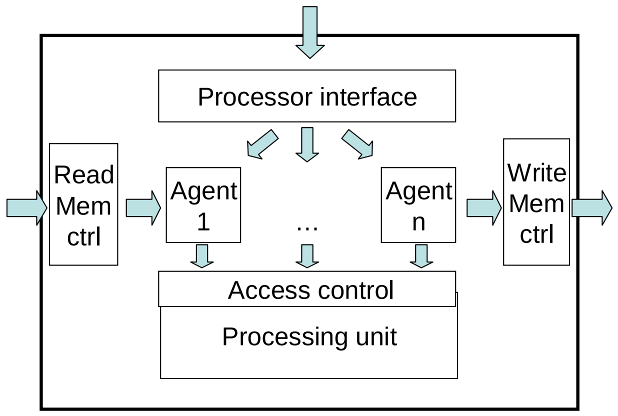

Figure 5-4 Distributed coprocessor architecture

The coprocessor processing unit contains the core logic needed to perform a specialized task. In traditional, coprocessor design, engineers have concentrated on this part to accelerate calculations. It is often large and complex; therefore, it is important that it can be used efficiently.

The agents are composed of a state machine and registers to store the context for a particular channel. It is responsible for ensuring that a function is performed correctly on a particular data block or stream in a particular context. In general, agents cycle through the following operations:

- fetch data from input port,
- perform operations on data using the datapath,
- write results to output port.

Because multiple agents exist in a single coprocessor access control logic must be used to access the datapath and interface ports. In general, the addition of agents to a coprocessor incurs very little area overhead due to its simple control operations.

The modular architecture also provides an easy and flexible design platform. The interface protocol serves to isolate the requests made by the main processor from the active components performing the computation. This allows the number of agents and processing units to vary independent of application software. This results in smaller and more portable software (multiple versions for each dynamic configuration is no longer necessary).

The coprocessor can be configured by specification of the number of agent blocks and associated processing units. The two parameters, the number of agents and the number of processing units, affect the performance of the system differently. The number of agent blocks determines the number of simultaneous processes that may be handled. The number of processing units determines the maximum throughput which the system can support. For an efficient system, these two parameters are determined based on the required throughput and latency of the processes.

The agents and processing units all operate independently from each other. The agents are isolated from the main processor through the processor interface block. The processing units are isolated from the agents through the access control block.

5.4. System Case Study

To demonstrate and study the characteristics of a system that uses the proposed secure coprocessor architecture, a secure web-server application was developed. The server uses the AES algorithm to individually encrypt data from each of its communication channels. The hash algorithms, SHA1 and SHA-256 are also implemented as coprocessors. Hash algorithms are widely used in message authentication (MAC) which ensures that messages have not been modified during transmission. The objective is not only to evaluate the benefits of the proposed secure coprocessor architecture, but also to evaluate the design tools and procedures needed to implement a secure coprocessor based system.

5.4.1 Secure Web Server Application

A secure web server application was implemented to demonstrate how a real system interacts with the proposed security coprocessor. The application performs the following tasks: When a client establishes a connection with the server, a key and encryption mode of operation is negotiated. Data is then encrypted and sent to the client. Several clients can be handled simultaneously and a process is created to manage each connection.

During the testing of the system, several connections are made to the server, each spawning its own process. Each of these processes independently registers its security context with the secure coprocessor. When several files are transferred, the coprocessor is able to handle several encryption/decryption jobs simultaneously and without leakage of information. The footprint of the different software components is shown in Table 5-2.

Table 5-2 Size of software components

Server Application	2,794 bytes
SW AES (for system w/o coprocessor)	33,536 bytes
Coprocessor interface drivers	2,928 bytes
Quickthreads library	1,868 bytes
Multithreaded TCP/IP communications stack	106,957 bytes
System calls	4,508 bytes
TOTAL	152,591 bytes

From the system point of view, the size of the software can be reduced by 20% if a coprocessor is used to perform AES encryption. This shows that for many applications, it is possible to both reduce cost and increase performance and security.

5.4.2 Distributed Architecture

To build and evaluate the application, a proprietary development environment was used. The development environment enables co-simulation of the hardware and software components of the design as well as interactions with real network traffic. The following subsections describe the different components that make up this design environment.

5.4.2.1 Co-simulation Platform

The GEZEL language was used to implement the coprocessor. GEZEL is a system design language proven very effective in the design of coprocessors [28]. In addition to easy co-simulation software processes and hardware coprocessors, in this chapter GEZEL has been extended to allow interactions with a real networking

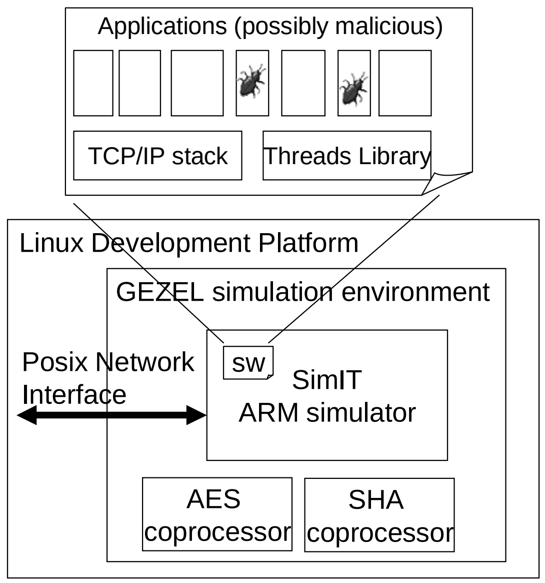

Figure 5-5 Co-simulation platform

environment. Thus using this platform, cycle true behavior of the system can be analyzed while allowing the system to be able to interact with the environment.

Figure 5-5 shows the co-simulation platform used to verify and examine the coprocessor design. The coprocessor was written in GEZEL and co-simulated with application software running on the SimIT-ARM [59] cycle-true instruction set simulator.

The software system was cross compiled and loaded onto the SimIt-ARM simulator and co-simulation was performed with the coprocessor in the GEZEL environment. To allow connections with clients outside of the simulation environment, the simulator was modified to support the POSIX select(), open(), and close() system calls. Client processes can then connect to the simulation through the TAP/TUN drivers [60]. These drivers help implement a virtual network device. This setup allows us to have external client processes connect to the server that is running in a co-simulation environment.

After verification of the design, the GEZEL tool was used to generate synthesizable VHDL. The Synplicity synthesis tools used this generated code to produce the area and speed measurements for the design.

5.4.2.2 Coprocessor Interface Drivers

Communications between a software process and the coprocessor takes place directly through the memory-mapped interface. Since context isolation is provided by the coprocessor, it is not necessary for the OS to provide special access control services to manage its use.

Commands to the coprocessor are written directly to its registers, a process that takes about 11 clock cycles per 32-bit word. This limits the maximum data throughput between the processor and coprocessor.

5.4.2.3 Threads Library

A full operating system is not necessary to demonstrate the benefits of the secure interface solution in a multitasking system. However, a threads library is necessary. A compact threads library was developed supporting a subset of the pthreads API. It was developed using QuickThreads [57], a simple and portable threads toolkit.

To illustrate a multitasking system, only the basic functions of thread initialization and context switching were implemented. The context switch flavor used is not preemptive. In the current design, a context switch occurs when a process releases a system lock.

5.4.2.4 Protocol Stack

The TCP/IP protocol stack is a suite of networking protocols necessary to communicate with the internet. The implementation of this is based on the lightweight internet protocol (lwip-0.7.1) code [58].

The stack is implemented for a multithreaded system. Each thread is in charge of a particular protocol or interface. Communications between the different network protocol threads is accomplished by a mailbox message passing system. Though the library supports all the features of an internet stack, the demonstration system built uses only a subset of this.

In the example implementation, networking functions are implemented with only two threads. At the lowest layer, an Ethernet interface is implemented to manage the basic input and output functions from the system to the network. The TCP/IP thread processes the packets and manages all the opening, closing and maintenance of connections. Applications can communicate with this stack through a sockets-like interface.

5.4.3 AES Coprocessor Architecture

The AES coprocessor was implemented according to the principles explained in Section 5.3. Using GEZEL, the coprocessor is described and linked with the SimIt-ARM simulator. It is in this environment that verification and study of the properties of the coprocessor was performed.

5.4.3.1 AES Algorithm

The AES cipher is a block cipher [49], which means that encryption and decryption operate only on fix blocks of data. In the implementation case study, 128 bit is the block size.

The algorithm consists of five main operators: AddRoundKey, SubBytes, ShiftRows, MixColumns, and KeyExpansion. The inverse of these operators are used for decryption. Figure 5-6 shows the how these operators are used to perform encryption and decryption.

Encryption	**Decryption**
AddRoundKey	AddRoundKey
For round = 1 to 9	For round = 1 to 9
SubBytes	InvShiftRows
ShiftRows	InvSubBytes
MixColumns	AddRoundKey
AddRoundKey	InvMixColumns
SubBytes	InvShiftRows

ShiftRows AddRoundKey	InvSubBytes AddRoundKey

Figure 5-6 Pseudocode for AES encryption and decryption

In both encryption and decryption algorithms, there is a FOR loop that runs through four of these steps. Hardware is efficiently realized by implementing only this group of operations (a single round) into hardware. The same hardware can then be used several times to perform a single encryption or decryption operation.

5.4.3.2 Interface Protocol

The microprocessor connects to the coprocessor through a memory-mapped interface. The interface protocol uses an instruction set designed to minimize the amount of communications necessary. Figure 5-7 shows the format of the instructions the coprocessor receives. Depending on the type of command, zero or more of the optional fields are used.

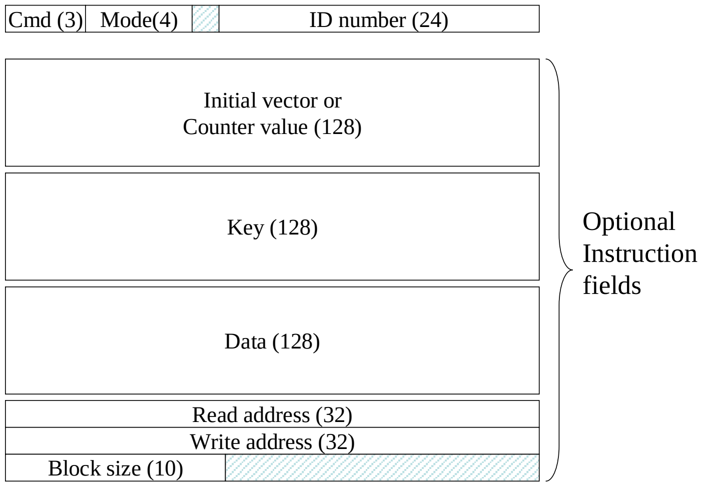

Figure 5-7 Instruction format of the AES coprocessor

The normal use of the protocol proceeds in the following manner: A process sends a command to the coprocessor to reserve some resources for future calculations. If resources are available, the coprocessor grants the request by returning a random and unique ID number. Future commands will use this ID number to reference the cryptographic context in which calculations occur. At the end of a process' life, the processor gives the command to the coprocessor to release the reserved resources.

The protocol also allows using the coprocessor without the reservation of resources. In this case, the coprocessor will return the result upon the completion of the calculation.

Table 5-3 shows the five types of commands that are available and the amount of communications needed to complete them. The commands correspond directly to the generic commands of Table 5-1.

Table 5-3 Commands accepted by the AES coprocessor

Command	Return value	32-bit words transferred (rd+wr)
Agent setup	ID number or FALSE	9+1
Agent check	DONE or FALSE	1+1
Agent release	DONE or FALSE	1+1
Agent single	AES result or FALSE	5+4
Agent continuous	DONE or FALSE	4+1
Immediate	AES result or FALSE	13+4

5.4.3.3 AES Coprocessor Architecture

Figure 5-8 shows the main logical blocks in the coprocessor. The processor interface block accepts instructions from the microprocessor through a memory-mapped interface. This block will then assign the work to one of the agent blocks. The agent blocks are dedicated controllers that are able to perform AES encryption and decryption in any mode of operation. The agent blocks each have enough registers to store the state of the calculation. The AES core performs the actual calculations. In the implementation example, this is a purely combinational block, which performs a single round of encryption or decryption; eleven rounds are necessary to perform a single AES calculation.

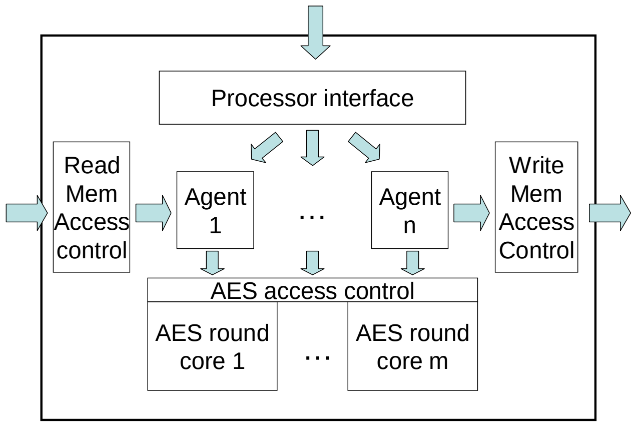

Figure 5-8   Architecture of AES multitasking coprocessor

There are several agents in the coprocessor managing multiple AES cores (each responsible for calculating a single round). The number of these elements changes depending on the throughput and latency requirements of the system. A round robin scheduling algorithm is used to ensure fair access to the AES cores. Memory read and write access control blocks are available to support the streaming or block processing architecture of Figure 5-2(b). For easy integration with popular architectures, all interfaces are 32-bit buses.

Both encryption and decryption support the following modes of operation: electronic codebook (ECB), cipher block chaining (CBC), cipher feedback (CFB), output feedback (OFB), and counter (CTR). Such flexibility enables support of a wide variety of popular security protocols including IPSec, SSH, and SSL/TLS.

5.4.3.4 Agent Blocks

The architecture of the agent block is shown in Figure 5-9. These blocks are responsible for managing calculations for a single process. The task given to an agent may be to encrypt a large data file. In this case, the agent block retrieves the data from memory, performs multiple AES calculations, and then writes the encrypted data back to memory. Because there is a tight coupling between a software process and its agent, many agent blocks exist within the multitasking coprocessor.

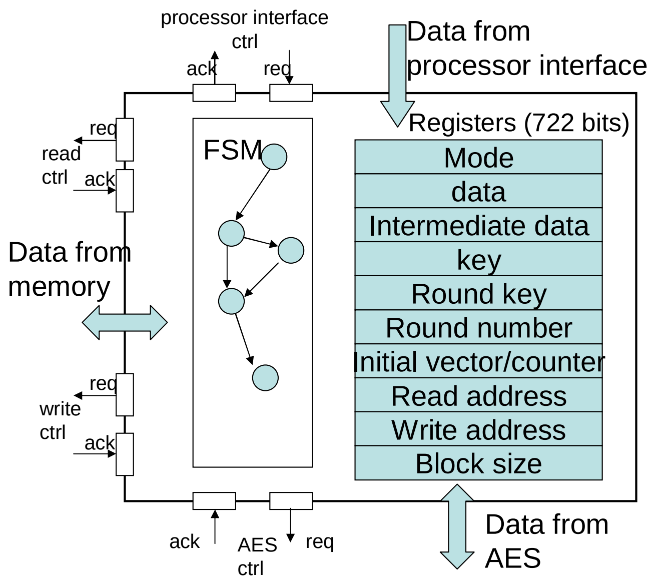

Figure 5-9 Architecture of the AES agent blocks

The agent consists of a small finite state machine with a collection of registers to remember the state of AES calculations.

The values stored in the registers add up to 722 bits of data and contribute to over 60% of the area of this block. In designs where many agents are required, area can be saved by using an AES core that performs a full AES calculation; intermediate data storage (which accounts for 30% of the total registers) will not be required in the agent. The result is a larger AES core and a system with slightly longer latencies. Instead of registers, a RAM module for each of the agents can also achieve a more area efficient but lower performance design.

5.4.3.5 Access Control

Access control blocks regulate admission to the AES core and the external memory blocks by the agents. Each of these blocks implements a round robin priority scheduler. This means that the priority of the agents to use the resources rotates each clock cycle. This ensures fairness among the agents competing to use the resources and guarantees that all calculations experience the same delays.

In the implementation, a generic and purely combinatorial AES core was implemented. The purpose was to compare the size of the multitasking processor with the size of a typical single round AES core.

5.4.4 SHA Coprocessor Architecture

In secure communications, encryption algorithms such as AES ensure that messages are kept private during transmission. Hash algorithms, in contrast, are used to ensure that the messages are not altered during transmission. The popular SHA-1 and

SHA-256 hash algorithms was implemented using the multithreaded coprocessor framework. Both algorithms are able to handle a maximum message size of 2^{64} bits and operate on 512 bit block sizes. Because of this, the effect of algorithmic properties on the coprocessor architecture can be studied.

5.4.4.1 SHA Hash Algorithm

SHA-1

```
Pad message to multiple of 512 bits
Split message into N 512-bit blocks
For message block = 1 to N
    For iteration = 1 to 80
        Do SHA-1 operation
    Accumulate intermediate hash
Return final 160-bit hash value
```

SHA-256

```
Pad Message to multiple of 512 bits
Split message into N 512-bit blocks
For message block = 1 to N
    For iteration = 1 to 64
        Do SHA-256 operation
    Accumulate intermediate hash
Return final 256-bit hash value
```

Figure 5-10 Pseudocode for the SHA hash algorithms

The SHA family of hash algorithms follows a common control structure but differ in the operations performed. Figure 5-10 shows the pseudo code for the SHA-1 and SHA-256 hash algorithms. The message to be hashed is first padded so that its size is a multiple of the block size. In the examples, the block size is 512 bits, so zeros must be added to the end of the message to extend the message size to nx512 bits. Multiple iterations of the hash algorithm are performed on each of the message blocks in message order. This operation creates an intermediate hash value that is used to compute the intermediate hash value of the next message block.

Table 5-4 Comparison of SHA-1 and SHA-256 hash algorithms

	Message size (bits)	Block size (bits)	Hash size (bits)	Security (bits)
SHA-1	$<2^{64}$	512	160	80
SHA-256	$<2^{64}$	512	256	128

The difference in the two SHA algorithms is summarized Table 5-4. The SHA1 algorithm performs 80 iterations for each message block and produces hash values of 160 bits. The SHA-256 algorithm performs 64 iterations and produces a hash value of 256 bits. Though it performs less iteration, the operations are more complex and the resulting hash value is considered more secure.

The implemented hardware core for the SHA coprocessors performs the operations found in the inner loop of the algorithm. Thus, the core is used 80 times for SHA-1 and 64 times for SHA-256 for each message block.

5.4.4.2 Interface Protocol

The microprocessor connects to the coprocessor using the memory mapped interface. Figure 5-11 shows the format of the instructions the SHA coprocessor is able to receive. Since both the SHA-1 and SHA256 algorithms take the same input, the instruction format can be reused in both cases. The optional parameter fields are used to specify the message to be hash. For messages coming from the processor, the Data field is used. For messages coming directly from memory, three Read Address and Message size registers are used.

The normal use of the protocol proceeds in the following manner: a process sends a command to the coprocessor to reserve an agent to perform a future hash operation. If an agent is available, the coprocessor reserves it and returns a random and unique ID number. The process will use this ID number to communicate the location of the messages to be hashed.

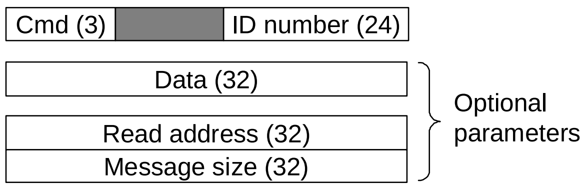

Figure 5-11 Instruction format of the SHA coprocessor

The SHA hash function has only a single mode of operation, so commands to the agent mainly involve specifying the contents of the message to be hash and/or the location in which it is located. When the calculate hash command is given, the agent will correctly pad the message and return the final has value to the process through the memory mapped interface. At this point the job is complete and the process gives the command to release the reserved resources.

Table 5-5 Commands accepted by the SHA coprocessor

Command	Description	Return value	32-bit words transferred (rd+wr)
Agent setup	Reserve and configure an agent for SHA calculation	ID number or FALSE	1+1
Agent check	Check to see if previous calculation has been completed	DONE or FALSE	1+1
Agent release	Clear a certain context from the coprocessor	DONE or FALSE	1+1
Agent single	Add a new work to the message block	AES result or FALSE	1+1
Agent continuous	Add a block of words to the message block from data taken directly from memory	DONE or FALSE	3+1
Get hash	Pad the current	AES result or	1+5 (SHA-1)

	message block and generate hash value	FALSE	1+8 (SHA-256)

Table 5-5 shows the 6 types of command that are available and the communications needed to complete them. Note that the SHA-256 has larger overhead due to the larger hash size.

5.4.4.3 SHA Coprocessor Architecture

Figure 5-12 shows the main logical blocks in the coprocessor. The processor interface block accepts instructions through a memory mapped interface and assigns work to the agent blocks. The agent blocks each have its own set of registers to store the intermediate state of hash operations. SHA core performs contains the logic that performs the actual operations. In the implementation, the core performs a single iteration of the SHA algorithm. In SHA-1, the core is invoked 80 times for each message block while in SHA-256 it is invoked only 64 times.

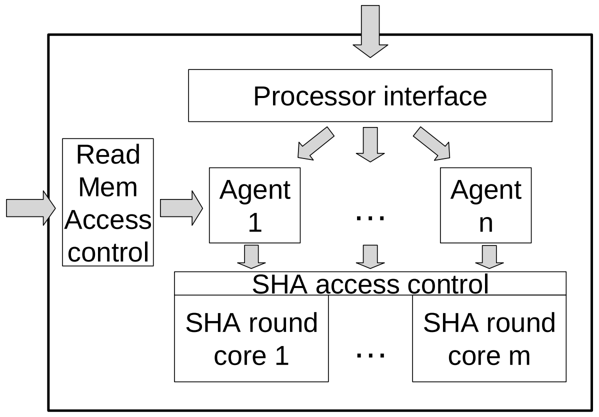

Figure 5-12 Architecture of SHA multitasking coprocessor

5.4.4.4 Agent Blocks

The architecture of the agent blocks are shown in Figure 5-13. Each block is responsible for calculating the hash value of a single message for a single process. To hash a large message, the agent is able to retrieve the data from memory, correctly pad and partition the message blocks and perform the hash functions. The final hash value is returned to the process through the memory mapped interface. There is a one to one relationship between a message to be hashed an the agent that performs the hash. Because of this, a software process may have multiple agents under its control and many agents exist within the multitasking coprocessor to service these jobs.

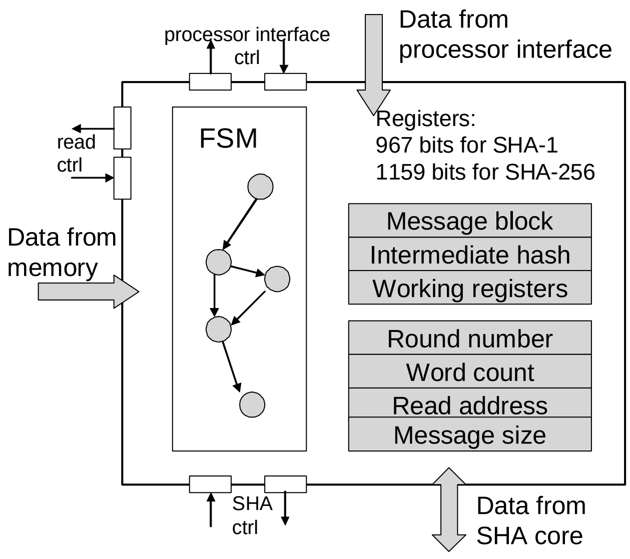

Figure 5-13 Architecture of the SHA agent blocks

The agent consists of a collection of registers that store the message block to be hashed and the intermediate hash values. Additional registers are used to store the current state of the calculation such as location of message and size of message. As with the AES coprocessor, a RAM module can significantly reduce the size of the agents.

5.4.5 System Performance

The coprocessor designs were implemented and tested in the system design environment. The following sections analyze the resulting performance and cost of the system.

5.4.5.1 Design Size and Speed

To examine the relative size of each of the modules in the design, the design was synthesized for the Virtex-II Pro FPGA using Synplicity. Table 5-6 shows the results for the AES coprocessor.

Table 5-6 Size and speed of modules in the AES coprocessor

Module	Slices	Critical path (ns)
AES core	3037	--
AES access controller	132	2.9
Agent (each unit)	1065	7.6
Read memory access controller	186	6.2
Write memory access controller	12	1.9
Microprocessor interface	623	5.8

The AES core takes three times more area than the agents. This suggests that for a system that uses many contexts, system performance can be inexpensively increased by the addition of more agents (instead of the addition of more cores). Compared to the addition of a separate coprocessor or AES core, the resulting area/performance ratio is lower.

The results for the SHA coprocessors are shown in Table 5-7 and Table 5-8 respectively.

Table 5-7 Size and speed of the modules in the SHA-1 coprocessor

Module	Slices	Critical path (ns)
SHA1 core	262	--
SHA1 access controller	86	2.0
Agent (each unit)	1090	6.7
Read memory access controller	58	1.8
Microprocessor interface	277	6.1

Table 5-8 Size and speed of the modules in the SHA-256 coprocessor

Module	Slices	Critical path (ns)
SHA256 core	269	--
SHA256 access controller	178	1.7
Agent (each unit)	1627	8.7
Read memory access controller	58	1.8
Microprocessor interface	294	5.3

The logic of the SHA hash algorithms is low in complexity and therefore the core takes much less area than the agents used to control it. The size of the agents is mainly due to the large number of registers needed to store a message block and the intermediate states of computation. In a traditional coprocessor design, these registers will be included in the size of the coprocessor core. In addition, the traditional design cannot make any claims of better security.

5.4.5.2 Coprocessor Efficiency

In internet applications such as SSL and SSH, encryption and hashing are both performed on the same data packet: encryption to protect the privacy of the communication, and hashing to protect the integrity of the message. Because of this, the same metrics are used to measure the performance of the AES function and the SHA hash functions.

In order to show the benefits of the coprocessor design at the system level, performance analysis was conducted using real world data. Studies such as [54] shows that 90% of internet traffic is under 1Kbytes. In the test scenario, a packet size of 1Kbytes is assumed. Using this assumption, the time it takes to process each of these packets can then be measured. Table 5-9 shows the results of the comparison.

Table 5-9 Comparison of overhead for bursty traffic loads

	Context switch (cycles)	AES calc (cycles)	Total Time (cycles)	Efficiency
Proposed	54	504	558	90 %
Traditional	194	504	698	72 %

This result shows that the proposed interface can handle 25% more 1Kbyte packets as the traditional coprocessor interface. This gap widens as the packet size decreases. The efficiency improvement is due to the storage of context in the agents; when a context switch occurs, this information need not be retransferred to the coprocessor. However, the actual AES core is still not running at full capacity. This suggests that further improvements are possible in the instruction set design of this type of coprocessor. Future versions should further optimize the design to increase the capacity for the bursty traffic model.

In the AES comparison (Table 5-9), the percent utilization of the processing core is compared between the proposed architecture and that of the traditional one. Both are assumed to have a similar DMA function for accessing data directly from memory. The proposed architecture showed significantly higher efficiency since context information such as secret keys, initial vectors, and modes of operation are kept in the coprocessor and do not need to be transferred at each context switch.

The SHA version of the coprocessor does not have this advantage. This is because hash functions do not need any initial parameters to operate. When compared to traditional architectures, both architectures require the transmission of the address locations where the data is located. In addition, the SHA hash algorithms have high utilization of the core – 80 iterations for SHA-1 and 64 iterations for SHA-256. Efficiency is therefore fixed at near 100% for both implementations.

5.4.5.3 Coprocessor Latency

The coprocessor is also able to encrypt several continuous data streams. This traffic pattern is common for multimedia type applications. Latency is often important in teleconferencing applications and they exhibit this type of traffic pattern. Table 5-10 shows how the latency changes as the coprocessor processes multiple data streams.

Table 5-10 Comparison of latency for different number of simultaneous streams in AES

Number of simultaneous streams	Proposed interface latency (clock cycles)	Traditional interface latency (clock cycles)
1	22	12
2	28	400
3	36	594
4	48	788

For a single stream, the traditional approach outperforms the proposed interface approach. However, for multiple data streams the multitasking coprocessor is able to scale much more gradually. In the multhreaded interface, context switching is performed in hardware at the AES round level. Consequently, the latency increases much more gradually.

This effect becomes much more serious for traditional interfaces when implemented in a network that processes both bursty packets and continuous streams. The high frequency of bursts can severely degrade the latency of the stream processes.

Note that agents are hardware objects designed to make efficient use of the computational resource, in this case, the computation of one AES round. The overall throughput of the system, however, is limited by the AES core. For systems requiring increased throughput, multiple cores must be created.

Table 5-11 Comparison of latency for different number of simultaneous streams in SHA-1

Number of simultaneous streams	Proposed interface latency (clock cycles)	Traditional interface latency (clock cycles)
1	113	113
2	160	226
3	240	339
4	320	452

Table 5-12 Comparison of latency for different number of simultaneous streams in SHA-256

Number of simultaneous streams	Proposed interface latency (clock cycles)	Traditional interface latency (clock cycles)
1	97	97
2	128	194
3	192	291
4	256	388

Hashing is usually used in conjunction with encryption/decryption to ensure that the transmitted message has not been modified. Because both operations must be performed on the transmitted or received message, latency of the operation is a very relevant metric. One of the main advantages of the proposed architecture is the fast hardware context switching that the coprocessor can perform independent of the main processor. Table 5-11 shows the comparison for SHA-1 algorithm and Table 5-12 shows the comparison for the SHA-256 algorithm. The new architecture shows latency improvements of 29% for SHA-1 and 34% for SHA256.

In the traditional interface, for each additional data stream the additional latency experienced is made up of the time it takes to transfer the new context and the time it takes to hash one hash message block. In contrast, in the proposed interface, the additional data stream only experiences the latency due to the calculation since context information is already stored in the coprocessor. For a coprocessor that contains only a single core, the additional latency for each data stream in the traditional coprocessor is 113 (23+80) cycles for SHA1 and 97 (33+64) for SHA2. The proposed interface experiences only 80 cycles and 64 cycles respectively. Further performance improvements to both systems can be obtained by the addition of a second core in the processor -- above two simultaneous streams, the additional latency due to calculation latency is halved (40 cycles for SHA1 and 32 cycles for SHA2).

5.4.6 Design Methodology

The implementation examples of the AES and SHA coprocessors show that there exist applications that can benefit from the addition of a secure coprocessor. The proposed coprocessor architecture can improve security in all multitasking systems, but the cost of that security greatly depends on the algorithm which it implements. A large and complex core which is shared by multiple small agents is the best case scenario since it demonstrates both increased performance and increased security with minimum overhead when compared with traditional coprocessor architectures.

Agents store the state of the computation in registers and they constitute the majority of the agent's area. From an implementation point of view, it would be beneficial to select (if possible) the algorithm that will give greatest amount of security while minimizing the size of the intermediate state.

Because of these costs, the proposed solution may not be appropriate for all systems and the designer should understand when it is needed. Applications that require many different communication channels or require a well used shared security functions can benefit the most from the proposed architecture.

In terms of efficiency of the coprocessor core, improvements can be made for functions that are highly programmable and require initial values or keys. In this respect, the coprocessor is suitable for encryption and decryption algorithms in general.

In terms of latency of multiple data streams, the proposed architecture will always outperform traditional architectures since context switches can be set up to run independent of the controlling processor. The rate in which the latency grows increases depending on the number of iterations required to processes one message block. For the AES algorithm, this works out to be a maximum of 12 cycles per additional data stream. For the SHA algorithms this works out to be 80 cycles for SHA-1 and 64 cycles for SHA-256.

5.5. Conclusions

Embedded systems are becoming more and more complex. Because of this the opportunities for compromising a system and the information in which it is processing is constantly increasing. Process isolation is one important component in the design process.

In terms of interface design, it is clear that secure systems have less complex driver software code running in kernel mode; this minimizes the chance that kernel code contains a security leak. In addition, though kernel mode provides some measure of protection to kernel processes, they remain vulnerable to software cache attacks and protection does not extend to external hardware such as coprocessor in the system. Because of this, the new coprocessor architecture provides process isolation protection with minimal software drivers. The functionality of the different security layers are combined in the coprocessor. Optimizations performed at this level produce an interface that is both efficient and secure. The idea is demonstrated and verified by building a secure web server which takes advantage of a secure AES coprocessor. Implementations of secure SHA coprocessors further validate the design approach.

Chapter 6

System Level Security Interface

In the previous chapter, the concept of a security interface was introduced. The design process, however, closely follows the traditional process (identify an operation and accelerating it through hardware). The security concern becomes an extra design parameter that must be considered during the optimization stage.

In this chapter, the cross-layer design process as it relates to the computing system as a whole is examined. In other words, we examine the design of an interface that provides isolation to all applications at all points of execution. To optimize this feature, the architecture of the processor itself must be reconsidered.

6.1. Computer Security

Computer architectures today are not adequate in providing security to the software programs that use it. At the base of all current systems is the concept of hierarchical security modes i.e. the core of the system is protected by layers of services with each layer further away from the hardware having more and more restrictions placed on its rights. At the very top of this stack sits the user software processes, which has the fewest rights of all the layers. This approach to designing secure systems is inherently more difficult to protect since it is usually the user processes that contain the secrets to protect.

The solution to this problem is to build a system that depends on a multilateral security model. In other words, a system must be built such that no single process has full system privileges and all processes are protected from each other. This concept of isolation must be extended all the way down to hardware such that there is no possibility of software subversion.

An ASIP design with additional instructions that implement a secure atomic context switch achieves this goal. A thread manager that sits in the processor's datapath manages the execution of all processes in the system and directly modifies the control flow to ensure secure context switching. The design presented represents only an incremental change to a general embedded processor but provides security features that fundamentally are not provided by current systems.

6.1.1 Multilateral Security

A multilateral security model distributes system rights among the processes in the system. All processes in the system will have its rights limited and none will have full system privileges. Without a concentration of power, malicious process will not have an obvious target to attack. In addition, trusted user processes will not have to trust software layers below it to protect its own secrets.

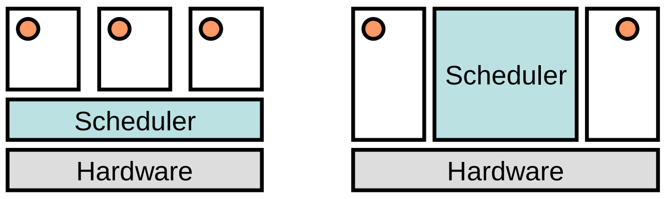

Figure 6-1 Hierarchical versus multilateral architecture

Figure 6-1 shows the difference between hierarchical and multilateral security architectures. In the hierarchical model, the software scheduler manages user processes and has full access to their memory space. Thus, the scheduler provides an obvious and easy target for a malicious process to attack. In a multilateral system, the scheduler is a process that cannot access the memory space of other processes in the system. A malicious process does not have a target that gives full access so subverting the system becomes more difficult.

In this chapter, a novel ASIP design is presented that can be easily integrated into embedded systems to support a multilateral security architecture. It accomplishes this by implementing a thread manager right within the datapath of the processor. Thread management instructions are integrated into the instruction set architecture (ISA) and operate directly on the processor's registers.

Though the security goals are unique, modification of the processor to support control flow is not unheard of. An interrupt is often handled with an automatic jump to the interrupt vector table together with an exchange of shadow registers. In digital signal processors, the repeat instruction and zero-overhead looping was added so that a particular section of code can be looped a fixed number of times without the overhead of branch instructions. Most control instructions added to ISA are purely to improve performance. While performance improvement is demonstrated in the ASIP design, *the main motivation is to design a processor that supports secure systems*.

Secure thread management is the base of any secure multilateral system. In order to enforce the unique and specific privileges of each process, the system must know which process is running at all times. In the proposed system, this responsibility is on the thread manager which is integrated into the processor's datapath. This ASIP solution provides system properties that are difficult to obtain with other architectures. These include:

- Guaranteed context switching
- Native thread management
- Strong identification of processes
- Hardware based isolation
- More distributed operating system

6.1.2 Current Security Architectures

In modern computing systems, security is built on the concept of hierarchical security modes. Even in the smallest embedded system, it is common to find processors supporting a supervisor/kernel mode and user mode. The kernel mode is usually occupied by the operating system (OS) which has all the rights in the system i.e. it has full access to all memory locations and can execute any instruction. In contrast, the user process has only limited access to memory and instructions.

To provide increased security, processors began to support more layers of security (4 layers is common in modern systems). Though many OS's still only make use of the traditional two layers, microkernel operating systems [55] took advantage of the extra layers by splitting up the monolithic OS into many different communicating processes occupying the three lower levels. The most well know microkernel OS is Minix 3 from Tanenbaum [71]. The most critical tasks such as interrupt handlers, schedulers, and timers ran in the lowest layer (ring 0). Recently, virtualization technologies have become popular as a solution to security [72]. It also adds another layer of software below the operating system. This layer, the virtual machine monitor (VMM) or hypervisor, is a thin software layer which reproduces the interface of the hardware so that multiple OS's can run on top of a single processor platform.

Virtualization technologies have become popular partly due to the claim of increased security through isolation. In fact, the popular x86 processor (a notoriously difficult processor for virtualization) is being redesigned by both Intel and AMD to better support isolation services. In the embedded space, the Denali project from the University of Washington [73], have attempted to build a VMM that is small and scalable for web server applications. Companies like VirtualLogix [74] and Trango [75] are already offering very small VMMs that are able to provide isolation services. However, though isolation is provided, it is not clear how strong it is when exposed to a determined malicious process.

Though with increasing difficulty, attacks on each of these systems exist and follow a general pattern. A malicious process enters the system and attacks the operating system to gain access to kernel mode. Once this is accomplished, it has full system privileges and can attack other processes in the system. Attacks include stealing secret information (keyloggers), subverting program execution (hijack), denial of service, etc.

In standard monolithic systems, gaining control of the OS is relatively easy – there are many interfaces with the user process and only one of the API's need be compromised to gain access to kernel mode. For microkernel type operating systems, several successive attacks need to be made at each level before the kernel mode is reached. This becomes a more difficult task, but is by no means impossible. Virtual machines provide yet another level of protection such that malicious attacks can be contained to a single OS instance. However, newer Malware is able to detect the presence of the VMM and mount attacks on them as well [76].

In the latest technology, a malicious virtual machine can be inserted below the VMM. This attack was revealed, in concept, by University of Michigan in the SubVirt project [77]. Not long afterwards, two real attacks on commercial systems were demonstrated at the Blackhat conference. The Blue Pill attack demonstrated such an attack on Windows Vista running on an AMD Pacifica processor [78]. The Vitriol rootkit-based attack was demonstrated on the Intel-VT processor running MacOS X [79].

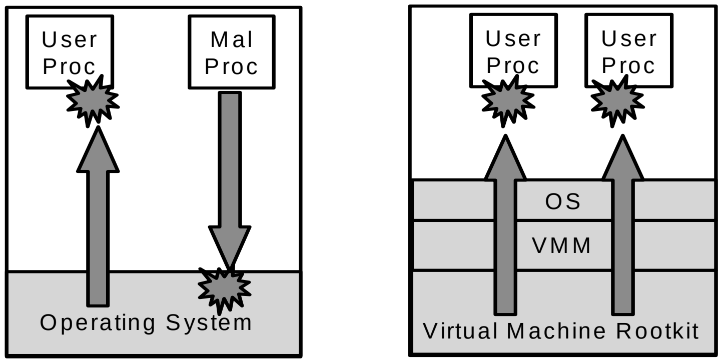

Figure 6-2 Attack vectors of malicious software

Figure 6-2 shows two different attack vectors. The traditional attack has the malicious process attack the OS to gain access to a user process. The new attack gains access to user processes by installing a malicious VMM running in the processor's kernel mode. Though the methods differ, there are two main commonalities to be noted:

In an hierarchical security architecture, all attacks are focused on attacking the software running in kernel mode

The ultimate target of the malicious attacks are other user processes, either to steal information or to subvert its normal execution.

6.2. Secure Context Switch

The secure context switch is the basis for a strong hardware supported multilateral computing system. It is able to ensure that processes are isolated from one another. In the proposed system, in order to precisely define the secure context switch, the attack model is first analyzed to determine what information is important to protect. Then operational primitives and rules are derived so that context switches can occur without compromising this secret data.

6.2.1 Attack Model

Though there is a myriad of attacks that a malicious process can perform in a system, their objectives can be summarized into three main types:

1. stealing secret information from another process
2. corruption or insertion of malicious information in another process
3. insertion of malicious execution code into another process

The stealing of secret keys is the most obvious attack that can be made. When secret information is compromised, a malicious process can gain access to bank accounts, medical records, and other personal and sensitive information. This type of attack must be prevented to maintain the property of privacy.

A malicious process is able to trick another process into operating on malicious information either by manipulation of pointers to data structures or by insertion of new parameters into another's stack. In such an attack a malicious process can trick a process into decrypting a secret message that it does not have the key to. Alternately, if a banking application is attacked, a malicious message (such as: 'transfer all funds to account X') can be inserted and then processed by an unprotected process.

A process can be hijacked by another by overwriting the return address of a function call that is stored in the stack. This allows malicious code to run with access to all the private data and privileges of the target application. This attack essentially allows the attacker to steal the identity of another process.

6.2.2 Process Identity

Processes in traditional systems are identified by process ID numbers assigned by the OS. While the system works well in the absence of malicious processes, these ID numbers can be easily spoofed. Identification of a process should be based on properties that cannot be spoofed. The identity of a process at any time can be identified by the history of what it has done and what it is doing now. To minimize implementation complexity and cost, the selection of the minimum features that can represent a process' identity was focused on. The two properties are unique to each process and cannot be spoofed and can be minimally represented by two register values: the stack pointer (SP) and the program counter (PC). The SP points to what the process has done in the past and may hold valuable secrets. The PC points to what it is doing now.

In the system, during a context switch, these two registers are replaced by the thread manager atomically. A potential malicious attacker has no opportunity to access these two values from another thread.

6.2.3 Context Switch Primitives

To protect and isolate the identities of all the processes in the system, three basic thread management primitives were defined. The description of these commands and how they are implemented is described below:

- Create()

This is the operation that registers a new process with the thread manager. For a normal system, parameters to this operation include the location of the program to run, the location of the program stack, and the parameters of the program. A secure system includes additional information such as the amount of memory needed along with the types of read or write privileges, amount of memory needed in the secret vault, and access needed to the coprocessors.

- Yield()

This operation performs the actual context switch operation i.e. it takes the current process's context information out of the processor and replaces it with the context information of the new process. A secure context switch ensures that no data from a previous context is accessible and also ensures that the two processes cannot have any direct or indirect interactions with each other during this time.

- Retire()

This operation executes when a process ends. Resources which are assigned are deallocated. The context information is taken away from the processor and replaced by the context of the next process. Like yield(), the system must ensure that no data or interactions can pass between the leaving process and the new one.

6.2.4 Fairness

The proposed computer architecture implies a distributed control structure; the operating system does not need to be involved in scheduling. Using a cooperative multithreading model, the processes in the system decide for themselves when to give up control of the processor. This system works well if all the processes are trusted and ‘fair’ with its execution time. However, this cannot be assumed in general. A malicious process may perform a denial of service attack on all the processes in the system by refusing to call **`yield`**. The effectiveness of such an attack is limited by implementing a limited cooperative multithreading model. In this system, after a context switch, the process is given a fixed amount of time to perform its calculations. When time expires, the **`yield`** instruction is automatically inserted into the processor.

For malicious processes, this forces them to give up control of the processor to other user processes – damage is thus limited. Read access to a decrementing hardware watchdog timer allows friendly processes to see how much time is left in their execution budget so that they can prepare for a context switch.

6.3. Implementation

Implementation of the ASIP involves the design of both hardware and software components. Figure 6-3 shows the architecture of the ASIP processor. The thread manager is added to the datapath. A watchdog timer is added outside the processor core to enforce time quotas. The ISA is modified to access the new thread management commands and software drivers have been developed to take advantage of these new instructions.

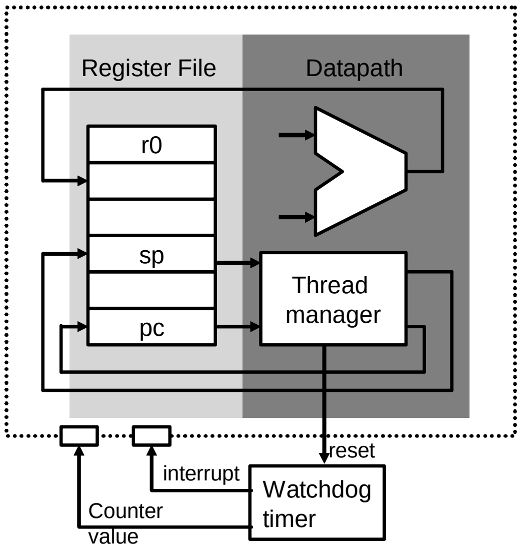

Figure 6-3 Addition of thread manager and watchdog timer to the datapath

6.3.1 Thread Manager

The thread manager is responsible for maintaining a queue of context information for all the processes in the system. Specifically, for each process in the system the thread manager stores the program counter (PC), and the stack pointer (SP). During a context switch the PC and SP registers are pushed into the queue and replaced by new PC and SP values.

The thread manager is implemented within the processor datapath and accessed through three processor instructions which correspond to the context switch primitives. In the system, it is implemented as a FIFO queue using a circular buffer architecture. Implementation cost is a RAM for storing SP and PC for all the processes in the system, and two registers that corresponds to read and write pointers. The **`create`** instruction stores the current SP and PC into the queue, therefore only the write pointer is incremented. The **`yield`** instruction both stores the old context and provides a new one, so both the write and read pointers are incremented. The **`retire`** instruction only gives out a new context so only the read pointer is incremented.

The processor instructions to the thread manager do not allow any process to find the context information of another process. The only time context information is released is through a context switch. However, since the SP and PC are atomically replaced, a malicious process has no opportunity to access this information.

6.3.2 Watchdog Timer

The watchdog timer is a decrementing counter used to enforce the maximum time allowed for each process running on the processor. It is decremented at each clock cycle and generates an interrupt when the counter reaches zero. The counter is reset by the thread manager each time a context switch occurs. The interrupt handler contains a single **`yield`** instruction which immediately switches the PC and SP registers; this forces a context switch to occur.

This forced context switch is very abrupt, and there is a chance that information can leak through the other registers to the next process. To enable processes the ability to make cleaner context switches, the watchdog timer counter value can be read through a memory mapped interface. With this information, a process is able to know how much of its allotted time is left so that it can initiate its own context switch procedure.

```
volatile long *time_left;
```

```
*time_left = (volatile long*) 0x80000004;
```

The forced context switch ensures that malicious threads cannot monopolize the processor and perform a denial of service attack.

6.3.3 Create ( )

The **`create`** instruction from the processor's ISA takes the current register values of SP and PC and stores them in the queue of the thread manager. Direct usage of this instruction essentially clones the parent process. To spawn a new process, extra instructions must be added to distinguish one from another. Note that the implicit assumption here is that the parent process is not malicious to the child process.

```
Create (child_sp, child_pc)
        mov    r0, #0
        mov    r1, #1
        mov    r4, sp
        mov    sp, child_sp
        stmfd  sp!, {r0-r12, r14}
        create
        ldmfd  sp!, {r0-r12, r14}
        cmp    r1, #1
        beq    PARENT
        mov    lr, pc
        mov    pc, child_pc
```

Figure 6-4 ARM assembly code for the create() operation

Figure 6-4 shows the ARM assembly commands that are used to create a new process. Before Create() is called, memory must be allocated for the stack of the new process and the entry address to the new function must be known.

In the function, before **`create`** is called, a flag (`r1`) is set to indicate the parent process. The address of the child_sp is then stored in the SP register. All registers are then pushed onto the new stack and **`create`** is called. This is the point where the new process will start its execution.

After **`create`**, the values in the stack are popped and the flag is checked. A set flag indicates that the current process is the parent. In this case the flag is reset and all registers are pushed back onto the stack. The original stack pointer (from the parent) is returned to the SP register and execution continues.

In the case of the child process, the flag has already been modified by the parent (it executes first) and will be reset. This indicates to the child process that it can now branch to the process entry function.

6.3.4 Yield ( )

The **`yield`** instruction takes the SP and PC register values and replaces them with new ones. Since these two values wholly represent the identity of a process, the new process is able to run immediately. However, there is a possibility that information leaks between the two processes through the general purpose registers. A clean context switch requires extra instructions.

```
stmfd sp!, {r0-r12, r14}
mov r0, #0
 . . . (clear all registers)
mov r12, #0
mov r14, #0
yield
ldmfd sp!, {r0-r12, r14}
```

Figure 6-5 ARM assembly code for the yield() operation

Figure 6-5 shows the ARM assembly commands that can implement a clean context switch. All the general purpose registers are first pushed into the stack. Then the values of all the registers are cleared before **`yield`** is call. This ensures that the next process will not be able to gain any extra information through leftover values in the registers.

6.3.5 Retire ( )

Like the **`yield`** instruction, **`retire`** invokes a context switch by replacing the SP and PC. In addition to clearing all the registers, the memory allocated to the stack should be freed as well.

```
mov r0,sp
bl  free
mov r0, #0
mov r1, #0
 . . . (clear all registers)
mov r12, #0
```

Figure 6-6 ARM assembly code for the retire() operation

Figure 6-6 show the ARM machine code that is run before a **`retire`** is called. The memory allocated to the stack is first deallocated. All the general purpose registers are then cleared to zero.

A special case occurs when the current process is the last or only process in the system. **`Retire`** would normally replace the SP and PC with new values from the next process. If there is no other process in the system, then the PC is set to a special address in non-writeable memory. This address location contains an infinite loop to a NOP instruction.

6.4. Results

The GEZEL language was used to implement the ASIP datapath modifications. GEZEL is a system design language proven very effective in the design and simulation of domain-specific micro-architectures [28]. It is also a design environment that enables easy co-simulation between user described logic and a wide variety of intellectual property.

The thread manager and watchdog timer was written in GEZEL and co-simulated with software running on the Simit-ARM [36] cycle-true instruction set simulator. After verification of the design, the GEZEL tool was used to generate synthesizable VHDL. Synopsys synthesis tools used this generated code to produce the area and speed measurements for the design.

Table 6-1 Comparison of speed and area of datapath modifications

Module	**Size (Kgates)**	**Critical path (ns)**

N-ARM7TM	50	17
Thread manager	1	7

To examine the relative size of the thread manager, circuit synthesis was performed using Synopsis. The TSMC 0.18um CMOS standard cell library with conservative wire load model was used. Table 6-1 shows the results. The results were compared to the synthesized N_ARM7TM design from [37]. The additional memory needed for the memory depends on the maximum number threads that the system is designed to support and requires 64 bits per thread. Not including the two port memory of the thread queue, the thread management unit only takes 2% of the total ARM processor area.

Table 6-2 Comparison of execution time for thread management operations

Module	**SW (cycles)**	**ASIP (cycles)**
Create()	743	450
Retire()	348	53
Yield()	191	63
Check_watchdog()	---	27

Test software was run on an ARM simulator to compare the performance between a software thread manager and the ASIP implementation. Table 6-2 shows the speed comparison between the purely software based context switch and the native thread management instructions of the ASIP. The software threads library was developed using QuickThreads [57], a simple and portable threads toolkit.

In a multitasking system, the yield() command is used most often. The results show that the ASIP solution is 3 times faster than the equivalent software routine. The retire() operation is almost 7 times faster. The check_watchdog() function has no equivalent in software. It is implemented with a memory mapped interface. Reading the 32-bit value from the watchdog time takes only 27 cycles.

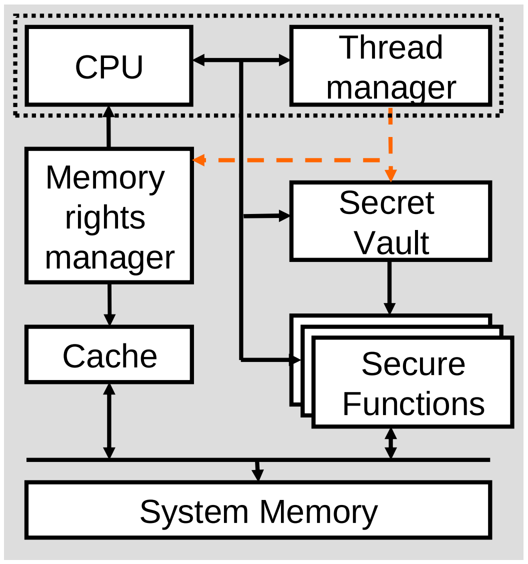

Figure 6-7 Secure embedded system based on ASIP

6.5. Discussion

By removing hierarchical software layers, isolation of the software processes in the system becomes the responsibility of the hardware platform. The context switch defines the border between two processes in the system and must be secured so that the boundary is not vulnerable to information leaks or malicious attacks. This chapter demonstrates that with minor modifications to a processor core, an ASIP can be built that will support secure context switching. There are, however, other parts of the system, such as the memory, where attacks can occur.

While the proposed ASIP processor currently does not support these functions, it is an integral component a secure system that does. Figure 6-7 shows an example of such a system.

Connected to the ASIP core are two special modules: the memory rights manager and the secret vault. The memory rights manager sits between the processor core and the memory subsystem. It determines which memory locations the processor has access to. Also, access to sections of memory can be defined as read only, write only, or both read and write. The secret vault is a small memory that is used to store keys and other sensitive information that will be used by security coprocessors in the system.

The role of the thread manager in this system is to broadcast to the external modules the identity of the currently running thread. For the memory subsystem, this means that each software process can be assigned unique, non-overlapping blocks of memory. For processes that need to communicate with each other, shared memory locations can be negotiated. The memory access rights of a software process cannot be compromised since the process does not have direct access to the memory manager – the settings are determined by the currently running process.

From the point of view of the processor, the secure vault is a small private write only memory. The secret vault output is connected directly to external secure coprocessor units like ECC or AES engines. Since it knows the identity of the currently running process through the thread manager, access to the secret information from the different processes in the system can be regulated. Thus the secret keys can only be used by the corresponding software process. For increased security, once the key is programmed, even the user process is not able to access its value. This prevents possible side channels due to insecure programming practices.

Thus the thread manager in the ASIP forms the basis of a strong and secure system. Similar architectures can be developed to protect access to I/O devices, networking services, etc.

6.6. Conclusions

There is much interest in adding security to computing systems and the technologies from general purpose computers are now trickling down to the embedded domain. Yet, all the technologies are based on hierarchical security architectures. Processors built using this architecture are focused on protecting operating systems instead of user processes where the root of trust resides.

The secure context switch is the basis of multilateral security architecture. In this chapter, an ASIP architecture that supports secure context switch is proposed. The addition of a thread manager to the datapath cost less than 2% in area of the ARM7 processor. In addition to providing a more secure platform, this addition is also able to increase the performance of context switch functions by more than 3 times. The proposed architecture can be used in the future to secure not only the context switch, but also provide user process centered rights to all the different subsystems external to the processor core.

Chapter 7

Conclusion

In traditional embedded system design, the application is divided into independent blocks and implemented separately. Interfaces allow the different components to interact together to produce a functioning system. While this strategy produces systems that work, the interfaces between the modules introduce performance overheads. In the design of embedded systems, performance is often a key design criterion. It is, therefore, important to have a proven methodology to accelerate applications in embedded systems.

The main contribution of this dissertation is the development of a cross-layer co-design methodology to accelerate algorithms. It is able to produce high performance embedded systems and is applicable to applications that vary in size and function. The process consists of the following three steps:

1. Remove intermediate interfaces to consolidate processing

Intermediate interfaces incur performance overhead and are not functionally necessary for correct system operation. Removing them reduces interface overhead even before algorithmic optimizations are performed.

2. Optimize algorithm

Depending on the application domain, there are many optimization techniques that can be used. The removal of intermediate interfaces exposes more opportunities to make use of them. Given the existence of interfaces within the design, however, optimization of the algorithm also includes reducing the use of the interfaces and to organize the use of the interfaces efficiently.

3. Optimize the interfaces

For the interfaces that still exist in the system, performance can be further increased by changing the interface mechanism. One such technique for this is the saving of local context information so that computation parameters need not be repeated transferred. Another method is to separate the data flow and control flow components of a calculation; DMA is an implementation example of such a technique.

Because of the broad applicability of the methodology and its novel approach, its use in design also enables the discovery of new technologies and algorithms. In this dissertation, significant contributions have also been made in the areas of channel coding, signal processing architectures, optical networking, system design tools, and security architectures.

In the first part of the dissertation, the methodology was used to design a new type of optical network that is able to guarantee high data throughput. Following the co-design process, two key decisions allowed the achievement of the new and unique design: (1) the choice of an uncoordinated system immediately removed the complexities of a media access control module (2) the choice to consolidate all processing in digital hardware. Thus the methodology enabled the design of new channel codes and the corresponding high speed hardware structures that implement them. These results represent significant contributions to the areas of communication channel codes, signal processing, and optical networks.

The application of the methodology in the domain of system on chip (SOC) is discussed in the second part of the dissertation. In the domain of SOC, the designer is often limited by an underlying computer architecture. Assumption of this model places many restrictions on the interfaces. For example, the system bus bandwidth inherently restricts the speed in which information can be transferred through it. Using this methodology, we show how algorithms can be optimized to take advantage of the interfaces available. The contribution of this part is a set of tools that is able to support the design methodology from high level exploration stage all the way down to implementation.

The third part of the dissertation examines how the methodology can be used to efficiently support greater security in embedded systems. The resulting computer architecture designs support better security in embedded systems and, in addition, improve system performance. This result is a significant contribution to the field of secure computer architectures.

Bibliography

[1] Y. Matsuoka, P. Schaumont, K. Tiri, and I. Verbauwhede, “Java Cryptography on KVM and its Performance and Security Optimization using HW/SW Co-design Techniques,” CASES, 2004.

[2] A. P. Foong, T. R. Huff, H. H. Hum, J. P. Patwardhan, and G. J. Regnier, “TCP Performance Re-Visited,” IEEE 2003.

[3] J.S. Chase, A. J. Gallatin, and K. G. Yocum, “End System Optimizations for High-Speed TCP,” IEEE Communications Magazine, April 2001.

[4] A. Romanow and S. Bailey, “An Overview of RDMA over IP”, First International Workshop on Protocols for Fast Long-Distance Networks, 2003.

[5] IEEE 802.3-2005 “Part 3: CSMA/CD access method and physical layer specifications”, 2005.

[6] IEEE Std 802.3an™-2006 "Amendment 1: Physical Layer and Management Parameters for 10 Gb/s Operation", Sept. 2006.

[7] F.E. Ross and J.R. Hamstra, “Forging FDDI, IEEE Journal on Selected Areas in Communications, vol. 11, issue 2, Feb. 1993, pp. 181-190.

[8] R. Bosch, GmbH, CAN specification 2.0 (1991)

[9] D. R. Boggs, J.C. Mogul, and C.A. Kent, “Measured Capacity of an Ethernet: Myths and Reality,” ACM SIGCOMM ’88 Symposium on Communications Architectures and Protocols, pp. 222-234.

[10] M. Griot, A. Vila Casado and R. Wesel, “Nonlinear Turbo Codes for Interleaver-Division Multiple-Access on the OR Channel,” IEEE GlobeCom’06, December 2006.

[11] M. Griot, A. Vila Casado, W.Y. Weng, H. Chan, J. Basak, E. Yablonovitch, I. Verbauwhede, B. Jalali and R. Wesel, “Trellis Codes with Low Ones Density for the OR Multiple Access Channel,” IEEE International Symposium on Info. Theory, July 2006.

[12] L. Ping, K. Y. Wu, and L. Liu, “A Simple, Unified Approach to Nearly Optimal Multiuser Detection and Space-time Coding,” ITW 2002, India, October 2002.

[13] L. Ping, L. Liu, and W. K. Leung, “A Simple Approach to Near-Optimal Multiuser Detection: Interleaver-Division Multiple Access,” IEEE Wireless Communications and Networking Conference, pp. 391-396, 2003.

[14] G. Ungerboeck, “Channel Coding with Multilevel Phase Signals,” IEEE Transactions on Information Theory, volume 28, pp. 52-67, 1982.

[15] Virtex-II Pro Platform FPGA Handbook (UG012) found at http://www.xilinx.com/bvdocs/userguides/ug012.pdf

[16] Y. Zhu and M. Benaissa, "A Novel ACS Scheme for Area Efficient Viterbi Decoders," Proceedings of the 2003 International Symposium on Circuits and Systems, May 2003, pp. 264-267.

[17] M. Guo, O. Ahmad, M. Swamy, and C. Wang, "A Low Power Systolic Array Based Adaptive Viterbi Decoder and its FPGA Implementation," Proceedings of the 2003 International Symposium on Circuits and Systems, May 2003, pp. 276-279.

[18] A. Abdul Shakoor, V. Szwarc, and T. Kwasniewski, "High Speed Viterbi Decoder for W-LAN and Broadband Applications," 2nd Annual IEEE Northeast Workshop on Circuits and Systems, June 2004, pp. 25-28.

[19] D. Knuth, *The Art of Computer Programming Vol. 3 Sorting and Searching*, Addison-Wesley, Reading, MA, 1973, pp.229-232.

[20] I. Pupeza, A. Kavcic, and L. Ping, "Efficient Generation of Interleavers for IDMA," IEEEE International Conference on Communications, June 2006.

[21] based on code found at http://www.humanistic.org/~hendrik/

[22] T. Ohara, H. Takara, T. Yamamoto, H. Masuda, T. Morioka, M. Abe and H. Takahashi, "Over-1000-channel Ultradense WDM transmission with Supercontinuum Multicarrier Source", J. Lightwave Tech., Vol. 24, No. 6, pp 2311-2317, June 2006.

[23] A. C. Bordonalli, C. Walton, and A. J. Seeds, "High-Performance Phase Locking of Wide Linewidth Semiconductor Lasers by Combined Use of Optical Injection

Locking and Optical Phase-Lock Loop", *J. Lightwave Technol*. vol. 17, no. 2, pp. 328-342, Feb. 1999.

[24] K. Petermann, *Laser Diode Modulation and Noise,* Kluwer Academic Publishers, 1988.

[25] B. Saleh and M. Teich, *Fundamentals of Photonics*, John Wiley & Sons, 1991.

[26] E. L. Wooten, K. M. Kissa, A. Yi-Yan, E. J. Murphy, D. A. Lafaw, P. F. Hallemeier, D. Maack, D. V. Attanasio, D. J. Fritz, G. J. McBrien, and D. E. Bossi, "A Review of Lithium Niobate Modulators for Fiber-Optic Communication Systems", IEEE J. Selected Topics in Quantum Elect., Vol. 6, No 1, pp 69-81, 2000.

[27] E. I. Ackerman and C. H. Cox, "Effect of Pilot tone Based Modulator Bias control on External Modulation Link Performance", MWP 2000, pp 121-124, Tu4.6, 11-13 Sept. 2000. J. Hennessy, D. Patterson, "Computer Architectures: A quantitative approach, 2nd Edition" Ch. 5.3, MKP Publishers, 2002.

[28] P. Schaumont and I. Verbauwhede, "A Component-based Design Environment for Electronic System-level Design," IEEE Design and Test of Computers, special issue on Electronic System-Level Design, 23(5), pp. 338-347, September-October 2006.

[29] F. Ghenassia (ed), "Transaction Level Modeling with SystemC TLM Concepts and Applications for Embedded Systems," Springer, 2005.

[30] D. Gajski et al., “SpecC Specificatin Language and Methodology,” Kluwer Academic, 2000.

[31] XtensaTM microprocessor. Tensilica Inc. (http://www.tensilica.com).

[32] G. Martin, “Recent Developments in Configurable and Extensible Processors,” International Conference on Application-specific Systems, Architectures and Processors, Sept 2006, p. 39-44.

[33] D. Ching, P. Schaumont, and I. Verbauwhede, "Integrated modeling and generation of a reconfigurable network-on-chip," Int. J. Embedded Systems, Vol. 1, Nos. 3/4, 2005, p. 218-227.

[34] A. Hodjat and I. Verbauwhede, "Interfacing a high speed crypto accelerator to an embedded CPU," Proc. 38th Asilomar Conference on Signals, Systems, and Computers, Volume 1, pp. 488-492, November 2004.

[35] A. Hodjat, L. Batina, D. Hwang, I. Verbauwhede, "A Hyperelliptic Curve Cryto Coprocessor for an 8051 Microcontroller,” IEEE Workshop on Signal Processing Systems (SIPS 2005), pp.93-98, November 2005.

[36] W. Qin, S. Rajagopalan, S. Malik, “A Formal Concurrency Model Based Architecture Description Language for Synthesis of Software Development Tools,” ACM 2004 Conference on Languages, Compilers, and Tools for Embedded Systems, June 2004, pp. 47-56.

[37] I-J Huang, W-K Huang, R-T Gu, and C-F Kao, “A cost effective multimedia extension to ARM7 microprocessors,” IEEE International Symposium on Circuits and Systems, 2002.

[38] P. R. Panda, F. Catthoor, N. D. Dutt, K. Danckaert, E. Brockmeyer and A. Vandercappelle, “Data and Memory Optimization Techniques for Embedded Systems”. ACM Transactions on Design Automation of Electronic Systems, Vol 6, No. 2, pp.149-206, April 2001.

[39] R.W. Hartenstein, J. Becker, M. Herz, and U. Nageldinger, “A novel sequencer hardware for application specific computing”. IEEE International Conference on Application-Specific Systems, Architectures and Processors, 1997.

[40] M. Herz, R. Hartenstein, M. Miranda, E. Brockmeyer, and F. Catthoor, “Memory addressing organization for stream-based reconfigurable computing”, 9th International Conference on Electronics, Circuits and Systems, 2002.

[41] http://www.xilinx.com/univ/XUPV2P/Documentation/XUPV2P_User_Guide.zip

[42] http://www.xilinx.com/ise/embedded/edk91i_docs/edk_ctt.pdf

[43] K. Weber, “The accelerated integer GCD algorithm,” ACM Transactions on Mathematical Software, 1995.

[44] D. Knuth*, The Art of Computer Programming, Volume 2: Seminumerical Algorithms (3rd edition),* Reading, Massachusetts: Addison-Wesley, 1997.

[45] S. Chatterjee, A. R. Lebeck, P.K. Patnala, and M. Thottethodi, “Recursive array layouts and fast matrix multiplications,” IEEE Transactions on Parallel and Distributed Systems, Nov 2002.

[46] U.J. Kapasi, S. Rixner, W.J. Dally, B. Khailany, J.H. Ahn, P. Mattson, and J.D. Owens, “Programmable Stream Processors,” Computer, August 2003.

[47] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata, “Cell Broadband Engine Architecture and its first implementation – A performance view”, IBM Journal of Research and Development, 2007-08-14. http://www.research.ibm.com/journal/rd/515/chen.html

[48] J.A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the Cell multiprocessor", IBM Journal of Research and Development, 2005-09-07. http://researchweb.watson.ibm.com/journal/rd/494/kahle.html

[49] National Institute of Standards and Technology (U.S.), Advanced Encryption Standard. http://csrc.nist.gov/publication/drafts/dfips-AES.pdf

[50] P-H Kamp and R. Watson, “Building Systems to Be Shared, Securely,” ACM Queue, vol. 2, issue 5, pp. 42-51, July/August 2004.

[51] S.J. Vaughan-Nichols, “How trustworthy is trusted computing?” Computer, vol. 35, issue 3, pp.18-20, March 2003.

[52] R. Oppliger and R. Rytz, “Does trusted computing remedy computer security problems?” IEEE Security and Privacy Magazine, vol.3, issue 2, pp.16-19, March/April 2005.

[53] http://www.trusted-logic.com/mob_tech.html

[54] N. Brownlee and K. Claffy, “Internet stream size distributions,” Proc. of 2002 ACM SIGMETRICS international conference on Measurement and Modeling of Computer Systems, pp.282-283, June 2002.

[55] Tanenbaum, J. Herder, and Herbert Bos, “Can We Make Operating Systems Reliable and Secure?” Computer, vol.39, issue 5, pp.44-51, May 2006.

[56] T. Garfinkel and M. Rosenblum, “When Virtual is Harder than Real: Security Challenges in Virtual Machine Based Computing Environments,” Proceedings of the 10th Workshop on Hot Topics in Operating Systems (HOTOS-X), May 2005..

[57] Quickthreads, http://www.cs.washington.edu/research/compiler/papers.d/quickthreads.html

[58] Lwip protocol stack, http://savannah.nongnu.org/projects/lwip/

[59] W. Qin, and S. Malik, “Flexible and Formal Modeling of Microprocessors with Application to Retargetable Simulation,” Proc. of DATE 2003, pp. 556-561, March 2003.

[60] TAP/TUN universal interface, http://vtun.sourceforge.net/tun/

[61] http://www.arm.com/products/esd/trustzone_home.html

[62] I. Hiroaki, A. Ikeno, M. Kondo, J. Sakai, and M. Edahiro, “VIRTUS: A New Processor Virtualization Architecture for Security Oriented Next-Generation Mobile Terminals,” Proc. 2006 Design Automation Conference (DAC 2006), pp.484-489, July 2006.

[63] K. Shimizu, S. Nusser, W. Plouffe, V. Zbarsky, M. Sakamoto, and M. Murase, “Cell Broadband Engine processor security architecture and digital content protection,” Proceedings of the 4th ACM international workshop on Contents protection and security, October 2006.

[64] Trusted Computing Group, https://www.trustedcomputinggroup.org/home

[65] W. E. Kuhnhauser, “Root Kits: an operating systems viewpoint,” ACM SIGOPS Operating Systems Review, vol.38, issue 1, pp.12-23, January 2004.

[66] D. Page, “Partitioned Cache Architecture as a side Channel Defense Mechanism", Cryptology ePrint Archive, Report 2005/280, 2005.

[67] D. A. Osvik, A. Shamir and E. Tromer, “Cache attacks and Countermeasures: the Case of AES", Cryptology ePrint Archive, Report 2005/271, 2005.

[68] C. Percival, “Cache missing for fun and profit", Proc. of BSDCan 2005, Ottawa, manuscript available from http://www.daemonology.net/hyperthreading-considered-harmful

[69] D. Page, “Defending Against Cache Based side Channel Attacks", Information Security Technical Report 8(1):30-44, 2003.

[70] M. Gasser, Building a Secure Computer System. Van Nostrand Reinhold, May 1988.

[71] Minix 3 OS, http://www.minix3.org/

[72] M. Rosenblum, and T. Garfinkel, "Virtual machine monitors: current technology and future trends," Computer, vol.38, issue 5, pp.39-47, May 2005.

[73] A. Whitaker, M. Shaw, and S. Gribble, "Denali: A Scalable Isolation Kernel," *Proceedings of the Tenth ACM SIGOPS European Workshop*, Saint-Emilion, France, September 2002.

[74] http://www.virtuallogix.com/

[75] http://www.trango-systems.com/

[76] I. Arce, "Ghost in the Virtual Machine," IEEE Security and Privacy Magazine, July-Aug 2007.

[77] Samuel T. King, Peter M. Chen, Yi-Min Wang, Chad Verbowski, Helen J. Wang, Jacob R. Lorch, "SubVirt: Implementing malware with virtual machines", *Proceedings of the 2006 IEEE Symposium on Security and Privacy* , May 2006.

[78] J. Rutkowska, "Subverting VistaTM Kernel for Fun and Profit", BlackHat Briefings USA, August 2006, Las Vegas, NV.

[79] D. A. Dai Zovi, "Hardware Virtualization Rootkits," BlackHat Briefings USA, August 2006, Las Vegas, NV.

[80] R. York, “A New Foundation for CPU Systems Security: Security Extensions to the ARM Architecture”, ARM Limited, May 2003.

[81] K. Prettyjohns, “Hard Real-Time Microcontroller for Embedded Applications,” Embedded World, February 2007.

[82] Ubicom Inc., “The Ubicom IP3023 Wireless Network Processor,” 2003.

[83] K. Shimizu, S. Nusser, W. Plouffe, V. Zbarsky, M. Sakamoto, and M. Murase, “The Cell Broadband Engine processor security architecture and digital content protection,” Proceedings of the 4th ACM international workshop on contents protection and security (MCPS06), October 2006.

[84] H. Chan, P. Schaumont, and I. Verbauwhede, “Process Isolation for Reconfigurable Hardware,” 2006 International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA06) (Distinguished Paper), pp. 164-170, June 2006.