ARM-Based SoC Design And Emulation of A Compressed Bit-Vector Packet Classification Algorithm

by

Clinton Stuart

A Thesis submitted to the Faculty of Graduate Studies of

The University of Manitoba

in partial fulfillment of the requirements of the degree of

MASTER OF SCIENCE

Department of Electrical and Computer Engineering

University of Manitoba

Winnipeg, Manitoba

Thesis Advisor: R. D. McLeod

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES

*****

COPYRIGHT PERMISSION

ARM-Based SoC Design

And Emulation of a Compressed Bit-Vector

Packet Classification Algorithm

BY

Clinton Stuart

A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University of

Manitoba in partial fulfillment of the requirement of the degree

MASTER OF SCIENCE

Abstract

Packet classification (PC) is the problem of matching incoming packets to a router against a database of rules or filters. The rules specify a directive for incoming packets and provide a means of implementing new services such as *Quality of Service* (QoS) guarantees. This thesis presents state-of–the-art developments and emerging approaches to packet classification along with a set of requirements to meet the demands of existing and future large bandwidth connections. Although many schemes have been proposed to solve the multi-dimensional PC problem, none of them scale well beyond two dimensions in terms of speed and/or rule size. Based on this evaluation, a novel algorithm is proposed.

The new and innovative algorithm builds upon two geometric hardware-based approaches, namely the Lucent *Bit-Vector* (BV) and *Aggregated Bit-Vector* (ABV) algorithms [1], [2]. Two new ideas are presented, creating what shall be called the *Compressed Bit-Vector* (CBV) algorithm. This CBV scheme employs a divide-and-conquer technique breaking the general multi-dimensional problem into more manageable two-dimensional problems each producing compressed bit-vector solutions. This provides a framework that will allow for the incorporation of various two-dimensional search algorithms into one system that can perform multi-dimensional classification. Similar to the ABV and Lucent BV schemes, the CBV algorithm is amenable to a hardware and software implementation.

As a proof-of-concept, a portion of the algorithm is implemented and functionally verified on a *rapid prototyping platform* (RPP) by adhering to a *System-on-Chip* (SoC) design flow. Through this process, it is further demonstrated how platform-based design provides many benefits without a commitment to silicon. It allows for embedded software development early in the design cycle, the resolution of hardware and software integration problems, the ability to validate third party IP blocks in a system environment, and the capability to experiment with the implementation. It is also useful for acquiring realistic performance statistics to determine if *application-specific integrated circuit* (ASIC) performance targets are achievable. Using the platform, an extensive performance analysis of the CBV algorithm is achieved on synthetic rule-sets modeled after real firewall database statistics. Out of this study, it is shown that the algorithm's memory usage is scalable to large rule-sets. Recommendations are also made on how an ASIC implementation could sustain a *Gigabit Ethernet* (GE) line-rate on average size packets.

Acknowledgements

Numerous individuals and organizations were involved in this research and merit acknowledgement. To Dr. R. D. McLeod, my M. Sc. advisor from the University of Manitoba, I would like to express my sincere gratitude for his patience and guidance on this project. His involvement in the *System-On-Chip Research Network* (SOCRN) proved very helpful in obtaining resources and support for this project.

I would also like to express thanks to TRLabs and industry sponsor PMC-Sierra for providing and showing interest in this topic. TRLabs provided funding, an excellent working environment and also partnered with the University of Manitoba to supply tools for the FPGA development. I would also like to thank the *National Sciences and Engineering Research Council* (NSERC) and the Department of Public Safety and Emergency Preparedness for supporting this research, their interest in cyber security and ensuring the safety of our national infrastructure.

I am also grateful to the *Canadian Microelectronics Corporation* (CMC) and in particular Hugh Pollitt-Smith for providing the rapid prototyping platform, RLDRAM module and the RLDRAM controller netlist used to develop a working representation of the packet classifier. Furthermore, I would also like to thank IDERS and Vansco Electronics for guidance in layout and soldering the fine pitch surface mount connectors of the custom interface board.

Foremost, I would like to acknowledge my colleague Doug Cornelsen for his contributions, feedback and many ideas, which have contributed to the improvement of the packet classifier design. Doug shared in creating the design specifications and in software-hardware integration. However, his primary contribution was implementing the software tasks of the packet classifier design and rule generator. Doug also provided insightful discussions while implementing the hardware portion of the design.

Finally, I would like to thank my family and friends for their support during my Master's research.

Dedication

Behind any notable achievement, there is often a source of encouragement and continual support. For this thesis, that source has been my parents, Linda and Larry, and my darling Clarivel. It is to them that I dedicate this thesis.

To my parents, I am extremely grateful. They have instilled in me a strong work ethic and supported me in all of my academic pursuits.

To Clarivel, I realize many evenings were sacrificed while completing this thesis and working full-time. This was quite a test, but I thank you for your patience and unwaivering support, particularly when pulling this thesis together wore on in the final stages.

Table of Contents

List of Figures

List of Tables

List of Terms and Abbreviations

Acronym	Definition
ABV	Aggregated Bit-Vector
AHB	Advanced High-performance Bus
ALE	Address Latch Enable
AMBA	Advanced Microcontroller Bus Architecture
APB	Advanced Peripheral Bus
ARM	Advanced RISC Machine
ASB	Advanced System Bus
ASIC	Application-Specific Integrated Circuit
ATM	Asynchronous Transfer Mode
BFM	Bus Functional Model
BUFG	Global Clock Buffer
BV	Bit-Vector
CAM	Content Addressable Memory
CBV	Compressed Bit Vector
CDF	Cumulative Distribution Function
CIDR	Classless Inter-Domain Routing
CMC	Canadian Microelectronics Corporation
CPU	Central Processing Unit
CSR	Control Status Register
DC	Direct Current
DCI	Digitally Controlled Impedance
DCM	Digital Clock Manager
DDR	Double-Data-Rate
DIMM	Dual In-line Memory Module
DIP	Destination Internet Protocol Address
DNS	Domain Name System
DP	Destination Port
DPRAM	Dual-Port Random-Access Memory

Acronym	Definition
DRAM	Dynamic Random-Access Memory
EBI	External Bus Interface
EDIF	Electronic Data Interchange Format
FIFO	First In First Out
FIQ	Fast Interrupt Request
FPGA	Field-Programmable Gate Array
FTP	File Transfer Protocol
Gbps	Gigabits Per Second (10^9)
GBps	GigaBytes Per Second
GE	Gigabit Ethernet
GEM	Geometric Efficient Matching
GOT	Grid-of-Tries
GPIO	General Purpose Input/Output
GUI	Graphical User Interface
HDL	Hardware Description Language
HiCuts	Hierarchical Intelligent Cuttings
HW	Hardware
HTTP	HyperText Transfer Protocol
IANA	Internet Assigned Numbers Authority
IB	Inbound
IBUFG	Global Clock Input Buffer
IC	Integrated Circuit
ICE	In-Circuit Emulator
ICMP	Internet Control Message Protocol
Integrator/AP	Integrator ASIC Development Platform
IOB	Input/Output Block
IP	Intellectual Property, Internet Protocol
IPL	Internet Protocol Lookup
IRQ	Interrupt Request
ITC	Information Technology Communication
K	kilo binary (2^10 or 1024)

Acronym	Definition
KMI	Keyboard and Mouse Interface
LVDS	Low Voltage Differential Signaling
LED	Light Emitting Diode
LOC	Location constraints
LT	Logic Tile
JTAG	Joint Test Action Group
Mpps	Million Packets Per Second
NAT	Network Address Translation
NSERC	National Sciences and Engineering Research Council
OB	Outbound
OC	Optical Carrier
OSI	Open Systems Interconnection
PAR	Placement And Routing
PATRICIA	Practical Algorithm to Retrieve Information Coded in Alphanumeric
PBD	Platform-Based Design
PBV	Partial Bit vector
PC	Packet classification
PCI	Peripheral Component Interconnect
PFAAE	Packet Filter Acceleration Assist Engine
PLD	Programmable Logic Device
POS	Packet Over SONET
pps	Packets Per Second
QoS	Quality of Service
RAM	Random-Access Memory
RTC	Real-Time Clock
RL	Range Location
RLDRAM	Reduced Latency Dynamic Random-Access Memory
RLOC	Relative Location Constraints
RPM	Relationally Placed Macros
RPP	Rapid Prototyping Platform

Acronym	Definition
RTL	Register Transfer Level
SDRAM	Synchronous Dynamic Random-Access Memory
SerDes	Serializer/Deserializer
SIP	Source Internet Protocol Address
SoC	System-on-Chip
SOCRN	System-on-Chip Research Network
SOP	Start-of-Packet
SM	State Machine
SMTP	Simple Mail Transfer Protocol
SSRAM	Synchronous Static Random-Access Memory
SP	Source Port
SW	Software
TAP	Test Access Point
Tcl	Tool Command Language
TCP	Transmission Control Protocol
TELNET	Teletype Network
TSAP	Transport Service Access Point
UART	Universal Asynchronous Receivers and Transmitters
UCF	User Constraints File
UDP	User Datagram Protocol
us	Micro-Seconds
VHDL	VHSIC Hardware Description Language
VHSIC	Very-High-Speed Integrated Circuit
ZBT	Zero Bus Turnaround
1-D	One-Dimensional
2-D	Two-Dimensional
4-D	Four-Dimensional

Conventions

The following conventions are used in this document:

- Bits within parameters and registers, and signals are always listed from most significant bit (MSB) to the least significant bit (LSB).
- When a bit is described as "set," its value is set to 1. When a bit is described as "cleared" its value is set to 0.
- The word "assert" means that an active-high signal is pulled high or that an active-low signal is pulled low to ground.
- The word "de-assert" means an active-high signal is pulled low to ground or an active-low signal is pulled high.
- Bits within a parameter or register bits are indicated AA [n:0] when more than one bit is involved in a description.
- Hex values are indicated with "0x" preceding the hex value, as follows: 0x0000FFFF is the value written to a four-byte parameter.
- Binary values are denoted with a "'b" preceding the binary value. For example: 'b1010 is the value written to a four-bit parameter.

Chapter 1: Introduction

From their humble beginnings, computer networks have evolved into an indispensable instrument in managing day-to-day operations in government, industry, and academic institutions. Organizations are quickly realizing the benefits of implementing distributed, client-server architectures where servers and workstations communicate through networks. This architecture facilitates communications and helps provide access to information quickly while reducing costs. While enabling drastic improvements of organizational integration and management, enterprises are also connecting their networks to the Internet. This is necessary to remain competitive and maintain a visible and seamless business presence with customers, partners and suppliers. However, this assimilation of operations does not come without sacrifice. An enterprise's dependence on networks is elevated along with the potential risks and cost of intrusions, failures and compromises. In the business sector, attacks on networks may lead to a loss of reputation, revenues, time and sensitive information, whereas in the medical or National Defense sectors an attack may even lead to loss of lives. Fortunately, it is possible to mitigate these risks by ensuring the security of our networks. This is one reason the area of packet filtering or classification is a focus of active investigation and research.

In addition, the Internet is continually evolving and the traditionally best-effort service is not adequate to meet the growing demands for both bandwidth and new services. The ever-increasing number of users and the growth of multimedia content have led to an exponential increase in the demand for bandwidth and improved *Quality of Service* (QoS). To meet this demand and support emerging bandwidths, routers capable of handling traffic volumes on the order of *millions of packets per second* (Mpps) are needed. It is also clear that to support new differentiated services, routing and packet classifying technology must continue to advance from the basic destination field forwarding to incorporate a whole new range of capabilities. In turn, without new high-speed security tools such as packet filters, the Internet will become even more vulnerable and our networks more susceptible to attack.

Unfortunately, it is not easy for security products to keep pace with increasing bandwidth requirements and new services. Clearly, when offered a choice between a product that is secure and one that is not, most people would prefer a secure product provided the cost and performance are comparable. However, this is rarely the case. Security requires a significant investment in

resources and is typically the bottleneck in system performance. Furthermore, technology evolves so rapidly, *Information Technology Communication* (ITC) and network vendors often sacrifice security features to meet time-to-market. Therefore, for the Internet to successfully evolve towards a full service, reliable and secure network, it is imperative that a fast and scalable solution to the multi-dimensional *packet classification* (PC) problem be achieved.

1.1 Classification Problem Statement

The basic PC problem has been well articulated in literature [7], [4]. In abstract terms:

- A network element maintains a database of n rules $R = \{r_1, ..., r_n\}$ for processing incoming packets across k fields or dimensions.
- Each rule r_i consists of a filter F_i and an associated action. Filters are constraints on the values in the fields of the packet header.
- Each filter F_i has k elements $\{e_{i,1}, e_{i,2}, \ldots, e_{i,k}\}$ corresponding to the fields in packet headers, which it should match. Each header field, represented by a string of bits, is assigned a match type: exact match, wildcard match, prefix match, or range match.
- If more than one filter matches an incoming packet, the tie is broken by using priorities assigned to each filter.

In short, for every incoming packet, the classification algorithm performs a search operation using the header fields to find the best matching filter and then executes the associated action.

1.2 Thesis Statement and Objectives

To address the aforementioned needs, this thesis examines state-of-the-art packet classification schemes en route to developing a novel algorithm. To date, numerous approaches to the PC problem have been introduced and evaluated in literature. Generally, the techniques fall under computational geometry, basic search data structures, heuristics and hardware-based/accelerated approaches. As rule-sets and link speeds at the network edge continue to climb, hardware-assisted approaches are emerging as the front-runner, achieving wire-speed performance in network equipment.

The algorithm presented builds upon two such approaches, namely the Lucent *Bit-Vector* (BV) and *Aggregated Bit-Vector* (ABV) algorithms [1], [2]. Two new ideas are presented

creating what will be called the *Compressed Bit-Vector* (CBV) algorithm. Since it is well known that many efficient solutions exist for *two-dimensional* (2-D) packet classification, the CBV scheme breaks the general multi-dimensional problem into more manageable 2-D problems each producing compressed bit-vector solutions. This provides a possible framework that will allow for the incorporation of various two-dimensional search algorithms into one system for performing multi-dimensional classification. By strictly storing and operating on hierarchically compressed bit-vectors, it is shown that the algorithm's memory usage is scalable to large rule-sets. As with the ABV and Lucent BV schemes, the CBV algorithm is equally amenable to a hardware and software implementation. This is critical because it is a design objective to determine an optimal balance between hardware optimization and programmable flexibility.

The CBV algorithm is intended for an *application-specific integrated circuit* (ASIC), but by implementing and functionally verifying a portion of the algorithm on a *System-on-Chip* (SoC) *rapid prototyping platform* (RPP) a proof-of-concept implementation has been achieved. Platform-based design is a major facilitator in the development of complex *integrated circuits* (ICs) providing many benefits without a commitment to silicon. It allows for embedded software development early in the design cycle, the resolution of hardware and software integration problems, the ability to validate third party IP blocks in a system environment and the capability to experiment with the implementation. It is also useful for acquiring realistic performance statistics to determine if ASIC system performance targets are achievable. To grasp the exact issues considered by this thesis, Section 1.3 briefly explores the role of the CBV algorithm in a typical firewall application.

1.3 Protecting the Network

While the classification techniques presented in this thesis are applicable to many applications that require packet classification, some more detailed background is provided on network level firewalls. This is because security is not only a hot topic since the events of September 11, but it provides a concrete application of classification. Firewalls generally require multi-dimensional classification on fields such as the protocol, source and destination *Internet Protocol* (IP) addresses, and ports, using a combination of exact, prefix, range and wildcard matching techniques.

Firewalls are a tool that offset the associated risks and advantages of using the Internet at the network level. As our networks continue to grow, it becomes difficult to secure computers on a host basis. As such, more and more organizations look to secure their entire network. This is an obvious transition, where the focus is on securing and controlling network access to a group of host computers and the services provided rather than on an individual basis. The advantage of this approach is that one firewall can help protect numerous machines against attack from networks outside the firewall regardless of the individual host security. Figure 1-1 provides a simple example of an Internet firewall installed at the network edge or point where an internal network connects to the Internet.

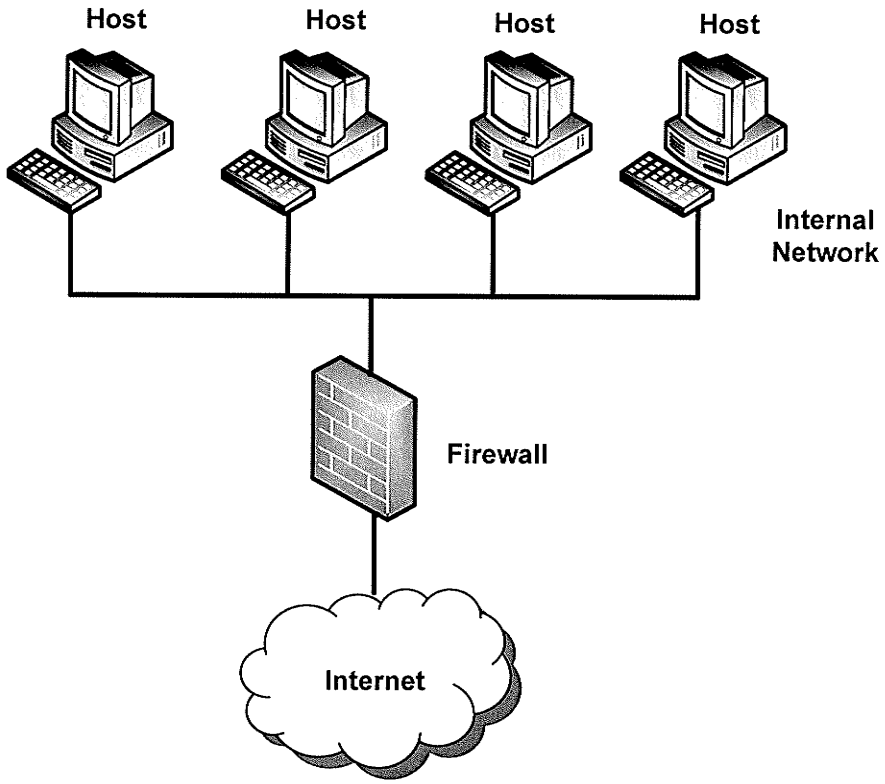


Figure 1-1: Firewall Separating an Internal Network from the Internet

Although the network topology in Figure 1-1 is extremely simple, it illustrates the role of a firewall in the network security sense. In this case, the entire enterprise is protected by the firewall. It places limitations on the amount and type of communication that can occur between the protected network and the external network. As in the diagram, firewalls generally have two or more interfaces where traffic comes in on one interface and comes out on another. However, it

is important to note that a firewall is rarely a single object. Typically, it will be made up of various combinations of screening routers, proxy servers or gateways and *network address translation* (NAT) systems. The screening router is the main component, routing between internal and external hosts. When packets traverse it, it has the opportunity to classify and filter packets into network flows based on a database of rules. Associated with each flow or rule is an action (e.g. allow or deny) reflecting a facility's own security policies. Decisions are often based on Layer 3 and Layer 4 headers and sometimes even application data. This is in contrast to conventional routers, which use only the destination IP address.

Most modern firewalls are also stateful. This means that once the first packet in a network flow is allowed to cross the firewall, subsequent packets belonging to the same flow including return traffic are also allowed through. Typically, statefulness is implemented in two separate search engines. A slow algorithm will implement "first match" semantics, checking the incoming packet against all rules in the database while a fast state lookup checks whether a packet belongs to an existing open flow. This two-part design is particularly efficient for long *Transmission Control Protocol* (TCP) connections where the fast state lookup handles the majority of packets [15].

The CBV algorithm presented in this thesis is intended to replace the slow algorithm. However, experimental trials show that this algorithm does not need to be inherently slow. Implemented on an ARM-based SoC platform, inbound and outbound throughputs of approximately 560,00 and 24,000 *packets per second* (pps) are achieved across four fields and 32K rules (16K in each direction). Based on this platform analysis, it is also demonstrated how an ASIC implementation of the algorithm could sustain a *Gigabit Ethernet* (GE) line-rate on 8K rule-sets across multiple fields. Thus the CBV algorithm can indeed be an efficient hardware-amenable algorithm for firewall packet filtering and classification.

1.4 Thesis Organization

This thesis is organized into eleven chapters involving SoC design, packet classification and an analysis of a novel packet classification algorithm. Chapter 2 outlines a SoC platform-based *field-programmable gate array* (FPGA) design flow for an ARM-based rapid prototyping platform. The remaining chapters elaborate on the various design steps from system specifications to system architecture mapping through design signoff. Chapter 3 introduces

background on packet classification. The criteria and match types that a packet classifier uses for typical network applications are illuminated. This is followed by a list of design considerations and performance metrics for evaluating PC algorithms. In Chapter 4, general approaches to the PC problem are elaborated along with examples from recent literature. This is followed by a comparative evaluation of current solutions to the problem. Chapter 5 builds on these findings, providing the details behind a new scheme. Justification for the design of the CBV algorithm is provided along with an example of the pre-processing and search operations. Chapter 6 continues with the system architecture and hardware/software partitioning of algorithm. Chapters 7 and 8 detail the hardware specific flow from design through implementation and simulation. Next, the hardware/software integration and the quality of the design are evaluated in chapters 9 and 10, respectively. Finally, Chapter 11 wraps up with some conclusions and recommendations for future work.

Chapter 2: System-on-Chip Design Methodology

System-on-Chip represents a complex IC that integrates the major functional elements of a complete end product into a single chip. In general, it will include processor(s), memory blocks, a *universal asynchronous receiver and transmitter* (UART), timer(s), external interfaces such as Ethernet, and *general purpose I/Os* (GPIOs) with an underlying bus architecture like ARM's AMBA (*Advanced Microcontroller Bus Architecture*) to tie it all together. In addition, a SoC also incorporates all of the necessary software for controlling the processors, external interfaces and peripherals.

Due to the multifaceted nature of SoC design, companies are beginning to include SoC platforms into their design methodology. In this case, the hardware is mapped onto an FPGA-based emulation platform that mimics the behavior of the SoC, and the software modules are loaded into the platform memory resources. Once programmed, the emulation platform enables the SoC hardware and software to be tested and debugged at near to full operational speed [44].

Including a SoC platform in the design flow provides several advantages. It allows for embedded software development early in the design cycle, eases integration risk by insuring that all IP works together, reduces licensing and contractual negotiation time by limiting the platform to a single license, and significantly cuts cost by allowing for the bulk of the platform to be reused in multiple follow-on designs [33].

2.1 SoC Platform Design Flow

Like most SoC designs, the CBV packet classifier includes both software and hardware components. Therefore, the design flow will involve both hardware and software flows and tools. Figure 2-1 illustrates the design flow used in this project. The main advantage of this flow is that it promotes the development of hardware and software in parallel. The flow was modeled after one developed by the *Canadian Microelectronics Corporation* (CMC) to accompany an FPGA-based ARM integration platform and is complete from design specification to hardware and software integration [38]. The architectural environment provided by this platform is described in further detail in later chapters.

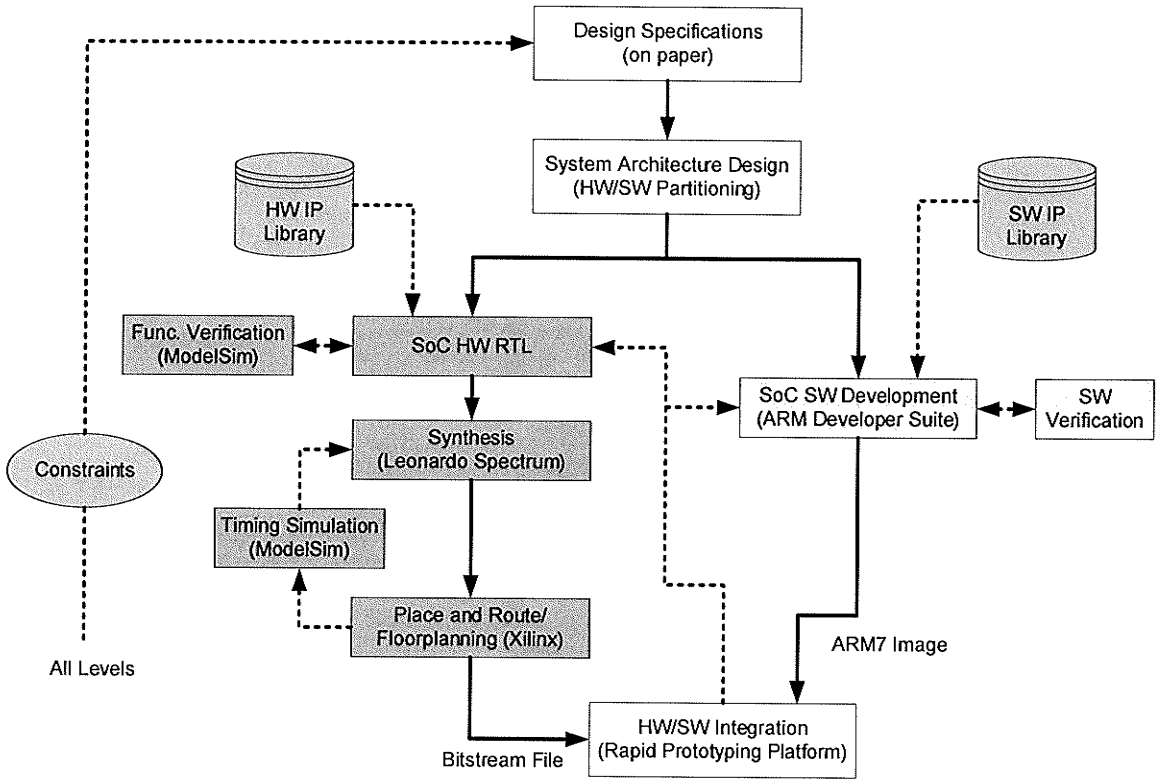


Figure 2-1: FPGA-Based System-on-Chip Design Flow [38]

The next few sections detail the specific steps in the flow that are covered within the scope of this document.

2.1.1 Design Specifications

The design flow typically begins with market research and setting out design requirements and specifications. The functionality of the system must be detailed along with design targets and constraints. These will be used to guide the designers and evaluate the design.

2.1.2 System Architecture Design and Partitioning

Once the functional design specifications have been developed, the architecture or possibly multiple architectures on which the system functionality may be realized are defined. For pre-existing *platform-based design* (PBD), a basic architecture is already in place. Therefore, the main task is mapping functional units onto architectural units. In some cases for generic platforms, additional modifications or enhancements to the platform may be required. Once the architectural mapping is complete, the design may be handed off to hardware and software teams.

2.1.3 Software Flow

The focus in this document is geared towards the hardware flow and implementation. The software flow is beyond the scope of this document.

2.1.4 Hardware Flow

2.1.4.1 SoC Hardware Description Language Coding

Based on the design specifications and architectural modeling, the design is typically coded in Verilog and/or VHDL with a simple text editor. As shown in Figure 2-1 it is also possible to tap into an existing hardware IP library. This can reduce the HDL design effort. For example, Xilinx provides a free library of IP cores with their tool. In this thesis, *intellectual property* (IP) cores and *register transfer level* (RTL) code available from Xilinx, ARM and Memec Design were used where possible.

2.1.4.2 Functional Simulation

The functionality of a design can be verified at several points in the design flow. Design simulation allows you to verify and debug individual blocks or complex functions in a relatively small amount of time with a large degree of design visibility. It is highly recommended to simulate a design before synthesis because it exposes logic errors that would otherwise be passed through the synthesis step. A popular software tool for performing this task is ModelSim from Mentor Graphics. This simulator will compile and interpret *hardware description language* (HDL) code into circuit functionality and display logical results of the described HDL as stimulus is applied. Testbenches for simulation are usually created alongside the HDL coding and may take the form of HDL or a higher-level language such as Tcl.

2.1.4.3 Synthesis

Directed by a set of timing and device constraints, synthesis optimizes and maps RTL code (typically from industry standard HDL languages such as Verilog and VHDL) into gate-level netlists that can be mapped into an FPGA's logic blocks. Popular tools for performing this task in FPGA-based design are Synplify Pro from Synplicity or Leonardo Spectrum from Mentor Graphics. These tools can also write VHDL and Verilog netlists after synthesis, which can be used to re-simulate in order to verify that synthesis has not interfered with functionality. If desired, designers can also use gate-level delays to perform a back-annotated timing simulation.

2.1.4.4 Implementation

Once synthesis is complete, design implementation is performed using tools supplied by the FPGA vendor. Xilinx and Altera are the most popular for this operation. This stage may be broken into four steps [34]. The first is translate, which merges the netlist produced by synthesis with a *user constraints file* (UCF). The UCF specifies pin constraints and additional timing information for the target device. Next, a mapping step fits the design into the available logic blocks and resources for the target device. *Placement and routing* (PAR) will then place the design and route it while attempting to meet the timing constraints. Once the design has been fully routed and timing constraints have been met, a bit stream file is generated that can be used to program the device. If desired, the bit file may be converted into different formats depending on how the FPGA is to be programmed. For example, it may be converted into an image for a flash device.

2.1.5 Hardware/Software Integration

While the hardware and software designs are often happening in parallel and verified separately, it is critical to ensure that they will work together. In extremely complex designs, hardware-software co-verification tools like Seamless may be used to link hardware and software simulations. This approach was not explored in this design but the communication between FPGA boundaries on the platform was verified early in the design cycle. For more detail on the hardware and software integration see Chapter 9.

2.2 Summary

This chapter has introduced the basic FPGA and platform-based design flow that is followed throughout this thesis. Chapters 5 through 9 aim to address each stage of the flow. Particular emphasis is placed on the design specifications, architectural mapping and partitioning, the hardware components: HDL coding and design, simulation, synthesis and implementation, and the hardware/software integration. Once hardware and software integration testing and debug are complete, the next step is to perform realistic system tests to evaluate the quality and performance of the design. Results from this stage may be used to improve or fine-tune the design. At this point, the overall design can continue with an ASIC design flow through fabrication or it may enter design signoff as a standalone proof-of-concept.

Chapter 3: Design Background

This chapter provides background on packet classification, its applications and the underlying research challenges it poses. As part of their packet processing stage, most network equipment such as firewalls must perform complex packet analysis or classification tasks. In brief, packet classification is the process of matching packets against a pre-defined database of often partially defined header fields. Since network applications are continuing to change, there is a strong demand for classification architectures that are scalable and flexible enough to support a variety of applications operating at wire-speeds. This chapter begins by defining common classification header fields such as source and destination addresses, protocols and port numbers. This is followed up in Section 3.2 by a description of popular classification match types. Finally, the chapter concludes by outlining a set of design criteria and performance metrics to assist researchers in developing and evaluating new algorithms.

3.1 Classification Criteria

The criteria a packet filter uses when inspecting packets is dependent on the network application. Every packet includes a set of headers fields, which contain information regarding packet characteristics. The most typical information used in classification comes from layers two, three and four of the *Open Systems Interconnection* (OSI) Reference Model.

3.1.1 Layer 2 Fields

Layer 2, also referred to as the Data Link Layer, provides the procedural and functional capability to transfer data between network entities. Typically, this is via Ethernet or *Asynchronous Transfer Mode* (ATM) signaling. An Ethernet frame is shown in Figure 3-1 possibly encapsulating the TCP/IP protocol. This frame will be transported across a physical cabling system to Layer 2 of the destination device.

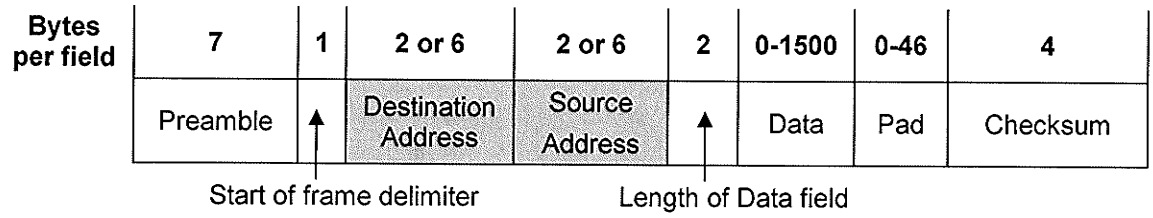


Figure 3-1: Ethernet Frame Format

From a classification standpoint, Data Link Layer headers contain two main fields of interest which are highlighted in Table 3-1. The first field is implied from the packet while the second field is part of the Ethernet frame. Ethernet uses addresses referred to as *Medium Access Control* (MAC) addresses. These are 48 bits long and each network card has a unique address configured by its manufacturer.

Table 3-1: Layer 2 Classification Header Fields

Layer 2	Length	Typical Format
Type of Packet		Ethernet, ATM
Ethernet MAC address	48 bits	00-0C-F1-7D-52-3C

3.1.2 Layer 3 Fields

Layer 3, also referred to as the Network Layer, allows hosts to inject packets into a network and allows them to travel independently to the destination host, which may be on a different network. The most common Layer 3 protocol is the Internet protocol. Figure 3-2 illustrates the packet header format for IP version 4. For the upcoming IP version 6, this header has changed slightly. A Flow ID has been added to assist in classification, the IP address lengths have been extended to 128 bits and some fields have been removed.

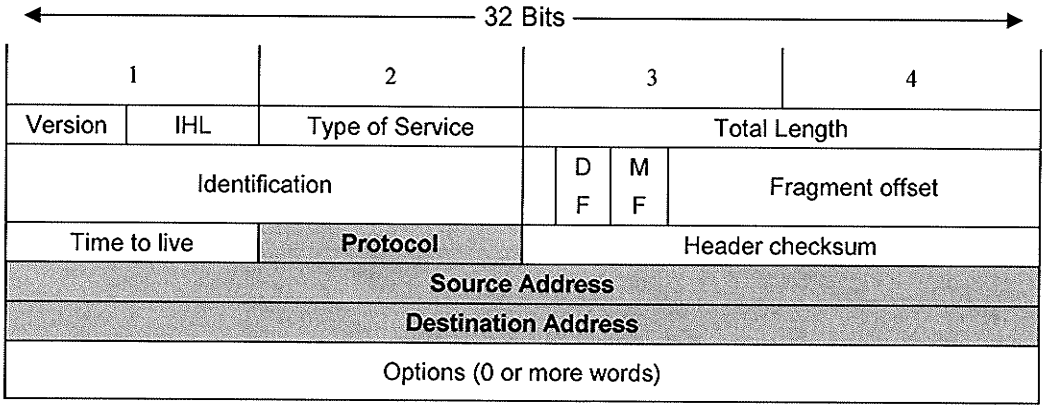


Figure 3-2: IPv4 Header

From a classification standpoint, the IP header contains three primary fields of interest. These fields are outlined in Table 3-2 along with their lengths and standard formats. The *destination IP address* (DIP) is likely the most frequently utilized. Each router between a source and destination performs a route lookup based on this field to determine the next hop or path for the packet. However, for basic classification operations, the *source IP address* (SIP) is also inspected. Together, these two fields indicate the network number and host number providing enough information to define a flow between two hosts. Another important field is the protocol field, which holds the type of protocol for Layer 4. The most common values for this field are the *Transmission Control Protocol* (TCP) and *User Datagram Protocol* (UDP), although there are many other protocols such as the *Internet Control Message Protocol* (ICMP).

Table 3-2: Layer 3 Classification Header Fields

Layer 3	Length	Typical Format
Source IP address (SIP)	32 bits	130.179.128.15
Destination IP address (DIP)	32 bits	130.179.128.15
IP protocol type: identifies the IP body packet type	8 bits	TCP, UDP, ICMP

3.1.3 Layer 4 Fields

Layer 4, also referred to as the Transport Layer, allows peer entities on the source and destination hosts to carry on a conversation. From a classification standpoint, only TCP and UDP provide any useful information. Figure 3-3 and Figure 3-4 illustrate the packet headers for these protocols while Table 3-3 describes the primary fields of interest for packet classification.

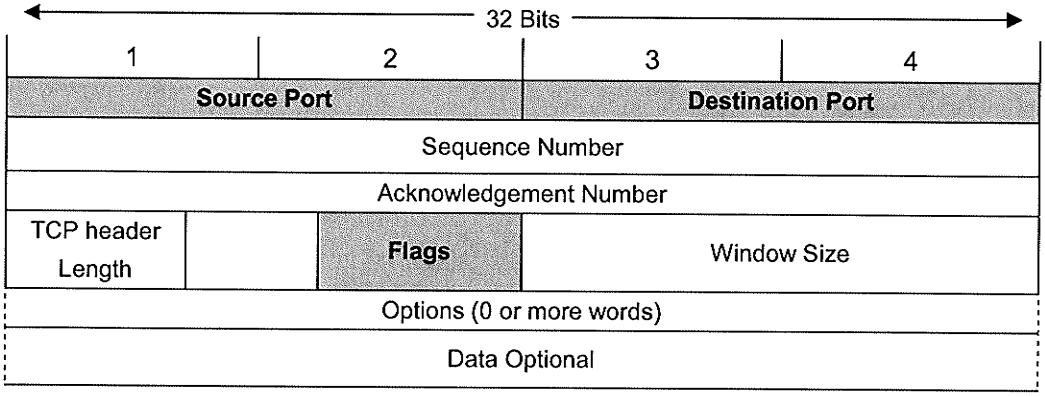


Figure 3-3: TCP Header

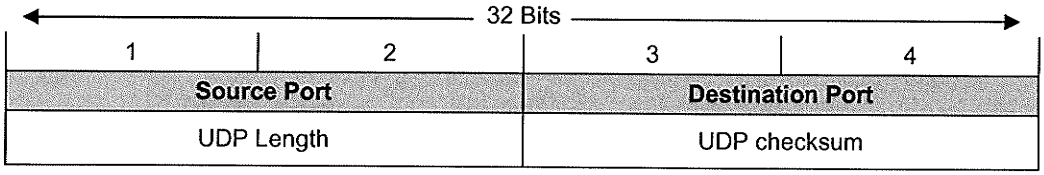


Figure 3-4: UDP Header

The most common fields are the *source and destination port* numbers (SP and DP). These fields are important because they are used to send packets to the appropriate applications and help to further characterize a flow. Together, a host IP address and port uniquely define a *Transport Service Access Point* (TSAP). Each host may decide how to allocate its ports starting at a value of 1024. Port numbers below 1024 are called well-known ports and are reserved for standard services. For example, if a source host wishes to connect to a destination host to transfer files using FTP, then a connection must be made to the destination port 21. Analogously, to form a TELNET connection port 23 is used.

Table 3-3: Layer 4 Classification Header Fields

Layer 4	Length	Typical Format
Source Port (SP): indicates what client or server process a packet is coming from on the source machine.	16 bits	80
Destination Port (DP): indicates what client or server process a packet is arriving to on the destination machine.	16 bits	1024
TCP flags: identifies special packets especially on establishing and closing connections.	8 bits	ACK, FIN, SYN...

Although Layers 2, 3 and 4 contain the most common fields for packet classification, it is also possible to look past the packet headers at data destined for Layer 7 (i.e. the Application Layer). The Application Layer is concerned with the high-level protocols such as HTTP, TELNET, FTP, SMTP and DNS.

3.2 Classification Match Types

Now that the basic packet classification header fields have been identified, it is important to understand the fundamental classification types. The matching type should be implicit from the specification of the fields within a rule. Sticking with the firewall application, the five-dimensional rule in Table 3-4 will be used to illustrate the basic matching types. This rule specifies a traffic flow addressed to subnet 128.179.*.* using TCP and destination ports in the range 0-1023.

Table 3-4: Typical Filter Rule

Field	Destination IP	Source IP	Protocol	Source Port	Destination Port
Rule	128.179*	128*	TCP	*	0-1023
Type	Prefix	Prefix	Exact	Wildcard	Range

3.2.1 Exact Matching

For exact matching, the packet's header field must exactly match the corresponding filter field in the rule. This type of matching may be associated with fields like the protocol field for the rule in Table 3-4 or TCP flag field for IPv4.

3.2.2 Prefix Matching

In prefix matching, the filter field should be a prefix of the corresponding header field. This type of matching is amenable for IP source and destination addresses. This may be useful in a firewall application to block access from an entire sub-network. Prefix rules also provide a built-in way to differentiate and prioritize overlapping rules. A prefix will be entirely contained within another prefix or will not overlap. The higher priority is assigned to the fully contained prefix.

3.2.3 Range Matching

In a range match, the header field must fall in a specific range outlined by the corresponding filter field. This type of matching could be used for matching port numbers in ranges as seen in Table 3-4. It is a more difficult operation than prefix matching and often classification algorithms will convert ranges into a prefix format.

3.2.4 Wildcard Matching

Wildcard matching is really an extension of the exact match providing a "match-all" case. It is considered a match when an exact match cannot be found. Unfortunately, this type of matching can increase the complexity especially when matching many fields against a multi-dimensional database of rules. The source port header field in Table 3-4 provides an example of a wildcard.

To further clarify matching techniques, consider a firewall example with a "default deny" approach. A "default deny" policy means to deny everything and then selectively allow certain traffic through the firewall. This approach is recommended because it errs on the side of caution

and makes writing a rule-set easier. With this policy, any packet with header fields matching the rule will be allowed to pass. Therefore, a packet with the header (128.179.130.15, 128.180.132.16, TCP, 1055, 23) will match the rule and be forwarded. However, a packet with the header (128.190.177.12, 128.180.132.16, TCP, 80, 23) does not match the destination IP address and will be blocked.

3.3 Design Considerations for Classification Algorithms

In the preceding discussions, the motivation for research in packet classification was illuminated along with some fundamentals of the PC problem. Organizations need to protect data, resources and their reputation, but for the most part are not willing to sacrifice performance. Therefore, a set of criteria to assist researchers in developing new packet classification algorithms and a set of performance metrics for evaluating algorithms must be formulated. In the extensive collection of papers that address the PC problem, much of the relevant design criteria for efficient classification have been presented [1], [6], [12], [16]. Of course, the actual requirements will depend on the application and where the classification is performed in the network.

In the ensuing sections, a general set of design considerations and metrics are summarized to evaluate PC algorithms. In subsequent chapters, the dominant design considerations will be used to assist in the design of a novel algorithm for future packet classification operations.

3.3.1 Speed

Internet service providers are building networks with link capacities of 10 *Gigabits per second* (Gbps) (10 GE or POS OC-192) and link speeds are predicted to exceed 40 Gbps (40 GE or POS OC-768) in the near future [2]. Any algorithm to be used in applications such as core routers must be fast enough for use with these networks and scalable to networks of the future. It is undesirable to have an algorithm where congestion and packet loss are a result of processing time and not the link capacity. Therefore, the ability to operate and support applications at wire-speed has and will continue to be a critical, if not the most crucial, design factor in classification algorithms. In this context, wire-speed is the ability of a network component to sustain the data bit transmission rate that is specified by the corresponding media transport protocol. For example, to support a link capacity of 1 GE requires packet classification on the order of 1.49 Mpps. Equation 3-1 shows how this number was derived.

Equation 3-1: Gigabit Ethernet Maximum Packet Throughput

$$Max\ Packet\ Throughput = \frac{Wire\ speed}{Smallest\ Pkt\ Size + Interframe\ gap + Preamble}$$

Wire speed : 1 Gbps

Smallest Packet Size : 64 bytes

Interframe gap : 12 bytes

Preamble : 8 bytes

$$Maximum\ Packet\ Throughput = \frac{1\ Gbps}{64\ bytes + 12\ bytes + 8\ bytes} = 1.49\ Mpps$$

The smallest packet size is obtained from the IEEE 802.3 standard, which states that valid Ethernet frames must be at least 64 bytes long [48]. Recalling the frame format from Figure 3-1, it is clear that 8 bytes must be added to account for the preamble and start frame bytes. Furthermore, between Ethernet frames there must be an inter-frame gap of at least 96 bit times or 12 bytes. Therefore, the minimum packet size is 72 bytes and to obtain an accurate measure of throughput, the inter-frame gap of 12 bytes is added to give 84 bytes.

3.3.2 Worst-Case Versus Average Case Performance

There is a widely held view that for access time performance of packet classification, one must focus on the worst-case rather than the average case. Ideally, the algorithm must have small worst-case execution times that are independent of traffic patterns. The worst-case search time is important because it identifies the peak bandwidth that an algorithm can sustain [1], [7]. However, numerous proposals base their results on typical rule-sets and traffic patterns obtained by monitoring the world-wide-web [12].

Table 3-5: Average Case Performance Requirements

Media-Protocol	1 Gigabit Ethernet	10 Gigabit Ethernet
Wire-Speed[1]	1 Gbps	10 Gbps
Average Packet Size	256 Bytes	256 Bytes
Average Packet Throughput	0.45 Mpps	4.53 Mpps
Average Time/Packet	2.2us	0.22 us

[1] The actual 10-bit encoded data for 1 and 10 GE would be transferred on the line at 1.25 Gbps and 12.5 Gbps respectively.

Table 3-5 illustrates the average case throughput and per-packet processing times for Ethernet. By comparing this with the maximum throughput required by 1 GE, there is a clear difference in performance requirements. This may be acceptable for an algorithm that targets a specific set of applications. However, meeting the worst-case requirements provides a guarantee that the algorithm will never take any longer and is therefore the most important. As well, the worst-case may occur frequently for some algorithms. As an example, if caching is used and there is a miss, this may cause worst-case searching behavior to occur.

3.3.3 Memory Accesses

Memory accesses should be minimized since they are the main bottleneck in performance [1]. From the above results, the per-packet memory access budget for state-of-the-art *Reduced Latency Dynamic Random Access Memory* (RLDRAM) devices with a random access latency of 20 ns is: approximately 100 memory accesses for GE, and approximately 10 memory accesses for 10 GE.

3.3.4 Storage Requirements

An algorithm should achieve the required target access speed while minimizing the amount of memory or space used. Good algorithms tend to store each rule only once or at least a small constant number of times. For the algorithm to scale to large rule-sets there must not be a memory explosion. For example, $O(n^k)$, where k is the number of dimensions.

3.3.5 Fields

While special hardware devices such as *Content Addressable Memories* (CAMs) and heuristic based algorithms have been employed to meet classification demands in the past, these algorithms tend to focus on specific applications. It is unclear which headers or fields should be used to provide the services of the future, only that there is need for classification algorithms that are flexible enough to support a wide variety of applications. Efforts should therefore focus on a more general problem: how to find the appropriate rules from a number of k-dimensional filters, based on a generic set of packet field headers.

3.3.6 Rules

3.3.6.1 Number of Rules

Rule databases are growing and are predicted to increase to several millions of rules. In the past, packet classification was used for security and firewalls, which generally led to relatively small databases on the order of a few thousand rules. However, with the new demand for differentiated services, it is likely that these databases will grow extremely large. Edge equipment normally maintains a database of a million or more flows and flow association requires a lookup operation against this large database [16].

3.3.6.2 Nature of Rules

The complexity and size of the rule-set is an indicator of the flexibility of the classification algorithm to support a variety of applications. Current routers use rules with prefix masks on destination IP addresses. However, general masks such as arbitrary ranges are more flexible. Table 3-6 illustrates a few simple filter rules for a firewall application.

Table 3-6: Example Firewall Rules (ANY = *)

Rule	Source Address	Destination Address	Protocol	Source Port	Destination Port
A	192.135.*.*	ANY	TCP	>1023	23 - 55
B	192.134.1.11	192.131.*.*	TCP	23	<1023
C	192.*.*.*	192.134.1.*	UDP	ANY	ANY

3.3.6.3 Updating Rules

The number of changes to the rule-set depends on the application. Changes could occur because of a policy change or, in stateful packet filtering when a new flow is inserted or deleted. To achieve this, an algorithm will need to perform inserts and deletes at speeds of 10-100 us [7].

3.3.7 Pre-computation

Pre-computation can be defined as the process of transforming the representation of a filter database to represent the same data in a way more suitable for a specific classification procedure. The goal is to reduce the storage requirements and/or search time. Although pre-computation can be used to optimize the results of almost every algorithm, by conditioning the data or representing it in some convenient form, update speed suffers. A good algorithm should attempt to look for

pre-computations and data structures that allow for incremental updates. At present, no known pre-computation scheme explicitly attempts to optimize the update rates [6].

3.3.8 Priority

It is possible that some packets will match more than one rule. The rule-set must allow for priorities to be imposed on the rules so that only one directive will finally be applicable to the packet (i.e. allows one to distinguish the lowest cost filter). In some applications, there may be an intuitive or natural way to prioritize, as is the case when performing IP lookups on prefixes. In other cases, alternative methods must be employed.

3.3.9 Hardware Implementation

For operation at very high speeds, the algorithm should be amenable to a hardware implementation. The algorithm structure should seek to take advantage of hardware parallelism and pipelining. This trend is seen in many commercial state-of-the-art packet classifiers.

3.4 Performance Metrics

Typically, by analyzing several candidate algorithms for a particular problem, the most efficient ones can be identified while the less effective algorithms can be weeded out. However, it would be extremely difficult and time consuming to analyze and determine the relative efficiency of algorithms based on the broad set of design criteria outlined above. Therefore, complexity theory is used and the asymptotic efficiencies of algorithms are compared.

Complexity theory is part of the theory of computation that concerns the resource requirements during computation to solve a problem given the problem size n, which is typically the number of items. After all, while computers are becoming fast and memory is becoming cheaper, they are still limited resources.

The complexity of a resource is expressed in what is commonly referred to as big-O notation. The big-O is a symbolism used to describe the asymptotic upper bound for the magnitude of a function in terms of another, usually simpler, function [45]. For a given function $g(n)$, the term $O(g(n))$ represents the set of functions in Equation 3-2 while a deeper explanation can be obtained in most algorithm books [18].

Equation 3-2: O-notation functions for $O(g(n))$

$O(g(n)) = \{f(n): \ 0 \le f(n) \le cg(n) \text{ for all } n \ge n_o \}$

c and n_o are positive constants that must be fixed for the function f and can not depend on n.

In other words, the O-notation gives an upper bound on a function to within a constant factor [18]. Some of the most common complexities with respect to time in increasing order of magnitude are interpreted in Table 3-7.

Table 3-7: Complexity Examples

Notation[1]	Name	Meaning/Description
$O(1)$	Constant	The algorithm requires a constant amount of time.
$O(\log n)$	Logarithmic	The algorithm requires logarithmic time. Often the result of binary splitting as in a binary search. The data is halved at each stage so the number of stages until you have a single item is given by $\log_2 n$.
$O((\log n)^c)$	Polylogarithmic	The algorithm requires polylogarithmic time. The problem is likely broken down into a number of nested steps which each take logarithmic time.
$O(n)$	Linear	Complexity is proportional to the size of n. If n is doubled the algorithm is expected to take twice as long.
$O(n \log n)$	Linearithmic	This is something like merge sort. It is a consequence of doing $\log n$ splits and then performing n amount of work at each split.
$O(n^2)$	Quadratic	Proportional to n squared. If you double n, the algorithm is expected to take four times longer
$O(n^c)$	Polynomial	Times such as $O(n^3)$. These are slow algorithms.
$O(c^n)$	Exponential	This algorithm requires an exponential amount of time. After this, factorial and double exponential times are encountered.

[1] n is variable and c is a constant.

From the examples in Table 3-7, it is obvious the order of growth of an algorithm's worst-case resource requirements provide a simple characterization of its efficiency. For algorithm analysis, time and space have been the most conventional measures of performance. Typically, space corresponds to memory consumption while time reflects the number of operations required on a virtual CPU like the Random Access Machine model [18]. This model is considered because it is convenient to define the steps of an operation as machine-independent as possible. Depending on the application, the complexity of other tasks and resources such as updates, rules,

bandwidth, power consumption and hardware may also be considered. However, for analyzing PC algorithms in this thesis, the resource requirements for time and space are predominately used.

3.4.1 Time Complexity

The time complexity of a problem is the asymptotic upper bound on the number of steps that it takes to solve an instance of the problem as a function of the size of the input in the worst-case. For the PC problem, this may be the maximum number of sequential steps to return a match or no match response to an input packet. With respect to firewalls, time-wise linear and binary search algorithms would have complexities of $O(n)$ and $O(\log n)$ as mentioned in Table 3-7. As well, perfect hashing functions or *one-dimensional* (1-D) classifiers using CAMs may be able to achieve $O(1)$. This implies that parallelism, hashing techniques and algorithms, which reduce the number of rules to match against, are enablers in reducing the time complexity [16]. Ideally, an efficient algorithm should have low time complexity to achieve high classification speeds.

3.4.2 Space Complexity

The space complexity is the asymptotic upper bound on the memory requirements to solve an instance of the problem as a function of the size of the input in the worst-case. In the PC problem, this includes the space required to store the rule database as well as the associated data structures [16]. A data structure may be interpreted as the method of storing and organizing data to facilitate access and modifications. For example, data structures may include things like linked lists, hash tables, binary search trees, tries and CAMs. Ideally, the algorithm should have low space complexity so that it is scalable to large rule-sets.

3.4.3 Update Complexity

The update complexity is the asymptotic upper bound on the number of steps required to perform an insert, delete or update to a rule in the rule database. In most current PC algorithms, the update time has been sacrificed for speed and memory [1]. Returning to the linear search algorithm example, it would have an update complexity of $O(n)$ in the worst-case. Ideally, the update complexity should be low to support efficient updates.

3.4.4 Rule Complexity

Defining the rule complexity is not as clear-cut. The rule complexity is somewhat independent from the other three metrics. Normally, it is a property of the algorithm's architecture and is not traded off as the space, time, and update metrics are. Iyer et al. [16] define one method of quantifying the rule complexity as the product of the total number of fields in the rule and the number of operations, which can be supported per field. By defining the rule complexity in this fashion, it reflects on the flexibility of the algorithm. The more complex the rule-set, the bigger the range of applications it can support. For example, an algorithm that can only perform exact matches has a rule complexity of one multiplied by the number of dimensions it can support. An algorithm that supports range matching and operators such as > and < may have a complexity of three times the dimensions. Ideally, the rule complexity should be high.

Chapter 4: Related Work

In this chapter, some of the classical approaches to packet classification are discussed and evaluated. Since the general idea of packet classification was introduced, numerous approaches to the problem have been evaluated in literature. It has been recognized that all of these algorithms may be categorized into four groups albeit with some overlap [6], [7]: basic search data structures [8], computational geometry approaches [1], [2], [9], heuristic-based algorithms [12] and hardware-specific approaches [1], [2]. The first four sections of this chapter describe these approaches along with representative algorithms that the work in this thesis builds on. In all cases the the terms n, d and W represent the number of filters or rules, the number of dimensions, and the length of the search fields respectively. To summarize, Section 4.5 tabulates the various trade-offs involved with these approaches.

4.1 Basic Data Structure Approaches

A data structure is a way of storing data so that it can be used efficiently. Often a carefully chosen data structure will allow a more efficient algorithm to be used. Some of the basic data structures used in classification algorithms include: linear collections, caches, hashes, and hierarchical tries.

4.1.1 Linear Search

A linear search is the most basic algorithm and has been employed in some firewall applications. Typically, the rules are stored in a linked list in order of priority, so that the search may terminate as soon as a match is discovered. While this algorithm is simple to implement and has $O(n)$ space requirements, it does not scale well in terms of time. If the data are distributed randomly, on average $n/2$ comparisons will be needed. In the best scenario, the value is equal to the first element tested, in which case only one comparison is needed. In the worst-case, the value is not in the list and n comparisons are needed. Therefore, the classification time is $O(n)$ and will grow linearly with the number of rules. The worst-case update time for this type of structure is also $O(n)$ and occurs if the rule has to be inserted at the top of the list.

4.1.2 Caching and Hashing

Caching has been another popular tactic to improve classification speeds, especially for IP address lookups. The motivation for using caches is clear; when a new packet arrives at an IP router interface more packets belonging to that flow are likely to follow. Therefore, it may be feasible to perform one in-depth packet classification using many fields and store the associated header and action into a cache. The next time a packet belonging to this flow arrives, it can be processed extremely fast using the cached header and action. However, current schemes based on caching do not perform well in practice. This is a consequence of poor hit rates and small lifetime of the flows [1]. In modern day routers, the number of active flows at a particular interface is also extremely high. This makes the use of hardware caches (usually implemented in CAMs) extremely difficult and one must resort to hash tables.

Like caches, hash tables are used in an attempt to obtain improved deterministic performance. There are many different ways to implement hash functions, but the desired performance is $O(1)$. Typically, this reflects only an average look-up time, where the worst-case may be poor. In addition, while hash tables are more scalable than hardware caches, they are still limited to 1-2 million entries. Considering the future number of prefixes that may be present with IPv6, it is unlikely that this technology will be able to keep pace with the growth and evolution of the Internet. However, it is possible to obtain $O(1)$ performance with perfect hashing at the expense of a significant amount of resources to build complex functions to locate the entries [18].

Furthermore, cache and hash based architectures are traffic dependent. Dependence the traffic patterns means the classification engine of the algorithm has the potential to become backed up by a queue of unrelated packets. In addition, keeping security in mind, any implementation based on caching and hashing will be more susceptible to attack. Once the limitations of an algorithm are known, like the size of the hash or cache table, it can be exploited by malicious users to generate traffic patterns which will cause the device to slow down and even drop packets. Regardless, hashing is being investigated especially for the two-dimensional classification of IP lookups in routers and, in some instances, can be used to successfully accelerate classification. At present, one state-of-the-art algorithm by EZchip uses patented deterministic hash lookups [40].

4.1.3 Tries and Hierarchical Tries

In its simplest form, a trie is like a binary branching tree. It has a tree-like structure, but its name is derived from the word 'retrieval'. Unlike a binary tree, no node in the tree stores the key associated with that node. Rather, its position in the tree shows what key is associated with it. Therefore, all the descendants of any one node have a common prefix. This begins with the root node which is associated with the empty string '*'. Tries are most useful when the keys are of varying lengths and they are particularly well suited for matching IP addresses. Many classification algorithms implement a space-optimized version of the basic trie known as the PATRICIA (*Practical Algorithm to Retrieve Information Coded in Alphanumeric)* trie [46]. To save space, the PATRICIA trie merges any parent node with only one child with its child. The result is that every internal node has at least two children and the edges may contain a sequence of characters opposed to just a single character. This can provide significant saving for sets which share long common prefixes.

Hierarchical tries are a simple extension to the one-dimensional trie. Basically, a one-dimensional trie is constructed with the set of prefixes belonging to dimension one. For each prefix in this trie, a (d-1)-dimensional hierarchical trie is constructed on those rules which match the prefix exactly in the first dimension. The hierarchical tries are linked using next trie pointers creating a total storage complexity of $O(ndW)$ for this structure. The classification procedure for this structure involves traversing the first trie and, at each node encountered following a next trie pointer, to traverse an additional trie representing the next dimension. The time complexity is $O(W^d)$ and incremental updates can be carried out in $O(d^2W)$ time since each component of the updated rule is stored in only one location. For the interested reader, Srinivasan, Varghese, Suri and Waldvogel provide a good example of an algorithm based on a *Grid-of-Tries* (GOT) in [8]. The GOT scheme performs extremely well for two-dimensional prefix matching of source and destination IPs with good memory and time scalability. Techniques are also provided which expand the algorithm's ability to handle port and protocol fields. However, GOT does not support a high degree of rule complexity and in practice does not extend well beyond two dimensions.

4.2 Geometric Approaches

The PC problem has a natural geometric interpretation. It can be viewed as the point location problem in d-dimensional space. Numerous papers regarding packet classification have taken this perspective and built on work surrounding the point location problem [1], [2], [4], [9], [15].

The general point location problem may be specified as follows:

- Each rule r_i forms a d-dimensional object (hypercube) and thus the set of rules create a set of d-dimensional objects. These objects may or may not be overlapping.
- Each packet will correspond to a point in this d-dimensional space.
- The goal is to find the lowest cost object that the point belongs too.

Special cases of this problem occur for one- and two-dimensional spaces. The *IP lookup* (IPL) and *range location* (RL) problems are two popular research topics with regards to packet classification in one dimension. For two-dimensional classification, each rule represents a rectangle on a 2-dimensional grid. The goal is to find the smallest cost rectangle a query point p falls into, if any.

4.2.1 The Geometric Efficient Matching Algorithm

Rovniagen and Wool purpose the *Geometric Efficient Matching* (GEM) algorithm based on a standard computational geometry technique which achieves logarithmic matching time with a worst-case space requirement of $O(n^d)$ [15]. While the space requirement is prohibitive, the authors show that in practice the algorithm uses near linear space and can sustain a matching rate of over 1 million packets per second when tested against rule-sets based on real firewall characteristics. They also introduce a space-time tradeoff by arbitrarily splitting the rule-set into l groups of size $\frac{n}{l}$. This results in a cost factor l slowdown in search time $O(l \log \frac{n}{l})$, but offers an l^{d-1} decrease in space complexity $O(\frac{n^d}{l^{d-1}})$. Rovniagen and Wool also show that the field order has a large impact on the space requirements and build time of their *four-dimensional* (4-D) data structure.

4.3 Heuristic Approaches

Since the multi-dimensional PC problem is extremely difficult, especially when considering worst-case results, some researchers have started to explore structures based on heuristics. The idea is that many rule-sets in current networks have a significant amount of structure and redundancy, which can be exploited by heuristics to provide efficient classification. Two pioneers of this research are Pankaj Gupta and Nick McKeown. They have developed algorithms on the subject by evaluating common databases or classifiers. One such algorithm is known as *Hierarchical Intelligent Cuttings* (HiCuts) [12].

4.3.1 The Hierarchical Intelligent Cuttings Scheme

The HiCuts algorithm takes a geometric view of the PC problem. The basic approach is to build a decision-tree by recursively partitioning (or cutting) the multi-dimensional space into smaller sub-regions. Heuristics, established by extensive pre-processing of the classifier (or rule-set) are used to guide this step and provide a balance between the depth of the tree and amount of memory used. At each node heuristics help select the number of cuts, determine which dimension(s) to cut along, maximize reuse of child nodes, and eliminate redundancies in the tree.

When a packet arrives in the system, the header field values are used to perform a lookup by traversing the decision-tree (i.e. follow the cutting sequence) until a leaf node is reached. The leaf node stores a small number of rules, which are linearly searched to find the desired match.

In theory the algorithm's worst-case search and space complexities are $O(d)$ and $O(n^d)$ respectively. However, the authors show that the results on realistic 4-D rule-sets are much better than the worst case-bounds, requiring 1 MB of storage and 20 memory accesses for a rule-set of 1700 [12]. The drawbacks of this algorithm are that the pre-processing time is quite lengthy and it is not evident whether the algorithm would scale to very large rule-sets.

4.4 Hardware Approaches

Hardware schemes are becoming increasingly popular as network speeds continue to increase. Designers have resorted to CAMs, bitmap intersections with bit-level parallelism and ASIC design to obtain maximum performance. The standard CAM is particularly useful for exact matching taking full advantage of hardware parallelism. Very good search performance can be

obtained as each memory location compares the input or header field values with the contents of its memory location. Furthermore, emerging Ternary CAMs allow the memory to hold masks, thereby supporting rules with a higher degree of complexity [49].

The main disadvantages with CAMs are clear. With growing rule-sets and the transition from IPv4 to IPv6 on the way, memory has become a critical factor in classification algorithms. CAM improvements have not kept pace with regular memory and are traditionally slower, smaller and require more power than conventional memory. This is a result of the complex hardware required to perform the parallel lookups. Therefore, it is unlikely that these memories will be able to keep pace with the database growth expected to be experienced by the recent push towards differentiated services. Updating rules is also costly since the priority of a filter can only be encoded by its memory address. Another drawback of hardware solutions is that they are not always as flexible as software and tend to become outdated as the Internet evolves. Still, some of the most efficient classification schemes in use today such as PMC Sierra's ClassiPI and Netlogic's NSE5512 search engine use hardware to accelerate and improve performance [16], [47]. Therefore, it seems evident that to achieve wire-speed performance in network equipment, some form of hardware assistance is required. Two hardware-assisted algorithms, which rely on bit-vector (BV) and bitmap intersections, are the Lucent Bit-Vector and the Aggregated Bit-Vector schemes [1], [2].

4.4.1 The Lucent Bit-Vector Scheme

The original BV algorithm developed by Lakshman and Stiliadis applies a divide-and-conquer approach where the d-dimensional problem is separated into d one-dimensional problems [1]. Figure 4-1 illustrates a block diagram of the implementation. The algorithm uses binary searches in each dimension with a time complexity of $O(\log n)$ and a bit-vector intersection (i.e. AND operations on bit-vectors) to combine results. The bit-vector intersection step requires examining each of the rules at least once, leading to $O(n)$ execution time. However, the use of bit-level parallelism and large internal buses (1000 bits wide) can accelerate this time by a constant factor. Further detail on the intersection problem is described in Section 4.4.1.1. Unfortunately, the space requirements of the algorithm are $O(dn^2)$. As a result, the algorithm is only reasonable for small rule-sets.

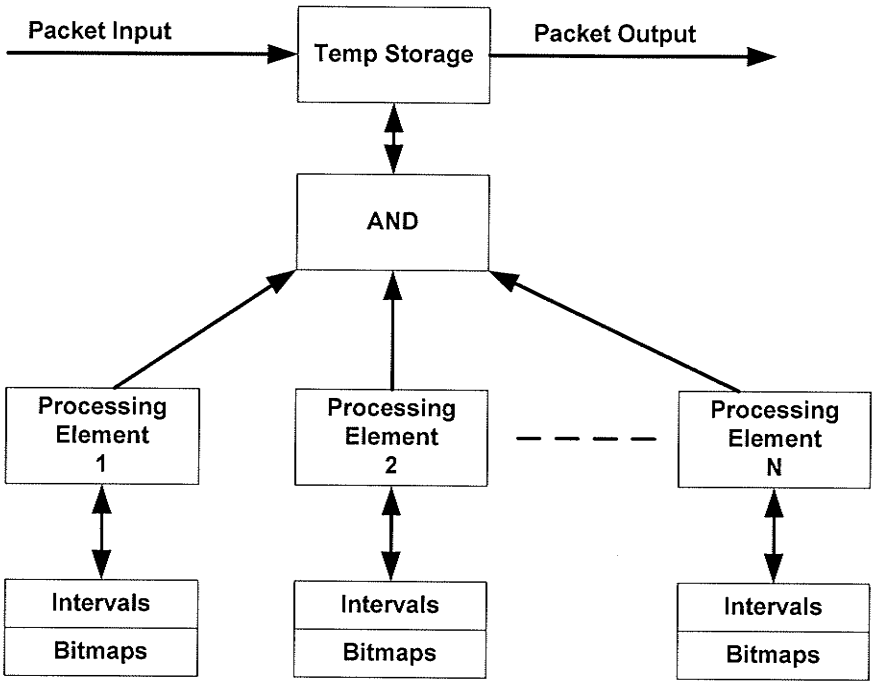


Figure 4-1: Parallel Implementation [1]

4.4.1.1 The Bit Vector Intersection Problem

Given a set of d binary vectors $v_1 \ldots v_d$ each of length l, the problem is to find which elements are positively common to all vectors. Each vector represents one of d dimensions, where d is an integer. The case d=1 is the trivial case and d=2 is the experimental case. This case has been selected for further testing because a significant amount of research has been performed in the area of 2-D classification, particularly for routing and IP forwarding. Figure 4-2 provides a simple example, intersecting two bit-vectors to find applicable rules.

Bit-Vector 1	0	1	0	.	.	.	1	0	1
Bit-Vector 2	0	1	0	.	.	.	1	1	0
Matches	0	1	0	.	.	.	1	0	0

Figure 4-2: Bit-Vector Intersection Example

With all applicable rules found, the one with the highest priority will determine the directive for the packet. The real problem is how to find the common pairs or elements, so to speak, in a timely and efficient manner. One approach is to use bit-level parallelism.

4.4.2 The Aggregated Bit-Vector Scheme

Baboescu and Varghese propose the Aggregated Bit-Vector scheme which provides enhancements to the Lucent Bit-Vector solution by making two distinct contributions, the recursive aggregation of bitmaps and filter rearrangement [2]. The paper suggests that it takes logarithmic time for many databases. Aggregation is used to reduce the memory accesses, based on the assumption that the number of rules set (i.e. asserted bits) in each bit-vector will be very sparse. In this scheme, each bit-vector is represented by an aggregate bit-vector with word size A. Each bit in this vector would then represent n/A elements in the original bit-vector. If nothing is set in that range, it contains a zero. The ABV then becomes the OR of the corresponding bits in the original bit-vector. The goal of this system is to effectively construct the bitmap intersection without looking at all of the leaf bitmap values for each field. This allows one to quickly filter out bit positions where there are no matches, thereby reducing the average search time. Rearrangement is further used to localize matches, creating fewer matches in the ABV. However, additional vectors must be stored to retain the mapping of priorities.

The time required for an insertion or deletion of a rule in this algorithm is $O(n)$ similar to the BV scheme. This is because the ABV is updated each time the associated bit-vector is updated. Therefore, updates can be expensive because adding a filter can potentially change the bitmaps of several nodes.

4.5 Packet Classification Tradeoffs

Now that some of the existing approaches to the packet classification problem have been summarized, it should be clear there are inherent tradeoffs amongst the complexity metrics outlined in Section 3.4. Table 4-1 attempts to compare the performance of some of the algorithms presented in literature according to these metrics. The ideal values of the metrics are also included to emphasize the deficiencies of existing solutions with respect to at least one metric. However, it should be noted, it is difficult to perform a fair comparison in this fashion due to the variability in implementation and simulation parameters, as well as the rule-sets used to benchmark each design.

Table 4-1: Comparison of Packet Classification Algorithms

Algorithm	Type	Worst –Case Search Time	Worst–Case Storage	Attributes	Ref
Linear Search	basic structure	$O(n)$	$O(n)$	Does not scale to large rule-sets in time.	[18]
GOT (2-D)	basic structure	$O(W^{d-1})$	$O(ndW)$	Does not do range matching. Is limited to prefix matching.	[8]
Hierarchical Tries	basic structure	$O(W^{d})$	$O(ndW)$	Does not do range matching. Is limited to prefix matching.	[8]
GEM	geometric	$O(\log n)$	$O(n^{d})$	Scales well on realistic rule sets but exhibits worst-case on random rule-sets.	[4]
Area-based Quad Tree (2-D)	geometric	$O(\alpha W)$	$O(n)$	2-D implementation where α is a tunable parameter used to trade look-up time for update time.	[9]
HiCuts	geometric/ heuristic	$O(d)$	$O(n^{d})$	Supports slow incremental updates but performance is dependent on heuristics.	[12]
Ternary CAMs	hardware	$O(1)$	$O(n)$	Brute force method that does not scale to large rule-sets with existing hardware. It also does not update well.	
Lucent BV Scheme	hardware/ geometric	$O(n)$	$O(dn^{2})$	Does not scale to large rule-sets. Allows search results from multiple dimensions to be merged together.	[1]
Lucent 2-D prefix scheme	hardware/ geometric	$O(ls + \log n)$ *ls* - tree height	$O(n)$	Special implementation based on tries and fractional cascading used to reduce search time.	[1]
ABV scheme	hardware/ geometric	$O(n)$	$O(dn^{2})$	Allows rearrangement of rules and tunable aggregate size *A*. Performs 10-times better than Lucent BV scheme on real rule-sets.	[2]
IDEAL		$O(1)$	$O(n)$	Incremental updates and high rule complexity.	

From a theoretical and computational perspective, the best algorithms for the general PC problem considering time and space require either $O\left((\log n)^{d-1}\right)$ time complexity with linear space complexity or $O(\log n)$ time complexity with $O\left(n^d\right)$ space [1]. Unfortunately, the worst-case space and memory costs are high. Consider an example of 10K rules where d=5 and the logarithms are of base 2. This will require a search cost of more than 31K memory accesses and linear memory or a search of around 13 memory accesses and memory cost of 10^{20} bits. Thus, it should be clear that the multi-dimensional PC problem is extremely difficult. It is unlikely that a fast and scalable solution exists for completely arbitrary multi-dimensional classifiers. Pankaj Gupta has made some insightful conclusions regarding this dilemma:

"The theoretical bounds tell us that arriving at a practical worst-case solution is not possible. No single algorithm will perform well for all cases. Fortunately, we don't have to; Real-life classifiers have some inherent structure that apparently we can exploit; A hybrid scheme might combine the advantages of several approaches." [12]

It is also the belief in this thesis that future PC algorithms capable of supporting emerging bandwidths will almost certainly be a hybrid technique building upon ideas from numerous previous works. An observation from Table 4-1 that supports this train of thought is that the generic worst-case bounds do not apply for $d < 3$. Overall, one- and two-dimensional schemes have been known to scale well. Some of these algorithms may achieve logarithmic time and linear storage using techniques such as range trees with fractional cascading [1], hierarchical tries [8], and Area-based Quad trees [9]. For instance, Grid-of-Tries performs well for two-dimensional prefix matching. Furthermore, even better results can be achieved using pre-computation, hashing and CAMs.

While algorithms failing to target the general PC problem are not particularly interesting, the main points are that solutions to one- and two-dimensional classification seem to be special cases, where any extension to higher dimensions will result in significantly slower performance and reduced flexibility of the algorithm. These realizations are applied as a new algorithm is developed in the next chapter.

Chapter 5: Design Specifications

Chapter 5 presents the fundamental data structures and searching techniques used to develop a novel packet classification algorithm in this thesis. The chapter begins with a brief justification of the design strategy. Next, the principles behind the hardware and software related structures are presented, including bit-vectors, hierarchical compression and B-trees (Sections 5.2 and 5.3). A full description of the system architecture follows in Section 5.4 with an accompanying example of the system pre-processing and query operations. Overall, the system strives to achieve a balance between hardware optimization and programmable flexibility.

5.1 Design Strategy

The approach taken in this thesis is to decompose the multi-dimensional problem into a set of two-dimensional problems each producing bit-vector solutions. The multi-dimensional query then becomes the bit-vector intersection of a few simpler two-dimensional queries. The main challenges in using this structure will be to find scalable solutions to the intersection problem and memory usage to meet the requirements of modern day networks. At present, no known algorithms exist that attempt to decompose the general PC problem into two-dimensional problems. However, solutions have been proposed which attempt to decompose the PC problem into groups of one-dimensional problems [1], [2].

The main benefit of decomposing the PC problem in this manner is that one can take advantage of existing two-dimensional schemes. A tunable framework can then be created that allows for the incorporation of various 2-D search algorithms into one system that will perform multi-dimensional classification. This is significant because different classification algorithms (search schemes) are optimal for particular applications depending on the size and complexity of the rule-set. In other words, some two-dimensional schemes are better suited for prefix matching while others are more amenable to range matching or exact matching. As an example, it would be highly desirable to use something like Grid-of-Tries [8] for two-dimensional prefix lookups. In this thesis, a B-tree search structure has been selected for the 2-D queries primarily because it can support each of the classification match types described in Section 3.2. B-trees also provide a good balance between search times and update performance and are known to perform well in systems with a large cache line, where an entire node can be returned in a single memory access.

5.2 Bit-Vectors

To support operations at wire-speed, an algorithm must be amenable to hardware implementation. This provides the advantages of hardware pipelining and parallelism, which are enablers in reducing the time complexity. As in the BV and ABV schemes, bit-vectors and bit-level parallelism are used to achieve these goals.

Bit-vectors are a common structure used to represent data sets and, in some instances, are often much more efficient than linked lists or arrays. In general, it is difficult to do better than one bit per object. In addition, all logical operations can be implemented using bit-wise operations, which are among the fastest. In particular, bit-vectors give excellent performance for operations like unions and intersections where it is easy to take advantage of bit-level parallelism.

The disadvantage of bit-vectors becomes clear when attempting to encode a database of rules that has potentially millions of entries. Conventional bit-vector representations need to reserve a significant amount of memory and are as wide as the number of distinct objects in the collection or rule-set [1], [2]. This can outweigh all of the advantages that bit-vectors offer. In the Lucent BV scheme, this means that the intersection will still take too long to operate at wire-speeds for large rule-sets regardless of any practical level of bit parallelism. The ABV scheme attempts to counter this by introducing aggregation and rearrangement so that the intersection does not have to occur on the entire bit-vector. However, using the conventional representation also means a great deal of storage space is required for the large number of bit-vectors. In both the BV and ABV schemes, the memory requirements fail to scale to large databases.

Consider a database of one million rules. One bitmap representing any aspect of the collection will take approximately 123K bytes. Storing only 100 or 1000 bit-vectors of that size will require 12 and 123 MB respectively. Obviously, most applications can't afford pure bit-vectors. Of course, when storing the bit-vectors in a database, some general compression method could be applied such as arithmetic coding or Huffman trees. Nevertheless, accessing the database through the compressed vectors would be very costly. If the decompression required involves numerous logical operations, the time required for processing could be prohibitive [24]. The objective is to effectively implement operations on bitmaps using on-the-fly adaptive compression without a significant performance sacrifice.

Fortunately, classification rule-sets do not keep random data. In all likelihood, these vectors are going to be extremely sparse, especially for two dimensions and above. In essence, there will be very few ones dispersed amongst a sea of zeros. Therefore, it is possible to exploit these features and write an efficient adaptive implementation of bit-vectors with embedded compression. Obviously, attempting to organize bit-vectors means there is a tradeoff between performance and space. However, with an efficient and flexible implementation, such a tradeoff can be affordable and acceptable.

5.2.1 Hierarchical Compression

Hierarchical compression is one technique that can naturally be applied to bit-vectors. In hierarchical compression, groups of bits are combined together into a tree-like structure of bit-vectors. A '0' (zero) at a top-level indicates that all descendants are also zero. A '1' indicates that the descendants contain some non-zero bits. In the first case, it is not necessary to keep the memory for lower levels, which results in a significant decrease of memory consumption. Consider representing a bitmap of 27 rules in hierarchical format with a branching ratio of three. Each bit in the top-level represents three bits in its descendents, providing two levels of hierarchy. In Figure 5-1, the shaded vectors are all that need to be stored, reducing the storage requirements from 27 bits to 12. While this may seem like a minimal savings, for large sparse bitmaps it can be huge.

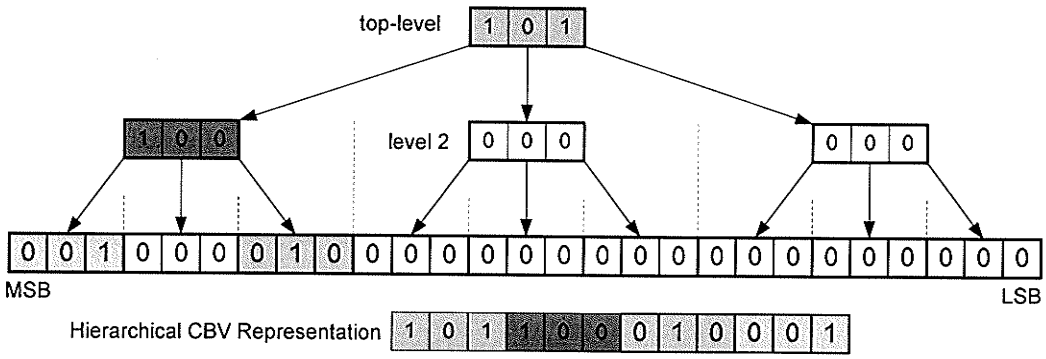


Figure 5-1: Hierarchical Compression of a Bit-Vector

The depth of the compression tree is kept low, so only a particularly low number of comparisons are needed to find a leaf and inspect a bit in it. In cases where the vector is sparse, only the root node needs to be inspected to determine whether the object is represented in the set. This is similar to the aggregated bit-vector of [2], except that rather than using one and two level aggregation to speed up the intersection, hierarchical compression is used to store the entire

vector and reduce memory use. There is no need to store the entire bitmap and, when a match is found, there is no need to go back and retrieve portions of the bitmap. Therefore, hierarchical compression is an efficient and flexible way to manipulate bit-vectors, making the performance and space tradeoff worthwhile.

5.3 The B-tree

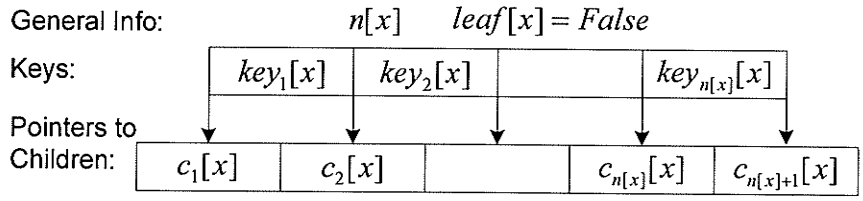
A B-tree is a rooted and balanced search tree, optimized for I/O operations. In a sense, B-trees generalize the simple binary search. That is, if an internal node x has $n[x]$ keys, then it will have $n[x]+1$ children. The keys are used as dividing points separating the range of keys handled by x into $n[x]+1$ sub-ranges, each of which are handled by the children $c_i[x]$, where $1 \le i \le n[x]+1$. Hence, when searching for a key in a B-tree, an $n[x]+1$ way decision is made. Some other distinguishing features of B-trees are that the branching factor can be quite large. As well, each n-node B-tree has a height of $O(\log_t n)$, where t is the minimum degree of the B-tree. Therefore, this structure can be used to implement many dynamic set operations in $O(\log_t n)$ time. A formal presentation of B-tree properties is provided below while further information on implementing B-trees operations can be found in [18].

5.3.1 B-tree Properties

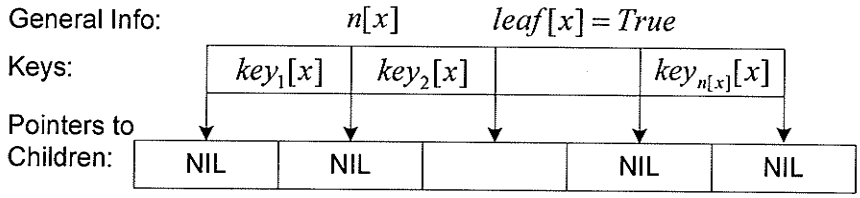
Each node in the tree must contain a few basic fields.

- The number of keys present in node x, represented by $n[x]$.
- The keys, stored in ascending order: $key_1[x] \le key_2[x] \le \cdots \le key_{n[x]}[x]$.
- A Boolean value that indicates if the node is a leaf or an internal node.
- Each internal node in the tree also maintains $n[x]+1$ pointers to the $c_1[x], c_2[x], \cdots, c_{n[x]+1}[x]$ children. For leaf nodes, this field is left undefined.

Figure 5-2 illustrates this basic information for internal and leaf nodes. It is also apparent from the diagrams that the keys separate the ranges of keys that are stored in each sub-tree (i.e. $k_1 \le key_1[x] \le k_2 \le key_2[x] \le \cdots \le key_{n[x]}[x] \le k_{n[x]+1}$).



a) Internal Node



b) Leaf Node

Figure 5-2: B-tree Node Fields: a) Internal Node b) Leaf Node

Some additional B-tree properties are:

- All leaves in the tree have the same depth, which is equal to the tree's height since it is a balanced tree.
- There are bounds placed on the number of keys a node can have. Typically, these bounds are expressed in terms of an integer $t \geq 2$. This is known as the minimum degree of the B-tree. Based on the degree of the tree, there are a couple of basic rules that nodes must adhere to. This is to ensure that the B-tree structure is never violated.

  a) Each node other than the root must maintain a minimum of t-1 keys. Consequently, every internal node other than the root will have a minimum of t children.

  b) Every node can contain a maximum of $2t-1$ keys (i.e. the node is full). This means that an internal node can have at most $2t$ children.

Thus, a B-tree with a minimum degree of 2 will contain internal nodes with 2, 3, or 4 children. An important feature of this structure is the worst-case height of the tree, since it plays a role in determining the overall performance of the search and update procedures. If $n \geq 1$, then for any n -key B-tree of height h and minimum degree $t \geq 2$ the worst-case height is defined by Equation 5-1. A formal proof for this statement is provided in [18].

Equation 5-1: Height of a B-tree

$$h \leq \log_t \frac{n+1}{2}$$

Figure 5-3 provides an example of a simple B-tree of degree 2 where the keys are integers from the set of natural numbers.

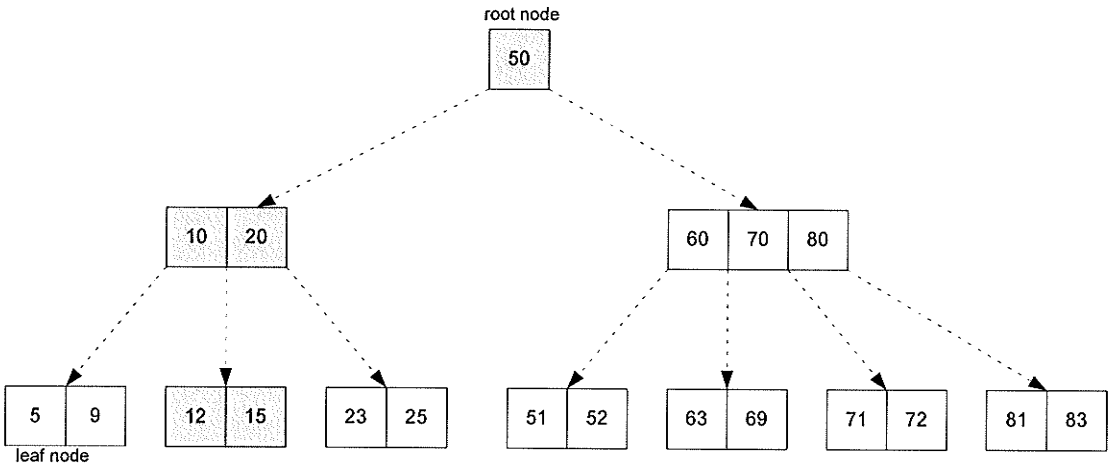


Figure 5-3: B-Tree Example ($t = 2$)

5.3.2 B-tree Operations

The fundamental operations that need to be performed with respect to the B-tree are its creation, insertion of keys, deletion of keys and searching. The B-tree search operation is very similar to searching a binary search tree. However, instead of performing a binary '0' or '1' branching decision at each node, a multi-way decision is made. The number of keys in the node determines the degree of this decision.

5.3.2.1 B-Tree Search

In general terms, the B-tree search receives a pointer to the root node of a sub-tree along with a key to look for. If the key is found, then the search returns a positive result containing the node and an index into the node where the key is located. If no match is found, a negative or null result is returned. Figure 5-3 provides an example of a search for the key 15. The lightly shaded nodes indicate which nodes would be inspected during the search. Since each node contains at most $2t-1$ keys, the time taken at each level of the tree is $O(t)$, and the theoretical worst-case search time is $O(th)$, where h represents the height of the tree. This can be re-written as $O(t \log_t n)$ based on Equation 5-1.

5.3.2.2 B-Tree Insert

Insertion of a key into a B-tree is slightly more difficult than inserting into a binary tree. In binary trees, one would search for the leaf position to insert the new key and then create a new leaf. For B-trees, this approach could lead to invalid data structures. Instead, the key must be inserted into an already existing leaf node. Furthermore, a new key cannot be inserted into a full leaf node, so a split operation may have to be exercised to maintain the tree structure. If there is a full node according to rule 2 of the B-tree properties, it should be split around its median into two nodes each having $t-1$ keys. The median value moves up the tree to its parent and separates the ranges for the two children. However, it is possible that the parent may be full, in which case it would require splitting. This process may propagate up the tree to the root. To avoid this condition, splits are performed on every full node encountered as the tree is searched for the insert location. Consequently, whenever a full leaf node needs to be split, there is a guarantee the parent is not full. Thus, a single insert can be performed in one pass down the tree and is an $O(\log_t n)$ operation. To build an entire tree with n insertions will take $O(n \log_t n)$.

From the above discussion, it is also clear that splitting is the only way in which the height of the tree may grow. Figure 5-4 illustrates the split operation of a node where t=4. The node y is split into two nodes and the median key (integer 5) is moved up into the parent's node.

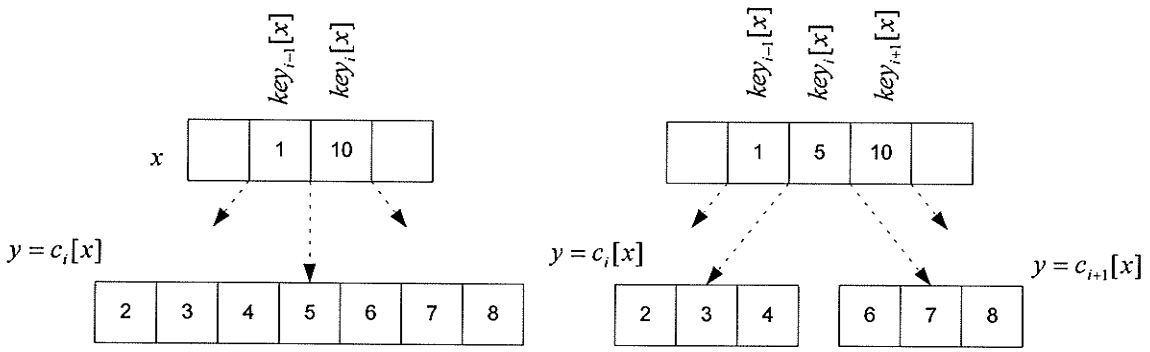


Figure 5-4: Splitting a B-tree Node of Degree t=4

The delete operation is similar to the insert but is slightly more complicated. For more information on this and other B-tree operations, the reader is directed to [18].

5.4 The CBV Packet Classification Algorithm

This section outlines the build and search algorithms for a packet classification scheme based on CBVs with a B-tree search. Figure 5-5 provides a block diagram for the system. Essentially, a number of two-dimensional queries occur in parallel, each returning bit-vector solutions, which are intersected to find the rule with the highest priority. For the two-dimensional queries, multiple B-trees of degree 3 are linked in one dimension with a set of B-trees and compressed bit-vectors in a second dimension. It should be noted that in this implementation the keys of the B-tree store ranges opposed to single values. As the build (pre-processing) and query operations are explained, the justification for this design will become clearer.

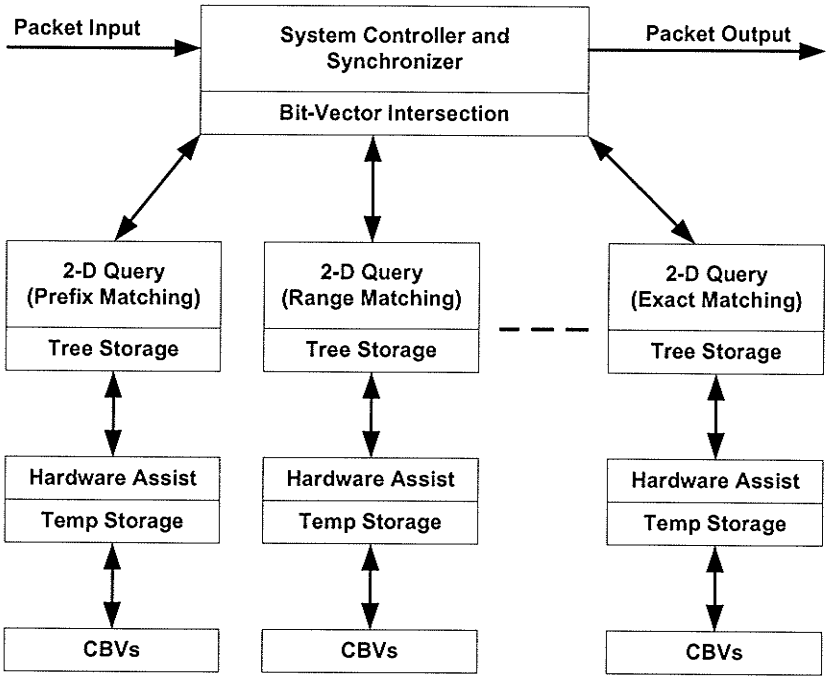


Figure 5-5: CBV Algorithm System Block Diagram

5.4.1 Inbound-Outbound Ratio

A typical firewall rule-set will consist of a mix of both inbound and outbound rules. In [15], it was discovered that the ratio of inbound to outbound rules significantly impacts the overall size of the search data structure. As expected, smaller data structures are produced when the rule-set is homogenous in nature (e.g. all *inbound* (IB) or *outbound* (OB) rules). For large rule-sets, the improved space performance may even warrant the implementation of two data structures, one for IB traffic and one for OB traffic. To take advantage of this observation, independent data structures are generated for all inbound and outbound rule-sets in this thesis. Given the

limitations of the testing methodology and the available memory on the prototyping environment, this will also allow testing of much larger rule-sets.

5.4.2 Algorithm Pre-processing

The build portion of the algorithm accepts three main inputs. The first is a set of n classification rules (inbound or outbound) defined as $R = \{r_1, ..., r_n\}$ in k dimensions. Therefore, each rule, r_m, in the set contains k fields representing the rule's range for each dimension. This set is represented by $r_m = \{e_{1,m}, e_{2,m}, \cdots, e_{k,m}\}$. The second input is the field order used to build each 2-D search tree in the structure. In [15], field order was found to have a significant impact on both search speed and size of the data structure. The third input, b, is known as a bucket factor and provides the algorithm with a space versus time tradeoff parameter. Based on this parameter, the rule-set can be partitioned into multiple sets which each produce a 2-D B-tree structure. A formal description of the build algorithm is provided below.

1. Partition the k fields defined by the rules into pairs, which shall be denoted by the set $P = \{p_1, p_2, \cdots p_j\}, 1 \le j \le \frac{k}{2}$, where $p_j = \{h_{j,1}, h_{j,2}\}$ and, $h_{j,1}$ and $h_{j,2}$ correspond to fields used to build the first and second dimension B-trees. Note: the selection of pairs should be made judiciously to ensure optimal searches and space requirements. If the number of dimensions k is not even, the remaining field will be searched in one dimension.

2. For each pair p_j, consider $h_{j,1}$, and generate b partial rule-sets $R_{j,b}, b \ge 1$. The criteria for partitioning the rule-set may vary but in this case, it is split based on range size. A rule r_m belongs to set $R_{j,b}$ if its range size in $h_{j,1}$ falls into the appropriate bin. Each new subset of rules will be used to generate a 2-dimensional B-tree structure, so the final data structure will contain multiple B-trees.

    - Ranges were selected as follows: [0x00000000:0x00000000FF], [0x00000100: 0x0000FFFF], [0x00010000:0x00FFFFFF] and [0x01000000: 0xFFFFFFFF]. This leads to 4 valid ranges for 32-bit fields, and 2 valid ranges for 16-bit fields.

3. The following steps are executed for each pair p_j with subset of rules $R_{j,b}$ $\forall j \in \left\{1...\frac{k}{2}\right\}, b \in \{1,2,3,4,...\}$:

   a. Project all intervals onto the $h_{j,1}$ axis. This will generate a maximum of $2n+1$ non-overlapping intervals along the axis [1]. These sets of intervals will be denoted by $I_{j,d,b}$, where $1 \le j \le \frac{k}{2}, b \ge 1$, and $d = 1$.

   b. For each interval $i \in I_{j,d,b}$ create sets of rules $R_{j,d,b,i}$, where $1 \le i \le 2n+1$, and $d = 1$ such that a rule $r_m \in R_{j,b}$ belongs to the set $R_{j,1,b,i}$ if and only if, the corresponding interval i overlaps with $e_{d,m}, 1 \le d \le k$. Recall $e_{d,m}$ is the element in r_m which maps to $h_{1,j}$ according to the division of the header fields. Each interval will then correspond to a key in the B-tree and its subset $R_{j,1,b,i}$ of $R = \{r_i, ..., r_n\}$ will be used to create a B-tree in the second dimension along $h_{j,2}$. A pointer to the secondary B-tree must be maintained in the first dimension along with the key.

   c. Creation of the secondary B-trees is similar to that of the first dimension. Project all non-overlapping intervals from $R_{j,d,b,i}$ where $1 \le i \le 2n+1$ and $d = 1$ onto the $h_{j,2}$ axis. Each set will again generate at most $2n+1$ non-overlapping intervals. The sets of these intervals will be denoted by $I_{j,d,b}$, where $1 \le j \le \frac{k}{2}, b \ge 1$ and $d = 2$.

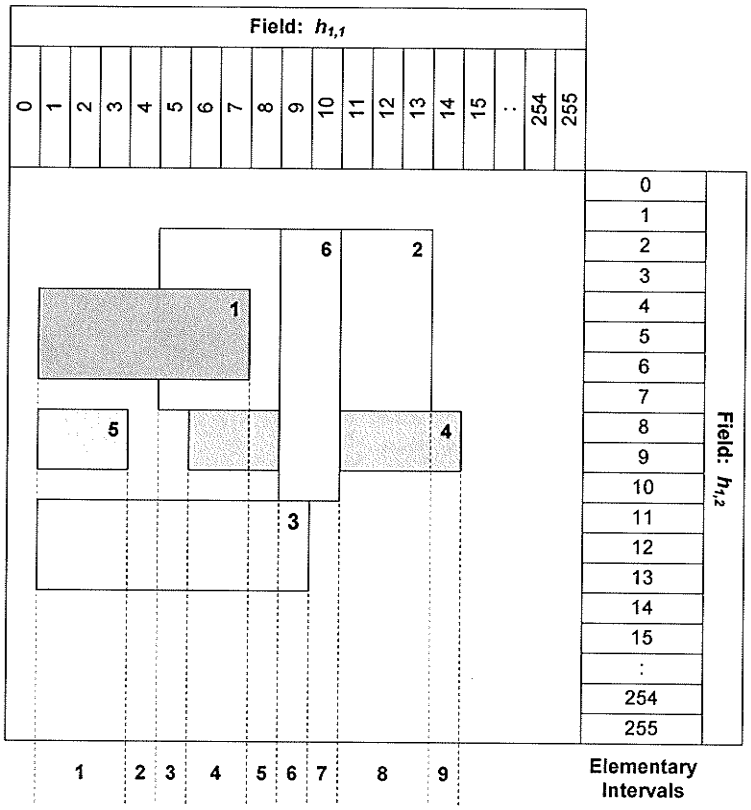
   d. For each interval $i \in I_{j,d,b}$, and $d = 2$, create sets of rules $R_{j,d,b,i}$, where $1 \le i \le 2n+1$, such that a rule $r_m \in R_{j,b}$ belongs to the set $R_{j,2,b,i}$ if, and only if, the corresponding interval i overlaps with $e_{d,m}$, where $1 \le d \le k$ and $e_{d,m}$ is the element in r_m, which maps to $h_{j,2}$.

e. The sets of rules produced in the second dimension are used to create compressed bit-vectors using hierarchical compression. To create the CBVs, the rules are written to a hardware-optimized *Packet Filter Assist Acceleration Engine* (PFAAE). The hardware creates the hierarchical representation, stores it in memory and returns a pointer to the location of the CBV.

To demonstrate the basic pre-processing steps an example rule-set is shown in Table 5-1 for a single pair of 8–bit fields. This rule-set is also used to illustrate the algorithm search operation in Section 5.4.3. As seen in Table 5-1, the rule-set has been split into 2 buckets creating two rule-subsets, $R_{j,b}$, where $j = 1, 1 \le b \le 2$. The first rule-subset, $R_{1,1}$, contains rules with an $h_{1,1}$ field range between 0x00 and 0x0F. The second subset, $R_{1,2}$, contains rules with an $h_{1,1}$ range between 0x10 and 0xFF. Figure 5-6 illustrates the geometric representation of the first subset, $R_{1,1} = \{R1, R2, R3, R4, R5, R6\}$ along with pre-processing steps (3a) and (3b).

Table 5-1: Example Rule-Set in Two Dimensions with 27 Rules

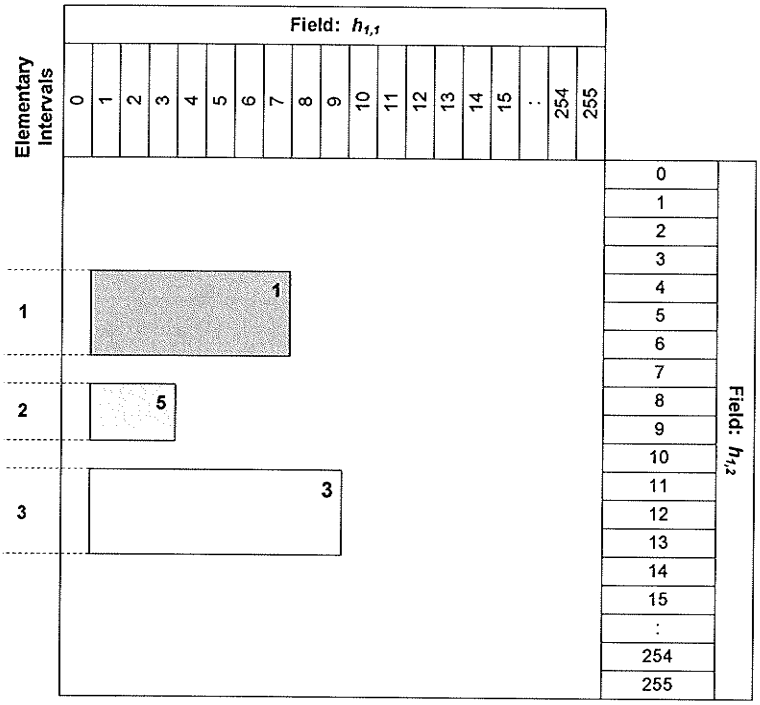
Rule ID	$h_{1,1}$	$h_{1,2}$	$R_{1,b}$,b=
R1	1-7	4-6	1
R2	5-13	2-7	1
R3	1-9	11-13	1
R4	6-14	8-9	1
R5	1-3	8-9	1
R6	9-10	2-10	1
R7	0-31	4-15	2
R8	8-31	32-63	2
R9	32-63	0-7	2
R10	64-127	16-31	2
R11	128-255	192-192	2
:	:	:	:
R27	0-127	0-31	2



Interval $i \in I_{1,1,1}$	Interval Range	Rules in Set $R_{1,1,1,i}$
1	1-3	R1,R3,R5
2	4-4	R1,R3
3	5-5	R1,R2,R3
4	6-7	R1,R2,R3,R4
5	8-8	R2,R3,R4
6	9-9	R2,R3,R4,R6
7	10-10	R2,R4,R6
8	11-13	R2,R4
9	14-14	R4

Figure 5-6: Geometric Representation of Rule-Subset, $R_{j,b}, j=1, b=1$ and Table of Elementary Intervals in the 1st Dimension

In Figure 5-6 the elementary intervals and rule-subsets are generated for the first dimension of the 2-D B-tree. This would result in a simple B-tree containing nine keys with nine rule-subsets $R_{j,1,b,i}$, where $j=1$, $b = 1$ and $1 \le i \le 9$. Next, a B-tree is generated in the second dimension from every rule-subset, $R_{1,1,1,i}$, and a pointer is attached to the corresponding key in the first dimension. For example, the subset corresponding to elementary interval one $i_1 \in I_{1,1,1}$ is used to generate the non-overlapping intervals on the $h_{1,2}$ axis as shown in Figure 5-7.



Interval $i \in I_{1,2,1}$	Interval Range	Rules in Set $R_{1,2,1,i}$
1	4-6	R1
2	8-9	R5
3	11-13	R3

Figure 5-7: Geometric Representation of Rule-Subset $R_{1,1,1,1}$ and Table of Elementary Intervals in the 2nd Dimension

From this procedure, there are three intervals created, which translate into 3 keys and rule-subsets. For a B-tree of degree 2, this would lead to one root node. Once these sets have been created in the second dimension, the rules are passed to custom hardware blocks, which will create the compressed bit-vectors and return pointers in their place.

5.4.2.1 Build-Time and Space Complexity

For the software tasks, the theoretical build-time and space complexity of a 2-D B-tree are easy to see. It is an $O(n \log_t n)$ operation to build the tree and requires $O(n)$ space to store the tree in each dimension. Thus, the total build-time complexity is $O\left(n^2 (\log_t n)^2\right)$ and the space complexity is $O(n^2)$. For the hardware operations the worst-case complexities to build and store each CBV are $O(n)$. Therefore, the combined data structure has overall build-time and space complexities of $O\left(n^3 (\log_t n)^2\right)$ and $O(n^3)$ respectively. Previous works would have dismissed this algorithm as unfeasible due to the large space requirement. However, by introducing bit-level parallelism, judicious decision of field order and space versus time tradeoffs such as the

bucket factor b, the findings in this thesis show that the algorithm is in fact a practical alternative when used with rule-sets based on real firewall statistics.

The bucket factor is a space-time trade-off similar to one used by GEM, where the rule-set is partitioned into b subsets and separate trees are built from each subset. The impact of splitting the rule-set equally into b buckets is a constant reduction in the software related space complexity $O\left(b\left(\frac{n}{b}\right)^2\right) = O\left(\frac{n^2}{b}\right)$ at the expense of search speed. If the rule-set is split according to some other criteria such as ranges, the exact impact is more difficult to predict. In this case, much more work needs to be done in order quantify the gains.

5.4.3 On-line Processing

Once the B-tree search structures and compressed bit-vectors have been created, search operations can be performed. To classify a packet (i.e. perform a query), it is assumed that rules are sorted based on priority, and a packet arrives in the system with a set of k generalized header fields. The search is performed as follows:

1. The relevant packet header fields are extracted and the appropriate header field pairs are sent to their respective two-dimensional query engines (i.e. the pair of headers corresponding to $p_j = \{h_{j,1}, h_{j,2}\}$ should be sent to the appropriate search engine). B-tree searches are performed in parallel for each 2-D query.

2. For each pair $p_j = \{h_{j,1}, h_{j,2}\}$, $1 \le j \le \frac{k}{2}$, the intervals $i \in I_{j,d,b}$, $d = 1$, $\forall b \in \{1,2,3,4,...\}$ that belong to the field $h_{j,1}$ are found in the first dimensional B-trees. Their pointers are followed to the second dimension where the appropriate intervals are again retrieved. Each search will return up to a maximum of b CBV pointers that are passed to the hardware.

3. For each 2-D query, the hardware retrieves and performs an ORing operation on the CBVs from step 2. The result is one CBV containing all possible filter matches across two dimensions.

4. The system controller must perform an intersection among all sets of applicable rules returned (i.e. on the ORed compressed bitmaps). This can be performed through a logical AND of the bit-vectors and is pipelined with the OR operation.

5. The rule corresponding to the highest priority is applied to the packet.

To illustrate the querying function of the algorithm, consider a two-dimensional example with a B-tree of degree 2 (i.e. nodes may have 1,2 or 3 keys) and a bucket factor of 2. This is a special case because there is only one pair p_1 and no intersection is required. It is also noted, that with regards to the actual implementation a B-tree of degree 3 and bucket factor of 4 are used. The rules for this example are expressed in Table 5-1.

The first step to performing a query with this architecture is to search the two-dimensional trees. Figure 5-8 illustrates a portion of the B-trees generated from the first bucket of rules (i.e. $R_{j,b}, j=1, b=1$). It also demonstrates the first steps of the software search procedure. The lightly shaded nodes indicate the path of nodes and keys examined in a search for incoming packet with $p_1 = \{h_{1,1}, h_{1,2}\}=\{4,9\}$, starting from the root. At each node the keys are examined left to right until one is found that contains 4 or is greater than 4. In the root node the first key is greater than 4 so a node pointer is followed left where the first key contains 4. Once the target key is found, a pointer is followed to a second dimension B-tree. The second key in the root node of this B-tree matches the key $h_{1,2} = 9$. In the diagram, it is indicated that rule 5 belongs at this node. However, this information will be stored in a CBV and only a pointer will be maintained at the node. Figure 5-9 reflects the representation of the bit-vector stored in memory. It is noted that in practice, some slight overhead (64-bits) is also stored to indicate the counts for level 2 and level 3 vectors. When the CBV pointer is received by the hardware-processing element, it will retrieve this vector and wait for the next pointer.

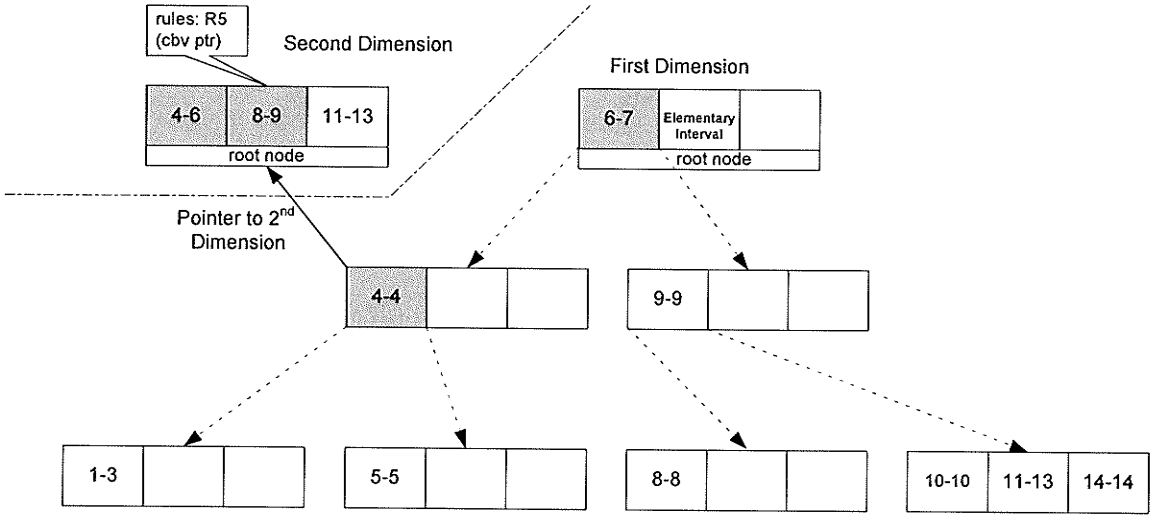


Figure 5-8: B-trees Generated from the First Subset of Rules (i.e. Bucket 1)

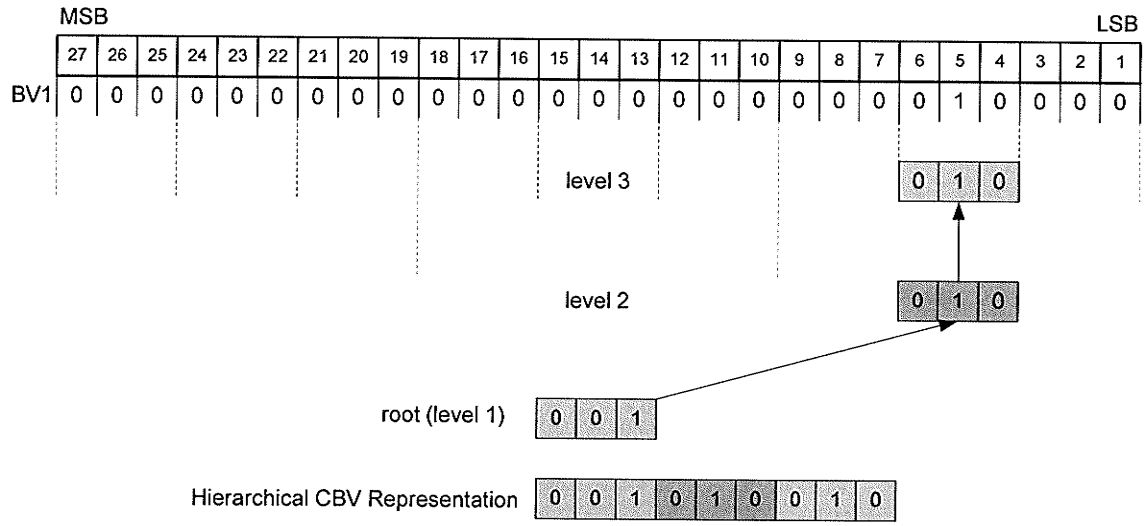


Figure 5-9: CBV 1 Representation in Memory

Figure 5-10 illustrates a portion of the B-tree generated from the second subset of rules (i.e. $R_{j,b}, j = 1, b = 2$) and reflects the next stage of the query. Once again, the lightly shaded nodes indicate the path examined in the search for an incoming packet with $p_1 = \{h_{1,1}, h_{1,2}\} = \{4,9\}$. The key of 4 is found in the second node and points to a second dimension B-tree. The second key in the root node of this B-tree contains the key $h_{1,2} = 9$. The diagram below indicates that rules 7 and 27 belong at this node (i.e. overlap this region). However, as before, only a pointer would be stored here. This pointer is returned to the hardware to determine all applicable rules which match the input. Figure 5-11 shows the representation of this bit-vector in memory.

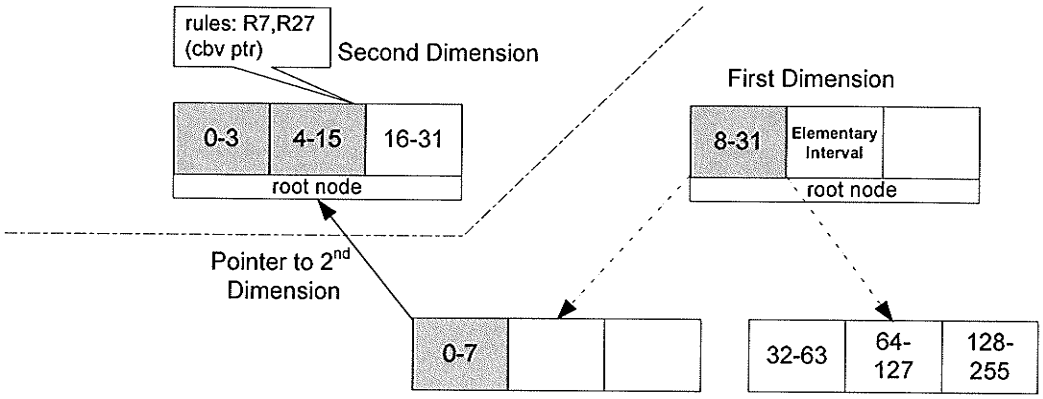


Figure 5-10: B-trees Generated from the Second Subset of Rules (i.e. Bucket 2)

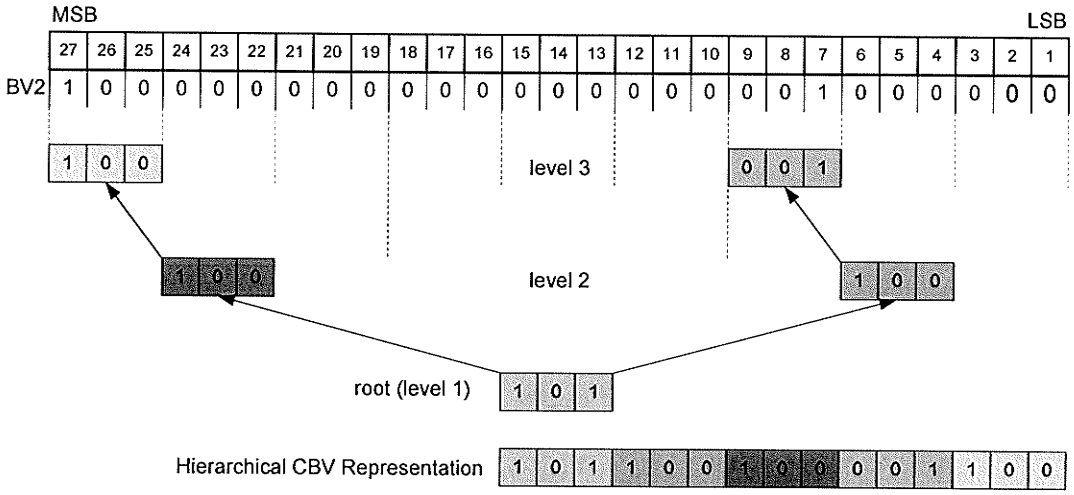


Figure 5-11: CBV 2 Representation in Memory

Once the hardware processing unit receives the second pointer, it retrieves this vector from memory and performs an OR operation on the two vectors. If a higher bucket factor is used, then additional bit-vectors may be simultaneously ORed together to produce one final search result. The ORing procedure for this example is illustrated in Figure 5-12 as if it were to occur on regular and compressed bit-vectors. In some cases, this process may involve partially decompressing the vector.

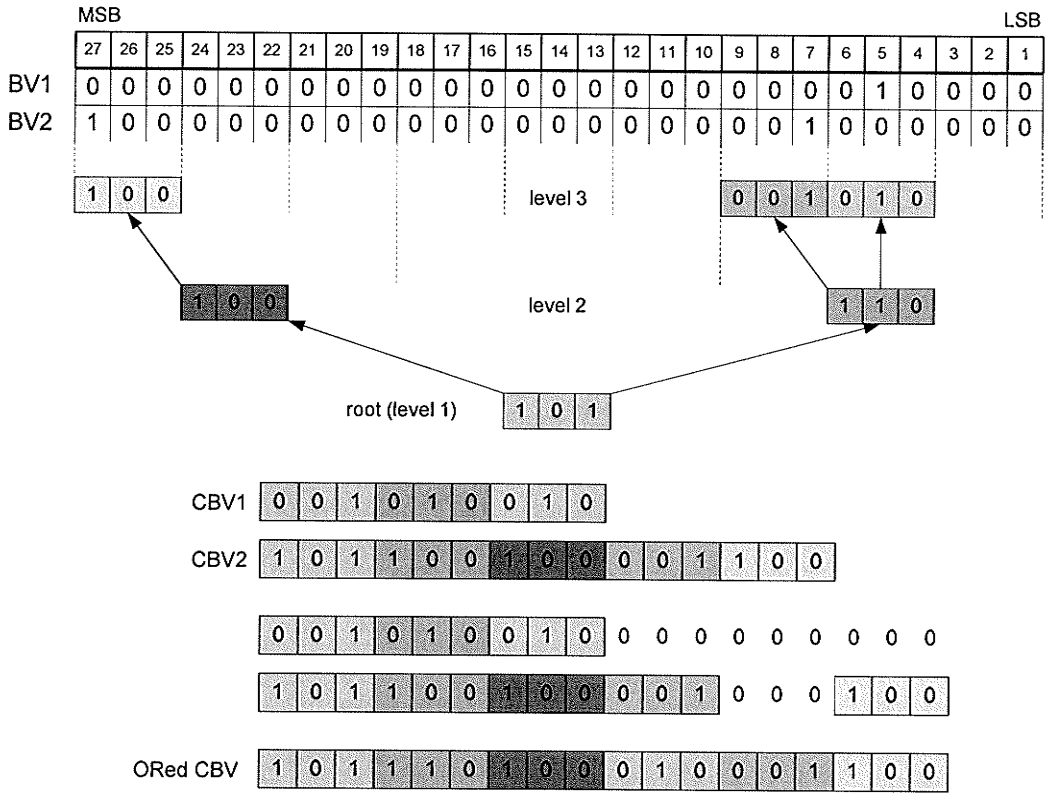


Figure 5-12: Compressed Bit-Vector OR Operation

This example has illustrated the query operation on two fields. If the query were performed on more than two fields, all the two-dimensional queries would occur in parallel. This system would also require an additional intersection operation. Once the ORing operations are complete, each two-dimensional system would return a compressed bit-vector with all applicable rules. The main controller in Figure 5-5 would be responsible for performing the intersection of these vectors through a logical AND operation. No additional time would be required because this operation would be pipelined with the OR operation from each of the 2-D queries.

5.4.3.1 Search Time Complexity

The search complexity for the software portion of the algorithm is the time taken at each level of the tree. This is $O(t)$ for a B-tree of degree t, which leads to a total search time of $O(th)$ where h is the height of the tree. Based on Equation 5-1, this can be re-written substituting for h. The search complexity becomes $O(t \log_t n)$ since $h \le \log_t\left(\frac{(k+1)}{2}\right)$, where k is the number of intervals and there are at most $(2n+1)$ non-overlapping intervals for n rules. Since there is a B-tree in each dimension, increasing by a factor d for the number of dimensions

leads to $O(d \times t \log_t n)$. However, for a constant t and d this can be simplified to $O(\log_t n)$. If a bucket factor is implemented, splitting the tree into b groups of size $\frac{n}{b}$, then b 2-D B-tree structures are created which leads to a slowdown in the software search speed of: $O(b \times d \times t \log_t \frac{n}{b})$.

For the hardware side, the on-line processing involves the CBV OR and intersection tasks among potential solution sets of classification rules. These are pipelined $O(n)$ operations since the potential sets may have cardinality of $O(n)$, when arbitrary overlaps are permitted. Therefore, this leads to an overall time complexity of $O(n)$. However, in practice, the execution time is largely accelerated since the hierarchical compressed vectors can quickly filter out blocks of rules which do not need to be examined. Bit-level parallelism can also improve the worst-case performance by operating on large blocks of memory or bit-vectors. A convenient approach is to tie the branching factor of the hierarchical compressed bit-vectors to the size of memory words of width w. By increasing w, the worst-case execution time is reduced as are the number of memory accesses.

5.5 Summary

Overall, by decomposing the search and using bit-vectors, a novel algorithm has been introduced which can support an arbitrary number of dimensions and is well suited for a System-on-Chip implementation containing processors, custom computing hardware and embedded memory. The query operations can be implemented using an existing processor and high-speed memory while compression and intersection components of the architecture can be implemented in custom hardware to exploit pipelining, parallelism and unique computational operations. As well, the use of bit-level parallelism should significantly accelerate the filtering operation for any practical implementation.

Chapter 6: Architectural Mapping and Design Partitioning

Chapter 6 presents the architectural mapping and partitioning phase of the design. During this stage, various architectural design options were evaluated. A *platform-based design* (PBD) approach was employed to reduce development time. PBD takes advantage of pre-designed architectural environments to facilitate reuse in the design and manufacturing of SoC applications in consumer-driven markets [29], [30].

The rapid prototyping platform selected for the CBV packet filter design is based on ARM Integrator series products and was provided by the *Canadian Microelectronics Corporation* (CMC) as part of the *System-On-Chip Research Network* (SOCRN) program. It is intended as a high-capacity FPGA-based hardware board with real-world interfaces for the purposes of IP evaluation, validation and system emulation. A picture of the overall system is shown in Figure 6-1.

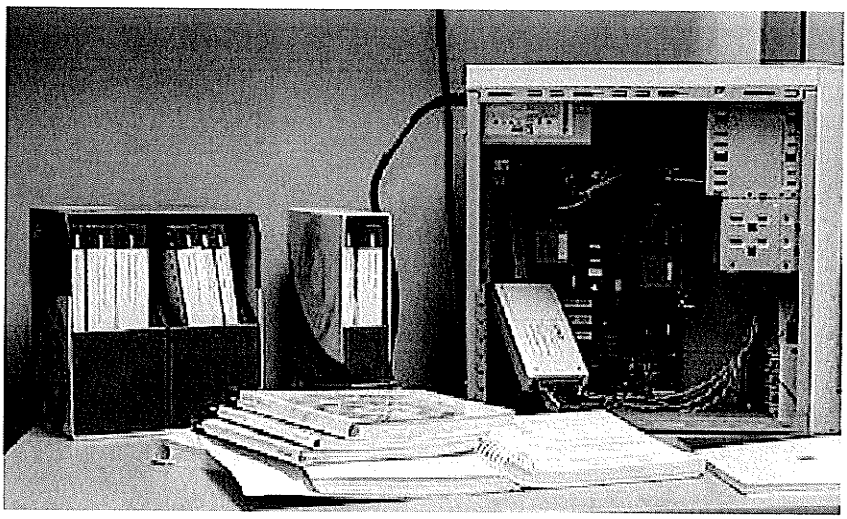


Figure 6-1: ARM Integrator Rapid-Prototyping Platform [42]

The RPP comes completes with hardware and software IP libraries and reference designs, all driving forces in the SoC revolution. The first three sections of this chapter elaborate on the target platform architecture, its capabilities, and the expansion requirements necessary for implementing a portion of the CBV packet classification algorithm.

Development of a functional prototype on a platform provides the benefits of fast verification of the overall system before committing to silicon as well as a proof-of-concept. In addition, it greatly simplifies the architectural mapping stage of the design cycle since the basic building blocks are already in place. Therefore, the main task is mapping functional units onto architectural units. This step is described in Section 6.4. The disadvantage of this approach is that it reduces flexibility and performance, constraining the designer to the limitations of the selected platform.

6.1 ARM Integrator/AP ASIC Development Platform

The ARM *Integrator ASIC Development Platform* (Integrator/AP) baseboard facilitates hardware and software co-development of devices and embedded systems based around ARM7TDMI microprocessor core modules and Xilinx FPGA logic modules or *logic tiles* (LTs). Figure 6-2 provides a close-up picture of the development platform and enhancements.

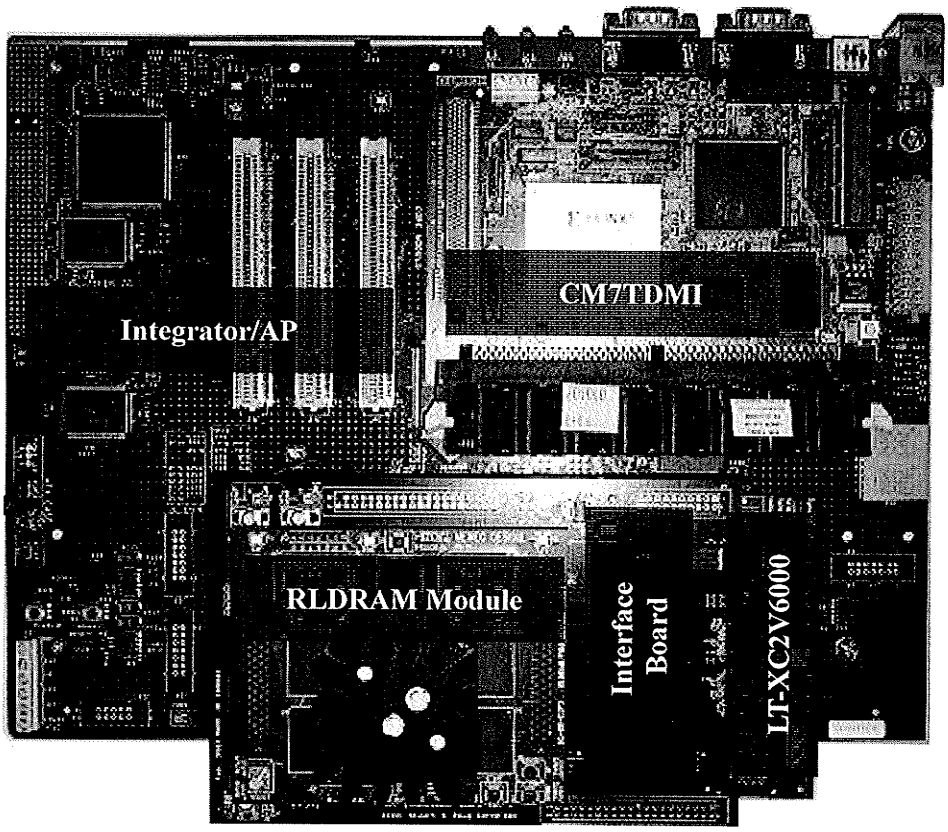


Figure 6-2: ARM Integrator Rapid-Prototyping Platform Hardware

The main hardware components and development tools associated with the platform are:

Hardware:

- ARM Integrator/AP ASIC development motherboard
- ARM CM7TDMI (ARM7 microprocessor) daughtercard
- ARM LT-XC2V6000 logic tile daughtercard
- ARM LM_LTI module
- ARM Multi-ICE (*in-circuit-emulator*) unit (debug)
- 128 MB DIMM Extra memory (expandable to 256MB DIMM)

Software:

- ARM Development Suite (ADS)
- Mentor VRTX Development System for ARM
- Mentor XRAY RDI for ARM

The Integrator/AP is quite versatile and allows the addition of *Peripheral Component Interconnect* (PCI) cards as well as the stacking of multiple core and logic tiles. The motherboard, logic modules and the core module communicate with each other and to peripherals using the ARM AMBA bus protocol [26]. Developers may even attach their own custom boards to the design, thereby enabling the prototyping of very complex SoC designs.

Unfortunately, due to hardware availability, the platform configuration was limited to a single ARM7TDMI and logic tile system. However, a P160 RLDRAM module was connected to the logic tile through a custom interface board to provide some high-speed memory for CBV storage. The addition of the P160 module provides an additional 128 MB of RLDRAM and a faster speed grade Xilinx FPGA.

6.1.1 Integrator/AP

The Integrator A/P motherboard is the main board for the platform [25]. The board features a system controller FPGA, a clock generator, 32 MB flash memory, 256 KB boot ROM, 512 KB SSRAM, two serial ports, and expansion connectors. The Integrator/AP can be enhanced by stacking up to four logic tiles or core modules up to a combined maximum of five. The Integrator/AP also has a PCI host bridge, which connects three PCI slots to the AMBA bus that is used in the rest of the system. The PCI slots allow the system to be expanded depending on the

application. There is even a PCI/PCI bridge, which maps a Compact PCI slot onto the system bus. Finally, there is an *External Bus Interface* (EBI) to allow additional memory to be added to the development system. A block diagram for the Integrator/AP board is shown in Figure 6-3.

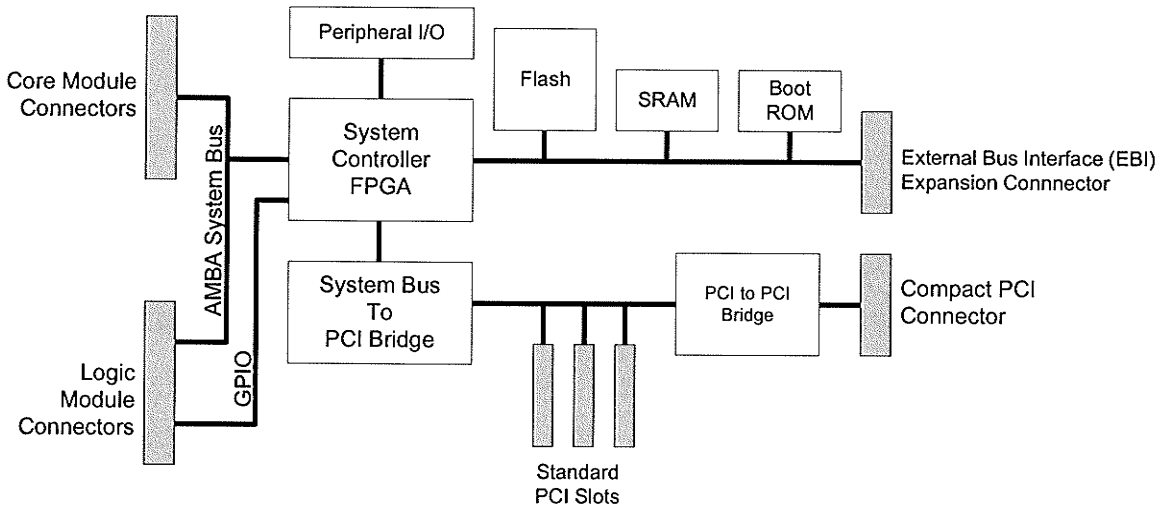


Figure 6-3: Integrator/AP Motherboard [25]

The system controller FPGA is pre-programmed to implement system controller functions for the rapid prototyping platform based on the AMBA protocol. The FPGA contains the system bus interface and arbiter, an interrupt controller, peripheral I/O controllers, three counters and timers, a reset controller and basic system status and controller registers. The system bus interface provides arbitration for the board as well as any connected modules. It supports transfers to and from the PCI bus, EBI and APB peripherals contained within the FPGA. The system bus arbiter also provides arbitration for up to five bus masters from logic or core modules and one from the PCI bridge. The interrupt controller handles interrupts for up to four ARM processors. Interrupts may originate from peripheral controllers, the PCI bus, or from devices on any of the logic tiles. Registers within the FPGA are used to enable, acknowledge and clear interrupts.

For system input/output (I/O), the Integrator/AP FPGA incorporates controllers for a *keyboard and mouse interface* (KMI), two UART serial ports, LEDs, alphanumeric displays, three 16-bit counters/timers, GPIO and a *Real-Time Clock* (RTC). These are all accessible from either a logic or core module through the system controller. For more information on the Integrator/AP FPGA, the reader can refer to [25].

6.1.2 Integrator/CM7TDMI

The Integrator/CM7TDMI module, or core module, is the main controller in the system [27]. The board features an ARM7TDMI microprocessor core, an FPGA, a *Dual In-line Memory Module* (DIMM) socket with support for up to 256 MB of SDRAM, a SSRAM controller with 256KB of SSRAM, a clock generator, system bus connectors, as well as Multi-ICE (*in-circuit-emulator*), logic analyzer and Trace connectors. Figure 6-4 provides a block diagram of the core module.

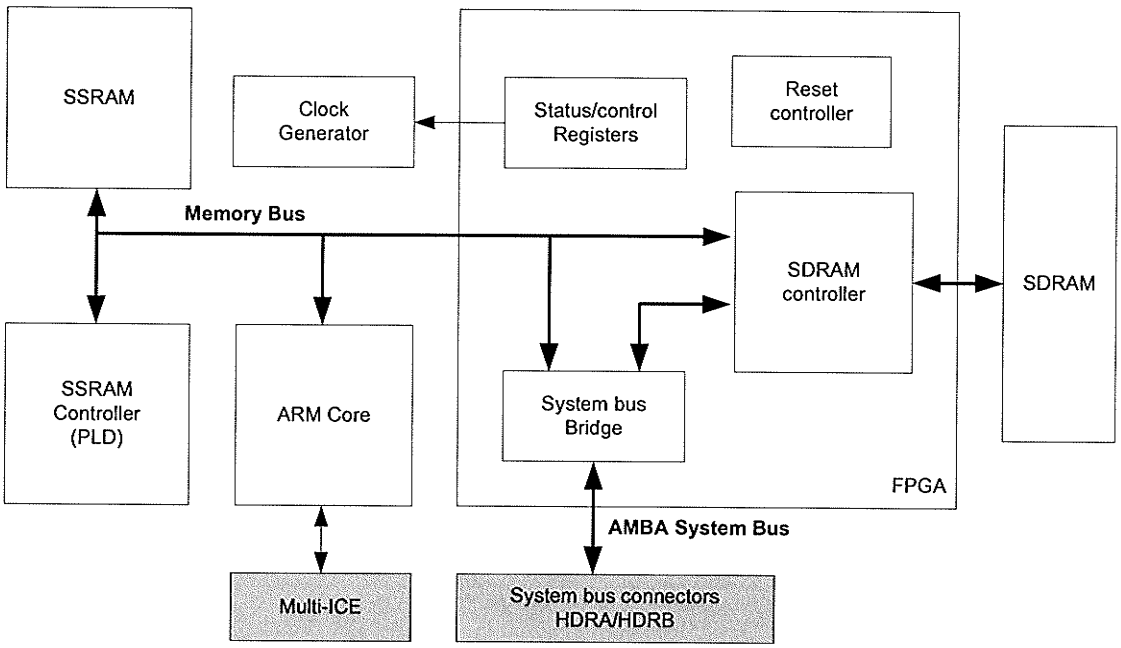


Figure 6-4: Integrator/CM7TDMI Core Module [27]

Like the Integrator/AP FPGA, the core module FPGA is already dedicated to perform system control functions. It is used to implement an SDRAM controller, system bus bridge, reset and interrupt controllers, as well as status, configuration and interrupt registers. For more information on the core module FPGA the reader can refer to [27].

6.1.3 Integrator/LT-XC2V6000

The Integrator/LT-XC2V6000 module or logic tile (LT) provides a jump-start for developing AMBA peripherals or custom logic for use with ARM cores [28]. It provides uncommitted FPGA resources and comes complete with an AMBA AHB reference design. To communicate over the system bus, the logic module must have an AMBA controller synthesized into the design. Figure 6-5 shows a block diagram of the LT architecture.

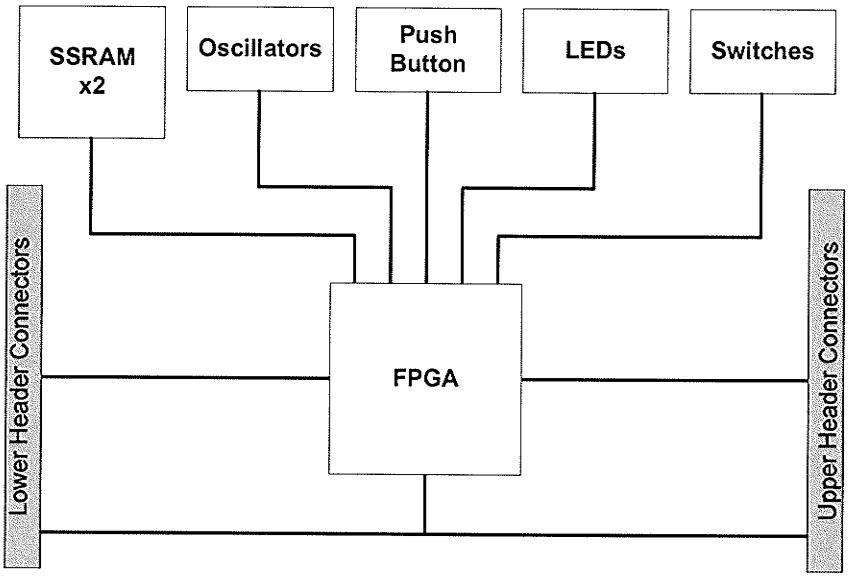


Figure 6-5: Integrator/LT-XC2V6000 Logic Tile [28]

The logic tile used in this research features a Xilinx Virtex II FPGA (XC2V6000-4FF1517). The majority of the I/O is routed directly to high-density connectors to support tile interconnection. On the module itself the FPGA directly interfaces with 3 programmable oscillators to control the clock frequency of the design, 2MB of external ZBT SSRAM, 8 MB of FLASH, as well as switches and LEDs that can be configured for system input and output. There is also a configuration PLD that loads images from the flash memory into the FPGA on reset. Since there is no JTAG connector on the logic tile itself, the Multi-ICE header on the interface board must be used to program the flash or the FPGA directly through the high-density connectors.

6.1.4 Interface Module IM-LT1

The IM-LT1 Interface module provides an interface between the high-density logic tile connectors and the Integrator AP motherboard. Its main purpose when connecting a motherboard to the logic tiles is to provide a 20-way box header for connection of JTAG run control units, such as RealView Multi-ICE.

6.2 P160 RLDRAM Module

The P160 RLDRAM module available from Memec Design provides an FPGA-based environment for evaluating RLDRAM. Figure 6-6 provides a block diagram of the RLDRAM development module.

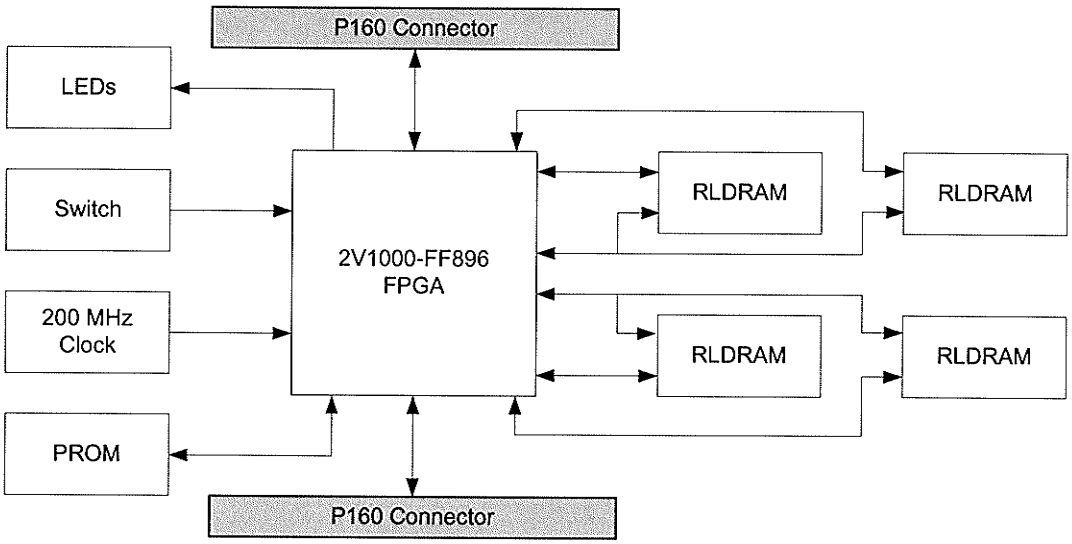


Figure 6-6: RLDRAM P160 Module Block Diagram

The key features of the RLDRAM module include a Xilinx Virtex II FPGA (XC2V1000-5FF896), four 256 Mbit Infineon Technologies' HYB18RL25632 32-bit wide RLDRAMs, an onboard 200 MHz LVDS clock, a physical interface which provides up to 400 Mbps *double-data-rate* (DDR) performance, and a standard P160 interface for expansion. The module was also delivered with a Memec Design RLDRAM controller in a black-boxed fixed netlist format. Designers can utilize Memec's proven RLDRAM controller to evaluate the performance of the RLDRAM technology or build and evaluate their own applications around the controller. The latter approach is taken in this research where the focus is to evaluate applications for the end market rather than devoting time and resources creating new IP solutions for a memory controller.

6.2.1 RLDRAM Technology

RLDRAM was co-developed by Infineon Technologies and Micron Technology to address memory density and performance-critical features needed by network applications. RLDRAM offers high density, extremely high bandwidth, reduced cycle time, and SRAM-like access. The internal architecture allows random access with row cycle times down to 20 ns. This is significantly faster when compared to standard DRAM row cycle times of 40 ns and above.

These characteristics position RLDRAMs as an ideal candidate for switch and router applications among other high-bandwidth, high-speed, and latency-sensitive applications and are the main reasons why RLDRAM was selected for use in the CBV packet filter [35].

6.3 P160 to Integrator/ LT-XC2V6000 Interface Board

A custom P160 to Integrator/LT-XC2V6000 interface board was developed to provide a compatible interface between the high-density connectors on the logic tile and the standard P160 RLDRAM module interface. Figure 6-7 provides a picture of the interface board. A P160 module specification, which defines an open standard for connecting expansion modules to existing P160 circuit boards, was loosely followed in the development [41]. The specification defines two 80- pin low insertion force connectors that connect the P160 module to a main board as well as the mechanical dimensions for the P160 I/O connectors. The details for the logic tile interface are provided in [28].

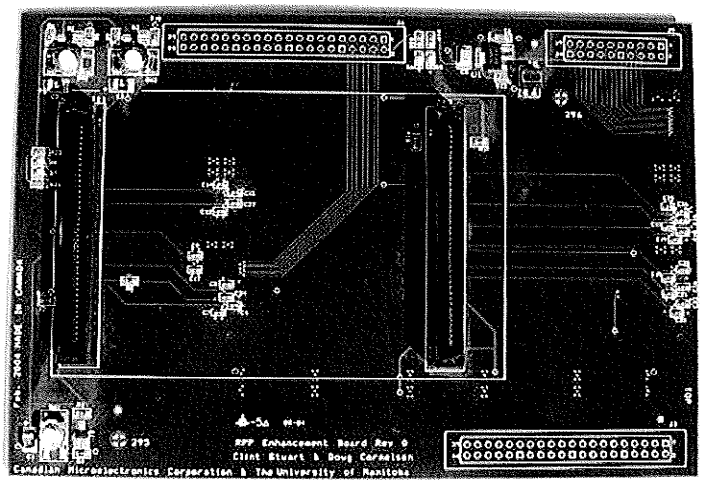


Figure 6-7: P160 to Integrator/LM-XCV6000 Interface Board

The key features of the board are a P160 standard interface, a logic tile compatible interface and a few general purpose LEDs and pushbuttons. To maximize the throughput between the two boards, the I/O traces were routed based on *low voltage differential signaling* (LVDS) standards.

6.4 SoC Blocks for CBV Packet Classification Algorithm

By combining knowledge of the platform architecture, system specifications, and IP libraries, the necessary inputs are in place to map and partition the functional behaviors of the packet filter design to platform resources. As mentioned previously, the platform was limited to a single core module and logic tile. This limits the implementation to one 2-dimensional search structure. Nonetheless, with the exception of the final intersection operation the implementation is still very representative of the overall design. All of the potential 2-D pair queries can still be built and performance analyzed on the platform. Figure 6-8 provides a block diagram of the design as it is partitioned across the RPP motherboard, daughtercards and the RLDRAM expansion module.

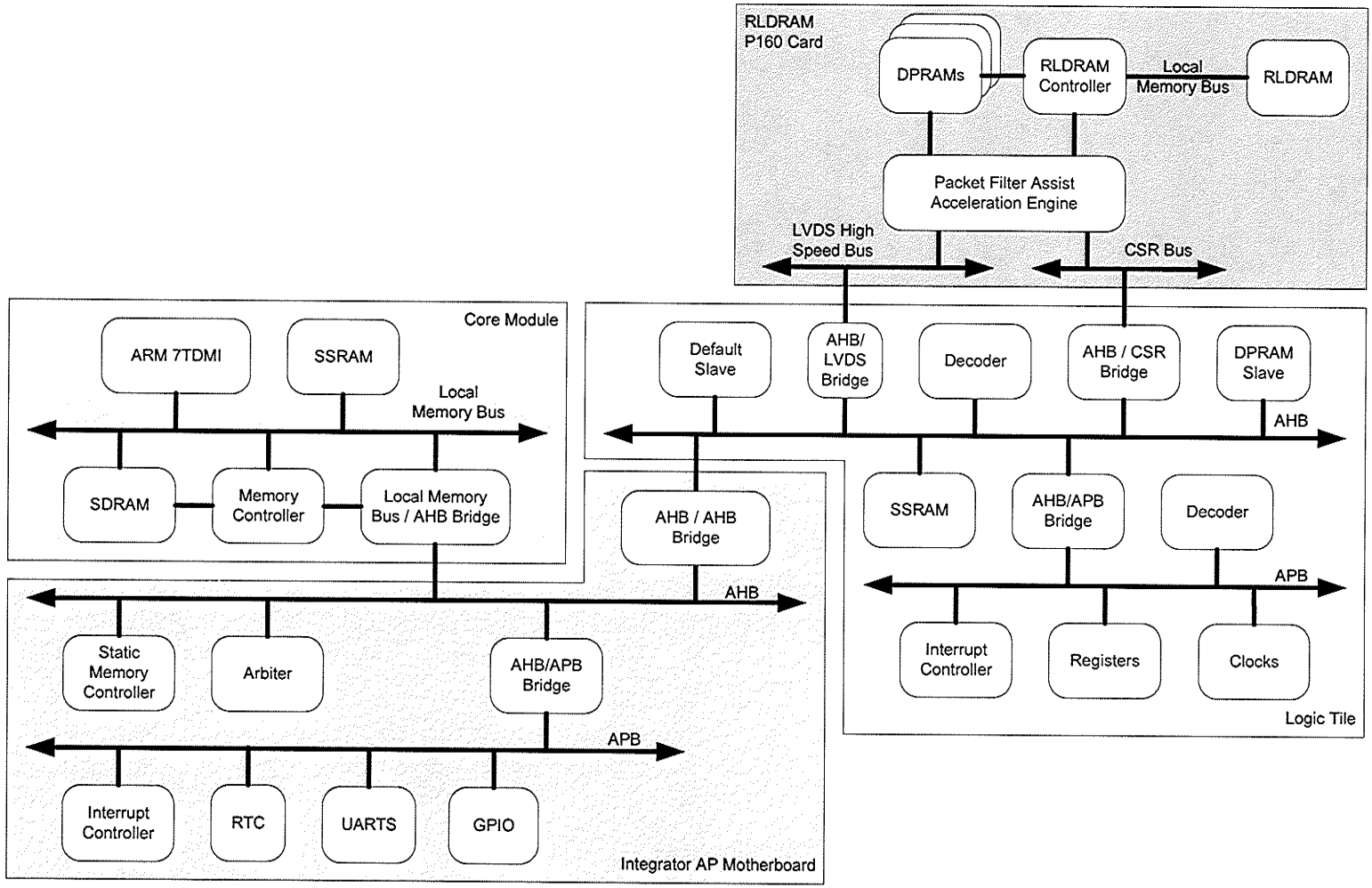


Figure 6-8: Block Diagram of CBV Packet Classifier SoC

From Figure 6-8 it is clear that the design makes efficient use of the available IP Libraries and reference designs. As well, there is a significant amount of custom IP required to execute the design functionality. The next few sections break down the main design blocks for the CBV packet classifier SoC.

6.4.1 System Buses

6.4.1.1 AMBA buses

AMBA buses are used throughout the design and are the de facto standard for on-chip buses. They provide a reusable design methodology by defining a common backbone for SoC modules.

6.4.1.2 Advanced High-Speed Bus (AHB)

Although the platform is capable of using either an AHB or an *Advanced System Bus* (ASB), AHB was selected as the primary bus in the design. Both buses are developed by ARM and intended to address the requirements of high performance synthesizable designs. However, the AHB is a newer generation of AMBA and its single clock edge protocol results in smoother integration with design automation tools used during typical ASIC development [26].

In the CBV SoC implementation the AHB bus is intended to run at 20 MHz with a shared 32-bit data bus and a single master. The processor, memory and other high-speed peripherals involved in the majority of data transfers are all connected to this bus. However, in an actual SoC this high-performance system bus would typically have multiple masters, separate high bandwidth read and write buses and would operate at speeds upwards of 200 MHz.

6.4.1.3 Advanced Peripheral Bus (APB)

The AMBA advanced peripheral bus is a secondary bus that appears as a single slave on the primary AHB bus. It is used to interface low bandwidth peripherals such as timers, UARTs, interrupt controllers and GPIO that do not require the high performance of the AHB bus. For more details on the AMBA APB the interested reader can refer to [26].

6.4.1.4 High-Speed Low Voltage Differential Signaling (LVDS) Bus

A high-speed *Serializer/Deserializer* (SerDes) LVDS link with eight lanes transfers data between the logic tile and the P160 RLDRAM module. Each lane consists of two differentially driven pairs of signal lines. One pair is for ingress data and the other for egress data. When

transmitting across a link with multiple lanes, the parallel representation is spread across all of the lanes. This provides greater link bandwidth for transmitting packets.

In this system and in digital systems in general, serialization of data makes perfect sense. For example, 64 bits of data at 50 MHz can use a 64-bit data link plus control lines to achieve 3.20 Gbps while the same data can be transmitted over 16 lines at 200 MHz providing a 75% pin savings. The same data can be serialized to 8 differential pairs (still 16 lines running at 200 MHz providing 400 Mbps DDR). This provides a pin savings of 75% and provides greater signal integrity and lower power.

6.4.1.5 Configuration Status Register (CSR) Bus

In addition to the LVDS link, a low-speed CSR bus connects the logic and RLDRAM modules. This allows the processor to read and write configuration and status info from the *Packet Filter Acceleration Assist Engine* (PFAAE) while it is busy performing CBV operations.

6.4.2 Core Module Blocks

The core module blocks are reserved by the platform but require a large degree of software configuration to perform the functional behaviors outlined in the design specifications.

6.4.2.1 ARM7TDMI processor

The ARM7TDMI core is a general-purpose 32-bit microprocessor whose architecture is based on the *reduced instruction set computer* (RISC) philosophy. The processor manages and schedules all of the software tasks related to the packet classifier. It handles interrupts, processes data, and sets up data transfer operations between itself, memory and other devices. The motivation for using this type of processor is that it offers high performance with low power consumption.

6.4.2.2 System Bus Bridge

The system bus bridge provides an asynchronous AMBA interface between the local memory bus and the system bus on the motherboard. It acts as a gateway allowing the ARM processor to access resources on the Integrator/AP and other connected modules. At power up the bridge can be configured to function as an ASB or AHB bridge [27]. In this design it is operating as an AHB bridge.

6.4.2.3 SDRAM Memory Controller

The SDRAM controller provides support for DIMMs with memory in the range of 16 to 256 MB and is implemented inside the core module FPGA. For the CBV packet classifier design, a 256 MB SDRAM DIMM was connected to the platform to be used as the primary memory for storing B-tree nodes.

6.4.2.4 Multi-ICE

A Multi-ICE connection allows JTAG hardware debugging.

6.4.3 Integrator/AP Blocks

The Integrator/AP hardware resources are reserved by the platform design and only require software configuration on power up. The blocks relevant to the CBV packet classifier application are outlined below.

6.4.3.1 AHB Arbiter

The AHB arbiter ensures that only one bus master is allowed to initiate data transfers at a time. In Figure 6-8, the ARM processor is the only bus master providing address and control information to initiate read and write operations.

6.4.3.2 AHB to APB Bridge

This bridge is a slave on the AHB and translates AHB transfers into an appropriate format for the slaves on the APB. This block also performs address decoding to generate slave select signals for APB peripherals.

6.4.3.3 Integrator AP System Input/Output Peripherals

For system input/output (I/O), the Integrator/AP FPGA incorporates controllers for a keyboard and mouse interface, two UART serial ports, LEDs and alphanumeric displays, three 16-bit counters/timers, GPIO and a RTC. These are all accessible from either a logic or core module through the system controller. Typically, one serial port is used to transfer data in serial form to and from a lab computer for debugging and logging. The system controller FPGA also has a 32-bit general-purpose Input /Output controller that connects to the high-density connectors between the AP and logic tile. This can be used for implementing a sideband bus between the two [25].

6.4.3.4 Interrupt Controller

An interrupt controller handles *interrupt requests* (IRQs) and *fast interrupt requests* (FIQs) for ARM processors. These may originate from peripheral controllers and devices on the logic module. Interrupts are enabled, acknowledged and cleared based on registers in the system controller FPGA on the motherboard [25].

6.4.3.5 Status and Control Registers

System status and control registers on the system controller FPGA allow software to configure and control such things as clock speeds and software resets. The local processor is also able to read this space to determine information such as the SDRAM size and configuration, the oscillator setup and type of processor.

6.4.4 Logic Tile Blocks

The logic module uses many components from the hardware IP library accompanying the ARM platform but also incorporates custom AHB slaves. Custom slaves are added to store configuration and status information as well as to interface with the PFAAE on the RLDRAM P160 module.

6.4.4.1 AHB Decoder

The AHB decoder is used to decode the address of each transfer and provide a select signal for the slave that is involved in the transfer. Possible targets include the AHB/APB bridge, AHB_CSR bridge, AHB/LVDS bridge, *dual-port RAM* (DPRAM), SSRAM and default slaves.

6.4.4.2 AHB Default Slave

The default slave is targeted when an address in the logic tile memory map is not covered by one of the other slaves. It responds back to the active master to satisfy the AMBA protocol.

6.4.4.3 AHB/LVDS Bridge

High-speed inter-module access between the logic tile and the P160 RLDRAM board is provided through the AHB/LVDS bridge. The bridge interacts with a pair of asynchronous command and response FIFOs. It translates egress AHB transfers into a suitable format for the PFAAE and ingress transfers into AHB format. Subsequent data massaging for the LVDS link is handled by SerDes logic.

6.4.4.4 AHB/CSR Bridge

A low-speed configuration and status register bus is provided between the logic tile and the P160 RLDRAM board. The bridge translates egress AHB transfers into a suitable format for the CSR bus and ingress responses into AHB format. Connected to the CSR bus is a bank of 255 x 16-bit registers used to configure and monitor the PFAAE.

6.4.4.5 Dual-Port RAM (DPRAM) Controller

The DPRAM controller provides support for accessing dual-port RAMs and registers inside the logic tile FPGA from the AHB. This memory is used to log packet classification performance and status information.

6.4.4.6 AHB RAM Controllers

Two SSRAM controllers provide byte, word and halfword operations to onboard Zero Bus Turnaround (ZBT) SSRAM.

6.4.4.7 AHB to APB Bridge

The AHB to APB bridge on the logic module is used to connect the APB peripherals to the AMBA AHB bus. It produces the select lines for each APB device.

6.4.4.8 APB Decoder

The APB decoder is used to decode the address of each transfer and provide a select signal for the slave that is involved in the transfer. Possible slaves include register, interrupt and clock control peripherals.

6.4.4.9 APB Peripherals

The three main APB peripherals used are the register, interrupt and clock control peripherals. Their functions are as follows:

- The APB register peripheral provides memory-mapped registers for configuring clock generators, writing LEDs and reading switch inputs.
- The APB interrupt controller contains basic interrupt controller registers for up to four APB interrupts.

- The APB clock control peripheral provides a parallel to serial interface to transfer clock divisors to onboard clock generators.

6.4.5 P160 RLDRAM Module Blocks

The P160 board uses an RLDRAM controller from the hardware IP libraries but also houses most of the custom logic required for the packet classifier application.

6.4.5.1 Packet Filter Assist Acceleration Engine (PFAAE)

The PFAAE controls all of the hardware-optimized tasks for the packet classifier. Its primary functions include memory management, processing rules, building CBVs, storing and retrieving CBVs as well as ORing them together during search operations. Additional detail on this block is provided in Chapter 7.

6.4.5.2 RLDRAM Controller

The RLDRAM controller provides support for two RLDRAM chips on the module and is implemented inside the RLDRAM module FPGA. To utilize all four RLDRAM chips two controllers would need to be implemented. Unfortunately, lack of FPGA resources ruled this option out.

6.4.5.3 Dual-Port RAMs

On-chip memory for use by the PFAAE tasks is provided by dual-port RAMs. Four 32w x 512d RAMs are connected to the PFAAE on one side and the RLDRAM controller on the other. This enables direct memory access type transfers between internal and external memory. A fifth dual-port memory is also instantiated inside the PFAAE block for local use while ORing CBVs.

Chapter 7: Development and Implementation

Once the design architecture has been mapped across platform resources, the next step is the design implementation phase, which is split among both hardware and software teams. While details on software specific development and verification is beyond the scope of this document, many of the design decisions, protocols and address mapping were mutually agreed upon between the software and hardware designs.

This chapter focuses on the custom hardware design and integration of reusable IP. The hardware design is broken across multiple FPGAs, one on the logic tile and one on the RLDRAM board. The major components implemented in each FPGA and explanations of the custom IP blocks developed are described in Sections 7.1 and 7.2. The chapter concludes with Section 7.3 outlining procedures for synthesizing and routing the design and specifying the FPGA utilization statistics.

7.1 ARM Logic Tile FPGA

The top-level block diagram illustrating the data flow and main components within the LT FPGA is shown in Figure 7-1.

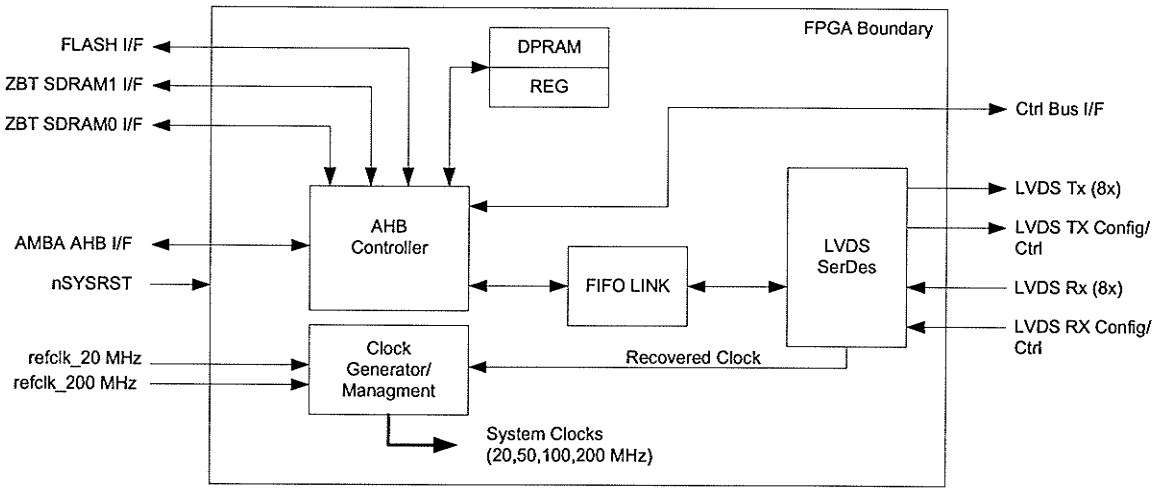


Figure 7-1: LT FPGA System Block Diagram

A brief summary of the operation of each of the functional blocks is provided in Table 7-1 while further detail on each of the blocks can be found in the remainder of this section.

Table 7-1: FPGA Functional Block Summary

Block	Function
LT_FPGA_top	Top-level HDL element for the logic tile that instantiates all of the components seen in Figure 7-1 as well as all of the support and glue logic to make a working system.
Clock Generator/Management	Generates all required system clocks.
AHB Controller	The AHB top-level controller connects the AHB slaves to the Integrator AHB bus. The following components are contained within this block: • AHB Decoder • AHB Default Slave • AHB Slave to master Mux • AHB SSRAM0/1 Controller • AHB APB bridge • APB Interrupt controller • APB Register peripheral • AHB/LVDS Bridge • AHB/CSR Bridge • AHB DPRAM
FIFO Link	Provides a pair of asynchronous command and response FIFOs between the AHB bus and LVDS bus connected to the RLDRAM module. This allows the AHB to operate at a much slower speed.
LVDS SerDes	LVDS SerDes logic to the RLDRAM FPGA.
DPRAM/REG	Memory and registers used for control/status information and logging packet classifier performance.

7.1.1 Logic Tile FPGA Pin Descriptions

The following table provides the name, direction, I/O standard and a description for each port (i.e. pin) in the top-level entity of the LT FPGA. A user constraints file is used to tie these ports to FPGA pins and guide placement and routing.

Table 7-2: LT FPGA Pin Descriptions

Name	Type	Function
reflck_200MHz	IN LVTTL 3.3V	200 MHz reference clock provided by onboard oscillators. Used for SerDes and AHB-LVDS bridge logic.
refclk_20MHz	IN LVTTL 3.3V	20 MHz reference clock for the AHB system bus.
nSYSRST	IN LVTTL 3.3V	This is a system wide reset input (active low) from the platform board. It is used to reset all user logic, peripherals and ARM cores.

Name	Type	Function
nSYSRST_out	OUT LVCMOS25	This is a system wide reset (active low) from the platform board. It is used to reset all user logic, peripherals and ARM cores as it is forwarded to the RLDRAM module.
AMBA AHB Interface		
HTRANS[1:0]	IN LVTTL 3.3V	Transfer type provided by the ARM7TDMI AHB master. Valid types include NONSEQUENTIAL, SEQUENTIAL, IDLE or BUSY.
HWRITE	IN LVTTL 3.3V	Write enable provided by the master. When asserted HIGH this signal indicates a write transfer and when LOW a read transfer.
HSIZE[1:0]	IN LVTTL 3.3V	Size of transfer provided by master. Indicates the size of the transfer, which is typically 8,16 or 32-bit.
HADDR[31:0]	IN LVTTL 3.3V	System address bus provided by master.
HDRID_IN[3:0]	IN LVTTL 3.3V	In a multi-LT system it denotes the position of the LT in the stack. Set to 'b1110 for a one logic tile system or first logic tile in a multi-tile system. Used in conjunction with the address to decode the target slave.
HREADY	BIDIR LVTTL 3.3V	Transfer done signal provided by slaves. When asserted HIGH, the HREADY signal indicates that a transfer on the bus has completed. This signal may be driven LOW to extend a transfer allowing the slave additional time to drive or sample data.
HDATA[31:0]	BIDIR LVTTL 3.3V	AHB data bus used to transfer data between the master and slaves. A bi-directional bus is used to save FPGA I/O pins.
HRESP[1:0]	OUT LVTTL 3.3V	Current transfer response status provided by slave. Possible indications include OKAY, ERROR, RETRY and SPLIT.
SYS_nIRQ0	OUT LVTTL 3.3V	Logic tile peripheral interrupt request. Combines interrupts from APB I/O and also an interrupt when the response FIFO from the RLDRAM module has data available.
LVDS Ingress Interface[1]		
rxclkina_p rxclkina_n	IN LVDS_25_DCI	LVDS receive clock differential pair. Used to generate a recovered clock and its complement.
dataina_p[7:0] dataina_n[7:0]	IN LVDS_25_DCI	LVDS data inputs. The receive data is clocked into DDR registers within the VIRTEX II FPGA. One bit of data is captured on each edge of the recovered clock providing 400 Mbps throughput per lane.
frameina_p frameina_n	IN LVDS_25_DCI	LVDS frame in. One frame signal is used for the entire link and ensures that the serial receive data from each lane is reproduced in its original 8-bit format.
clkin_locked	IN LVCMOS25	Clock synchronization input between FPGAs. Provides an indication when the RLDRAM FPGA has locked its DCM on the ARM FPGA LVDS clock output.
LVDS Egress Interface		
dataouta_p[7:0] dataouta_n[7:0]	OUT LVDS_25	LVDS data outputs. For each lane, eight bits of data are registered by the 50 MHz clock, multiplexed and re-timed by the 200 MHz clock and passed through a DDR register within the VIRTEX II IOB.

Name	Type	Function
clkouta_p clkouta_n	OUT LVDS_25	LVDS clock output. Generated through a DDR register within the VIRTEX II IOB with inputs connected to logic-high and logic-low.
frameouta_p frameouta_n	OUT LVDS_25	LVDS frame output. Generated through a DDR register in a VIRTEX II IOB. The output is logic-high for two clock periods and low for the next two.
FIFO_STATUS	OUT LVCMOS25	FIFO STATUS is a sideband signal providing an indication to the PFAAE when it should halt transmission of packets to prevent overflow. It is necessary because the AHB system bus is running at a much slower frequency than the system clock on the RLDRAM module.
clkout_locked	OUT LVCMOS25	Clock synchronization output between FPGAs. Provides an indication to the RLDRAM FPGA when the ARM FPGA has locked its DCM on its LVDS clock input.
Configuration Status Bus Interface		
CSR_DATA[15:0]	BIDIR LVCMOS25	Multiplexed address and data bus. Carries a 16-bit address when CSR_ALE =1, read data when CSR_nRd = 0 and Write data when CSR_nWR = 0. To/From AHB/CSR bridge slave.
CSR_CS_b	OUT LVCMOS25	Chip select (active low). The current bus cycle is targeting the RLDRAM FPGA when low.
CSR_RD_b	OUT LVCMOS25	Read enable (active low). The current bus cycle is a read when low. The FPGA should capture data onto the CSR_DATA bus.
CSR_WR_b	OUT LVCMOS25	Write enable (active low). The current bus cycle is a write when low. The FPGA should write data on the CSR_DATA bus.
CSR_ALE	OUT LVCMOS25	Address latch enable. The CSR_DATA bus carries an address when this is high.
Zero Bus Turnaround (ZBT) RAM0 / RAM1 Interfaces		
RAM0_SCLK RAM1_SCLK	OUT LVTTL 3.3V	ZBT CLK.
RAM0_SnWBYTE[3:0] RAM1_SnWBYTE[3:0]	OUT LVTTL 3.3V	ZBT byte write control signals.
RAM0_SnOE RAM1_SnOE	OUT LVTTL 3.3V	ZBT output enable. Enables the data output drivers.
RAM0_SnCE RAM1_SnCE	OUT LVTTL 3.3V	ZBT chip select. Enables the RAM chip.
RAM0_SADVnLD RAM1_SADVnLD	OUT LVTTL 3.3V	ZBT advance. Advances the internal burst counter.
RAM0_SnWR RAM1_SnWR	OUT LVTTL 3.3V	ZBT read/write. Determines the cycle type.
RAM0_SnCKE RAM1_SnCKE	OUT LVTTL 3.3V	ZBT clock enable. Enables the input clock on the RAM chip.
RAM0_SMODE RAM1_SMODE	OUT LVTTL 3.3V	ZBT mode signal.

Name	Type	Function
RAM0_SA[20;2] RAM1_SA[20;2]	OUT LVTTL 3.3V	ZBT address bus.
RAM0_SD[31:0] RAM1_SD[31:0]	BIDIR LVTTL 3.3V	ZBT data bus.
FLASH control signals		
FnOE	OUT LVTTL 3.3V	FLASH output enable. FLASH is disabled in the packet classifier design so FnOE is tied high.
FnWE	OUT LVTTL 3.3V	FLASH write enable. FLASH is disabled in the packet classifier design so FnWE is tied high.
FA0	OUT LVTTL 3.3V	FLASH address 0. FLASH is disabled in the packet classifier design so FA0 is tied low.
FA1	OUT LVTTL 3.3V	FLASH address 1. FLASH is disabled in the packet classifier design so FA1 is tied low.
FA21	OUT LVTTL 3.3V	FLASH address 21. FLASH is disabled in the packet classifier design so FA21 is tied low.
FA22	OUT LVTTL 3.3V	FLASH address 22. FLASH is disabled in the packet classifier design so FA22 is tied low.
FnBYTE	OUT LVTTL 3.3V	FLASH byte mode. FLASH is disabled in the packet classifier design so FnBYTE is tied high.
FnCE	OUT LVTTL 3.3V	FLASH chip enable. FLASH is disabled in the packet classifier design so FnCE is tied low.
Joint Test Action Group (JTAG)		
D_TCK	IN LVTTL 3.3V	JTAG debug test clock to be used with a TAP controller.
D_TDI	IN LVTTL 3.3V	JTAG debug test data in to be used with a TAP controller.
D_TDO	OUT LVTTL 3.3V	JTAG debug test data out, to be used in debug mode with TAP controller. Since no TAP controller is implemented in this design D_TDI is routed directly to D_TDO.
D_RTCK	OUT LVTTL 3.3V	JTAG debug return test clock (Multi-ICE feature) to be used with a TAP controller. Since no TAP controller is implemented in this design D_TCK is routed directly to D_RTCK.
PLD Configuration		
CLK_24MHZ_FPGA	IN LVTTL 3.3V	24 MHz clock for FPGA configuration. This clock is used for serially transferring data to the on-board PLD which also runs at 24 MHz.
ALWAYS_ONE	IN LVTTL 3.3V	This is a dual-function pin. Prior to FPGA configuration, it controls the transfer of data to the FPGA. After configuration, it controls the configuration of logic tile foldover switches, which govern the datapath of many pins on the logic tile high-density connectors.
SER_PLD_DATA	OUT LVTTL 3.3V	The FPGA uses this line to serially transfer configuration data to the PLD. It controls foldover switches on the logic tile to determine the routing of signals between the Integrator/AP, the logic tile and other stacked modules.

Name	Type	Function
Programmable Clock		
CLK_SCLK	OUT LVTTL 3.3V	Serial interface clock for configuring programmable clocks.
CLK_DATA	OUT LVTTL 3.3V	Serial interface data for configuring programmable clocks.
CLK_STROBE[2:0]	OUT LVTTL 3.3V	Serial interface strobes for configuring programmable clocks.
Miscellaneous		
ui_led[7:0]	OUT LVTTL 3.3V	General-purpose LEDs connected to the FPGA via the Integrator/IM_LT1 interface module for diagnostic purposes.
SW[3:0]	IN LVTTL 3.3V	General-purpose switch inputs connected to the FPGA. The inputs are high when the switches are ON.
LT1_SW[7:0]	IN LVTTL 3.3V	General-purpose switch inputs connected to the FPGA via the Integrator/IM_LT1 interface module.
nPRES[1:3] SYS_FIQ[1:3] SYS_IRQ[1:3] SREQ[1:3] SLOCK[1:3]	OUT LVTTL 3.3V	These signals are normally connected to HADDR on a second logic tile but since there is only one logic tile in this design, they are tied off internally to either logic-high or low.

[1] *Digitally controlled impedance* (DCI) was used for the FPGA-FPGA LVDS interface since neither the ARM board nor the RLDRAM board had onboard termination for the differential signaling. This is also the preferred method as it reduces the number of board components.

7.1.2 ARM Clocking

The following clocks are used in the LT FPGA:

- HCLK – 20 MHz. AHB system bus clock reference used for most logic in the part. It comes from a 20 MHz reference clock provided by an oscillator on the ARM Integrator/AP ASIC development motherboard.
- HCLK180 – 20 MHz. Complement of the AHB system bus clock used for most logic in the part.
- sysclk4x – 200 MHz. System clock used for LVDS transmit logic. This clock source is provided by one of the onboard general-purpose MicroClock ICS307 programmable clock generators. A 24 MHz crystal oscillator supplies the ICS307s with a reference clock and the serial control of the programmable clock is implemented in the FPGA.
- sysclk4x180 – 200 MHz. Complement of the sysclk4x.
- sysclk2x – 100 MHz. Used for the AHB-LVDS bridge receive logic. This clock is phase-aligned with sysclk4x.
- sysclk – 50 MHz. Used for LVDS SerDes logic. This clock is phase-aligned with sysclk4x.

- rxclk – 200 MHz. Recovered clock from rxclkina_n and rxclkina_p, which is forwarded from the LVDS transceiver on the RLDRAM FPGA. This clock is phase-shifted to provide a sampling point in the center of the rxdata valid bit periods.
- rxclk180 – 200 MHz. This clock is the complement of rxclk and is used for LVDS receive logic.
- clkouta_p, clkouta_n – 200 MHz. This is a regenerated differential version of the sysclk4x. It is forwarded to the LVDS receiver on the RLDRAM FPGA

Figure 7-2 provides a block diagram for the clock generation and management logic.

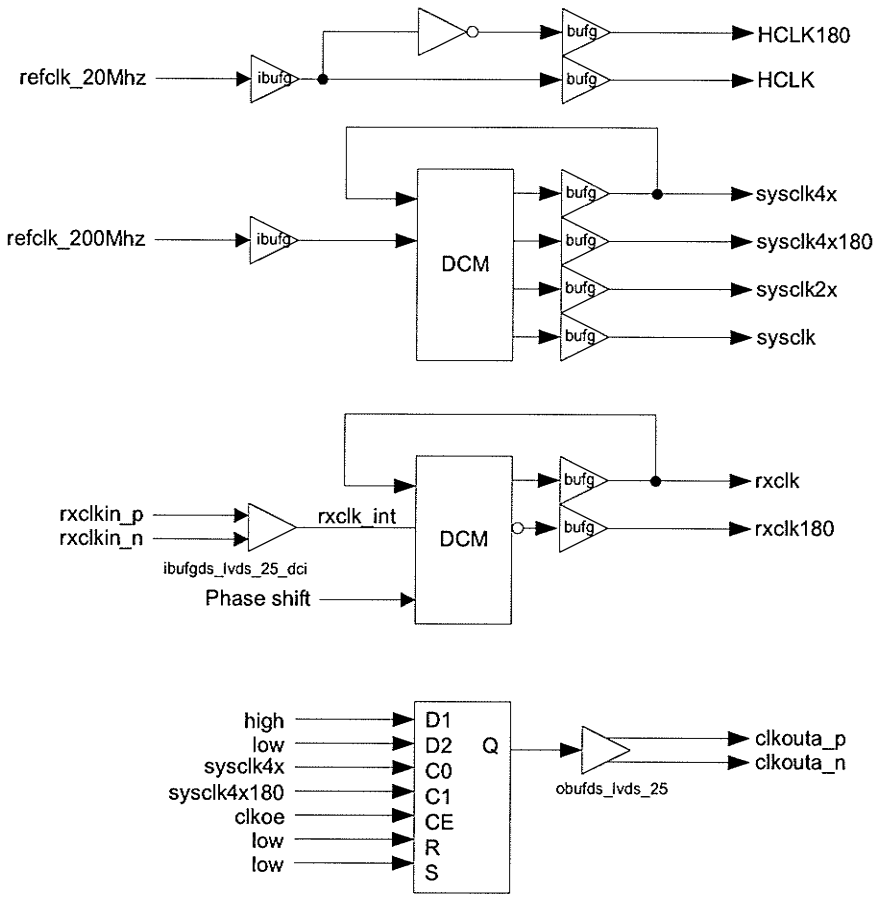


Figure 7-2: LT FPGA Clock Management

The clock logic has the following features:

- FPGA input clocks are sent through *global clock input buffers* (IBUFGs) to minimize skew.
- Clocks generated for use within the LT FPGA are sent through *global clock buffers* (BUFGs) to ensure global routing is used for all clock resources.

- The design uses two *Digital Clock Managers* (DCMs). A basic DCM accepts a buffered 200 MHz reference clock and generates phase-aligned 50, 100 and 200 MHz system clocks as well as an inverted version of the 200 MHz system clock. These clocks are used for the LVDS bridge and SerDes logic. A second dedicated phase shifting DCM is used for LVDS receive clocks. This DCM generates a reference clock at the receive port based on the forwarded differential clock. A fixed phase shift is required to provide a sampling point near the center of the receiver's valid bit periods.
- To guarantee that the AHB peripherals are synchronized with the AHB master on the core module, the 20 MHz system bus clock is buffered but not regenerated through a DCM.

7.1.3 AHB Controller

The majority of the code within the AHB controller is re-used from an IP library provided with the ARM logic tile [28]. Of course, a number of modifications and additional AHB slaves were added to suit the packet classifier application. Each of the blocks within the controller is briefly described in Table 7-3. The custom IP blocks, which had to be designed for the packet classification algorithm, are described in more detail in subsequent sections.

Table 7-3: AHB Controller IP [28]

File	Function
AHBTopLevel	This block is the top-level HDL instantiating all of the high-speed peripherals, decoder, and all the necessary support and glue logic for a working AMBA system.
AHBDecoder	The decoder block provides high-speed peripherals with select lines. These are generated from the address lines and the module ID (position in stack) signals from the motherboard. The decoder block also generates the select signals for the default slave (instantiated in the top-level module).
AHBMuxS2M	This module multiplexes the HREADY, HRESP and HRDATA signals from the slaves in the logic tile and generates the combined HREADY, HRESP and HRDATA signals used to interface with the AMBA system bus.
AHBZBTRAM	High-speed peripherals require that the SSRAM controller blocks support 8,16 and 32-bit operations to the SSRAM on the logic tile
AHB2APB	Bridge blocks connecting the APB peripherals to the high-speed AMBA AHB bus. They produce the peripheral select signals for each of the APB peripherals.
AHBAPBSys	The components required for an APB system are instantiated in this block. These include the bridge and the APB peripherals. This file also multiplexes the APB peripheral read buses and concatenates the interrupt sources to feed into the interrupt controller peripheral.
APBRegs	The APB register peripheral provides memory-mapped registers to be used for: • configuring the clock generators • writing to the user LEDs on the logic tile and interface module • reading the user switch inputs on the logic tile and interface module • latching the pressing of the push button to generate an expansion interrupt

File	Function
APBIntcon	The APB interrupt controller contains all of the standard interrupt controller registers and has an input port for four APB interrupts. Only one of them is used in the packet classifier design. The remaining three are set inactive in the AHBAPBSys block. Four software interrupts are implemented.
AHBDefaultSlave	The HREADY and HRESP signals are driven from the default slave if an address in the logic tile memory map is not covered by one of the peripherals in the design. This provides an indication to the software that no peripheral is present at that address.
APBClocks	Provides a parallel to serial interface that transfers the APB clock divisor register contents to the ICS307 clock generators.
APBClockArbiter	Implements a simple arbitration scheme to ensure that each of the three clock controllers are serviced independently
AHB/LVDS Bridge Slave	This custom block provides a controller for the pair of asynchronous FIFOs that communicate with the RLDRAM FPGA (i.e. an interface to the high-speed hardware peripheral that assists the packet classifier search).
AHB/CSR Bridge Slave	This custom block provides a controller for accessing configuration and status registers on the RLDRAM FPGA.
AHB DPRAM Slave	This custom block provides a DPRAM and register controller for the AHB bus. The DPRAM and registers are used for logging system performance measurements.

The platform architecture is based on a 32-bit address space. In other words, there is 4 GB worth of memory addresses. In practice, embedded systems will use only a subset of the total physical address space. Therefore, a memory map is used to define the locations of various types of memory, I/O and registers within the address space. The logic tile memory map is broken down in Figure 7-3.

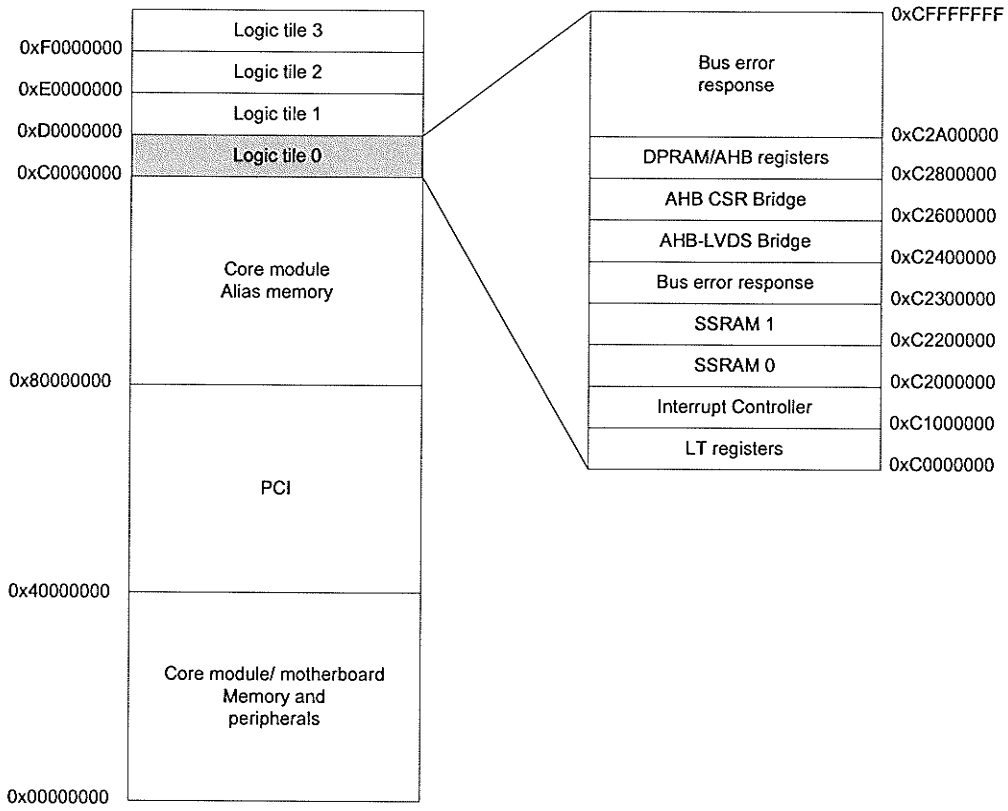


Figure 7-3: AHB Memory Map [28]

The main address decoder on the Integrator/AP motherboard assigns each connected logic tile a block of 256 MB. The base address of the logic tile is set by the HDRID signals and the relative position of the tile in the stack [28]. For the packet classifier design, a single logic tile is used with base address 0xC0000000. Figure 7-3 shows how the packet classifier decodes its assigned address space. It is split among the AHB slaves with a minimum size of 1 MB per slave. The address space assigned to the AHB2APB bridge is further decoded as shown in the memory map. It should also be noted that the memory map indicates a bus error response where addresses are accessed and no physical memory exists. This response is handled by the default slave.

7.1.4 AHB/LVDS Bridge

The function of the AHB-LVDS bridge is to provide a high-speed asynchronous bus interface between the AMBA AHB system bus connecting the motherboard, logic module and core module and the LVDS bus connecting the logic tile to the RLDRAM module. In total, the bridge is made up of three components: a slave that sits on the AHB bus, an elastic FIFO and the LVDS SerDes logic.

7.1.4.1 AHB/LVDS Bridge Slave

The slave acts as a controller allowing the ARM7TDMI processor to read and write a pair of asynchronous command and response FIFOs connected to the LVDS bus. The first FIFO provides write access and the second FIFO is for read access by the processor. The finite *state machine* (SM) for the controller is illustrated in Figure 7-4.

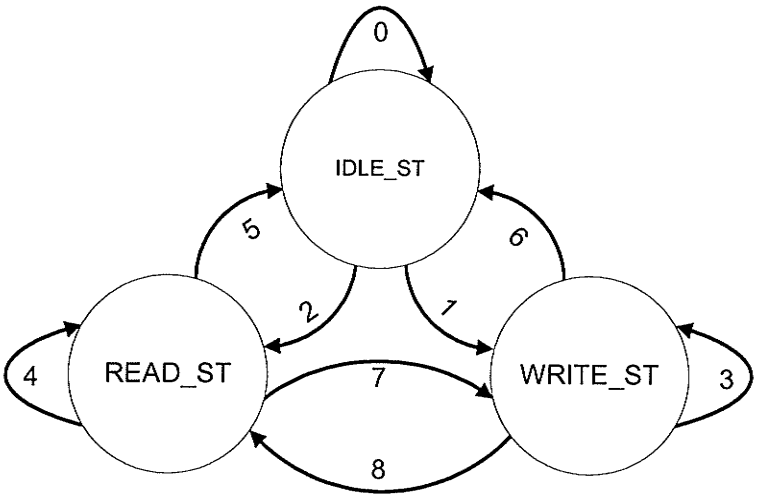


Figure 7-4: AHB/LVDS Bridge FIFO State Machine

The transitions of this state machine are summarized in Table 7-4.

Table 7-4: AHB/LVDS Bridge FIFO State Machine Transitions

Arc	From State	To State	Reason
0,5,6	ANY	IDLE_ST	No valid AHB read or write transactions are pending for the AHB slave. Valid AHB transfers only take place when a non-sequential or sequential transfer is shown on HTRANS and the AHB slave is selected based on address decoding.
1,3,7	ANY	WRITE_ST	HWRITE is asserted and a valid AHB transfer is detected.
2,4,8	ANY	READ_ST	HWRITE is de-asserted and a valid AHB transfer is detected.

The FIFOs are accessed when the ARM puts out the appropriate address (0xC2400000) and levels on the control lines. The slave will access the corresponding command or response FIFO, depending on the state of HWRITE, and return the appropriate AHB reply to the processor. For a write across the LVDS bus the processor would perform an AHB write to address 0xC2400000 and the data on HWDATA would be written to the transmit FIFO. For a read operation, the processor would perform a standard AHB read to address 0xC2400000. The slave would perform the required read from the receive FIFO and return the data to the processor on HRDATA.

7.1.4.2 FIFO Link

An elastic FIFO link is the second component of the AHB/LVDS bridge enabling inter-module access. It primarily consists of a pair of asynchronous command and response FIFOs (32w x 255d) based on Xilinx LogiCore's Asynchronous FIFO [36]. Block diagrams for the system bus read and write paths are shown in Figure 7-5 and Figure 7-6, respectively.

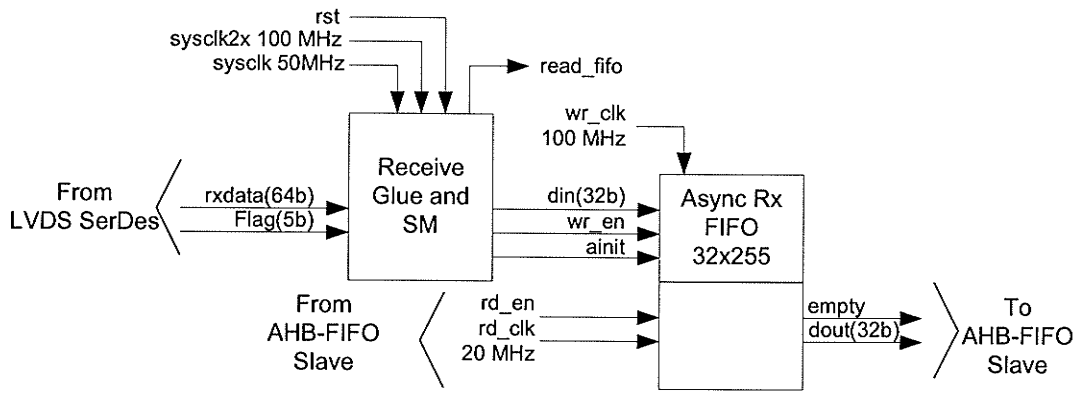


Figure 7-5: Receive Path (Reads) for system bus

The receive path has the following features:

- Operates at 20 MHz in the read clock domain. (AHB system bus side).
- Operates at 50 MHz on the LVDS SerDes side, but at 100MHz in the FIFO write clock domain.
- Once the LVDS link is synchronized, rxdata (64b) is passed to the receive/response FIFO on the rising edge of every 50 MHz clock cycle.
- A receive state machine is responsible for filtering idle data (zeros) and detecting *start-of-packet* (SOP) headers. Once the SOP is detected, the write enable signal is asserted and the packet is written into the receive FIFO, 32-bits at a time.
- An interrupt based on the receive FIFO's empty line is used to inform the processor when data is available to be read from the receive FIFO.
- To perform a read operation via the system bus, the processor makes an AHB read request to the AHB/LVDS bridge. If the receive FIFO is not empty, the AHB slave will assert the read enable for the FIFO and return the output data to the processor.

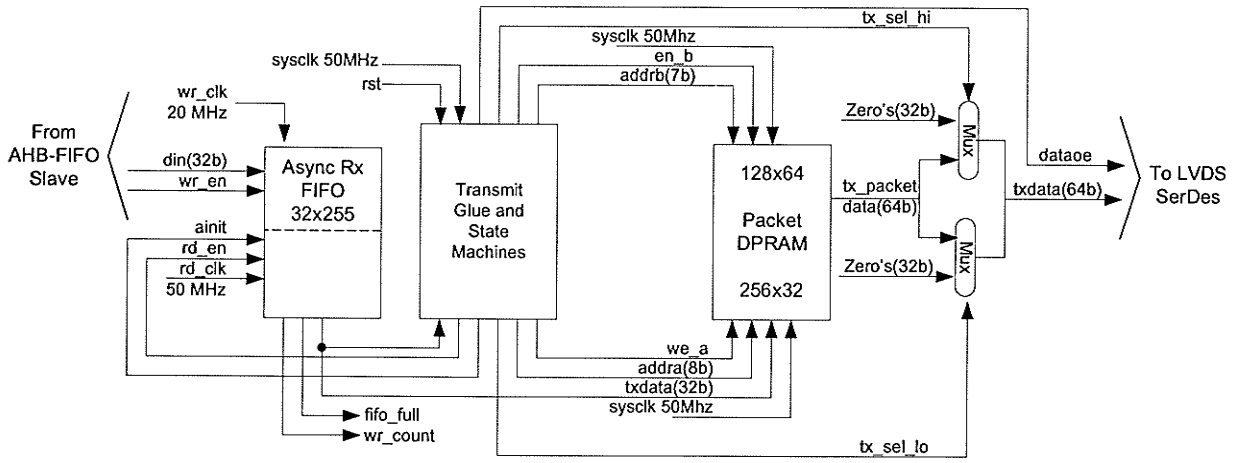


Figure 7-6: Transmit Path (Writes) for System Bus

The transmit path has the following features:

- Operates at 20 MHz in the FIFO write clock domain (AHB system bus side).
- Operates at 50 MHz in the FIFO read clock domain (LVDS SerDes side).
- Once the LVDS link is active, txdata (64b) is forwarded to the LVDS logic on the rising edge of every 50 MHz clock cycle.
- The transmit state machine is responsible for building entire packets to be sent across the link. It monitors the transmit FIFO data for SOPs and uses a small dual-port RAM as temporary storage until a full packet is received from the AHB interface. Once a full packet is available, it is burst out in 64-bit format to the SerDes logic.

7.1.4.3 LVDS SerDes Logic

The SerDes logic between the Virtex II FPGAs on the LT and the P160 RLDRAM board defines a physical implementation for transmitting and receiving packets between ports on an LVDS link. At the transmitting ports, packets are encoded into a serial bit stream. At the receiving ports, the packets are decoded from a serial bit stream. The design of this block is largely based on recommendations in a Xilinx application note for high-speed data serialization and de-serialization [32]. The application note provides possible system architectures along with synthesizable code of a four-lane LVDS SerDes building block. Figure 7-7 provides a block diagram of the SerDes logic implementation.

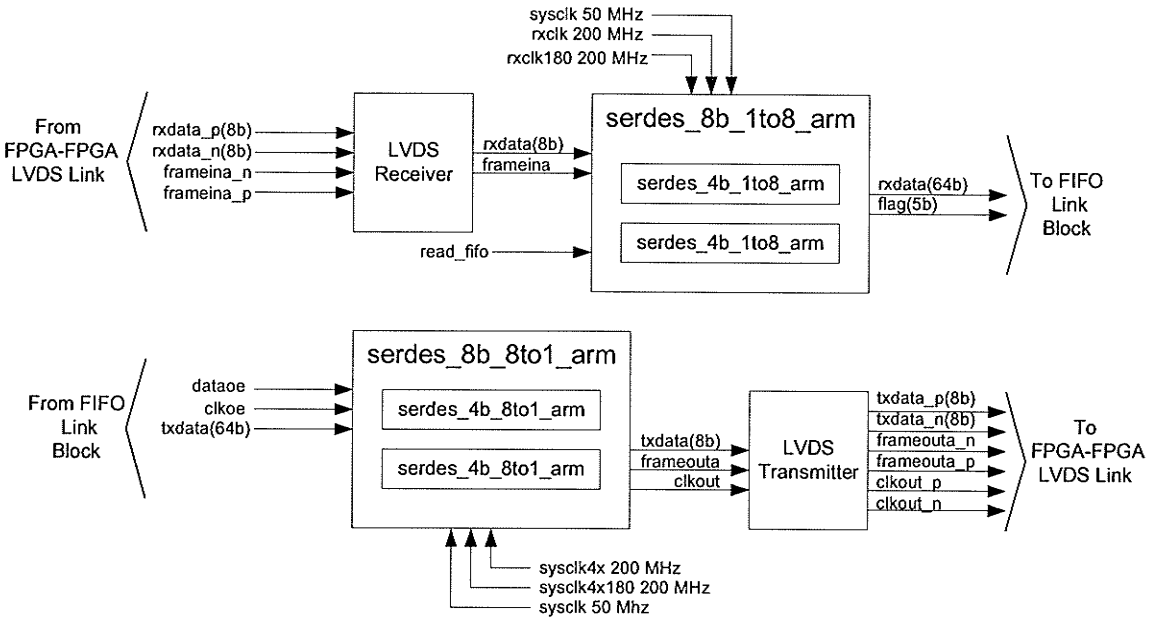


Figure 7-7: LVDS SerDes Block Diagram

This block has the following features:

- Operates at 50 MHz in the system clock domain
- Operates at 200 MHz in transmit and receive clock domains, which is half the data transmission frequency. Therefore, the data changes on every edge of the clock to provide double-data-rate signaling.
- Two four-lane LVDS SerDes blocks are tiled to provide a link consisting of 8 pairs of differential lanes. Each lane operates at 400 Mbps for a collective data transfer of 3.2 Gbps in each direction.
- Relative placement constraints are needed so that consistent performance will be maintained each time the RTL is routed.

Transmit Features:

- Using a clock forwarding technique, the data, clock and a framing signal are forwarded across the backplane.
- The forwarded 200 MHz LVDS transmission clock (clkout_n, clockout_p) is generated through a DDR flip-flop with inputs tied to high and low. Perhaps a better technique would be to embed the clock information into the data at the transmitter. The clock could then be recovered at the receiver.
- Provides a *direct current* (DC) balanced frame signal whose output is high for two clocks and low for two clocks.
- Accepts txdata (64b) from the FIFO link and converts it from parallel orientation into a serial byte stream. The 64 bits are spread out and sent eight bits per lane as seen in Figure 7-8.

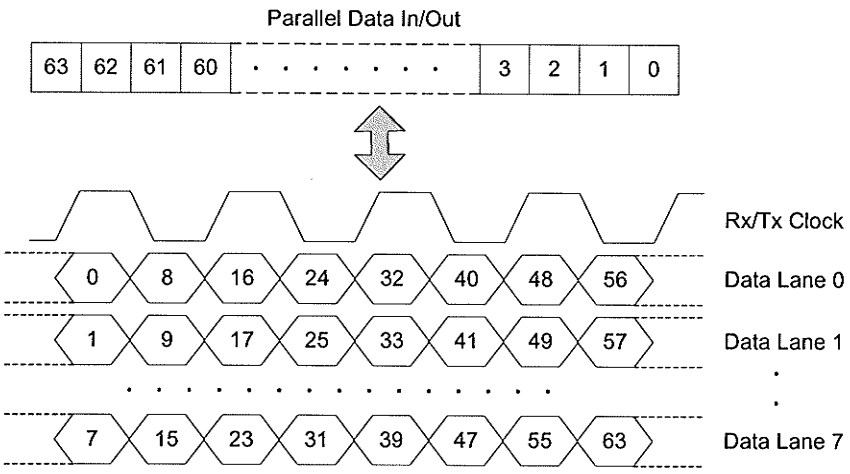


Figure 7-8: Serialization of Data In/Out

Receive Features:

- This block is responsible for sampling the incoming data on each edge of the recovered clock. The clock manager provides the rxclk and its complement with a digital phase adjustment to align the clock with the center of the ingress data. This is necessary to ensure that the receiver samples the data in the middle of the eye (i.e. valid bit period).
- Performs de-serialization of the rxdata (8b). The resulting bit streams from each of the 4-bit blocks are de-parsed into a single stream. These streams are converted back to parallel form and dumped into a simple asynchronous FIFO consisting of two Xilinx block RAMs. The input frame signal is used to ensure the 8-bit data from each lane is reconstructed in its original format.
- Provides status information to the rest of the system indicating the degree of FIFO fullness. This allows the rest of the system to operate at a different frequency or even phase than the receive logic.

7.1.5 AHB DPRAM SLAVE

The function of this AHB slave is to provide an interface to internal memory resources such as registers and Xilinx block RAMs to be used for configuration and logging performance results. This block has the following features:

- Uses a state machine similar to the AHB/LVDS Bridge for performing reads and writes to memory. The internal memory is accessed when the ARM processor puts out the appropriate address and levels on the control lines. The slave will access the corresponding DPRAM or register, depending on the state of HWRITE, and return the appropriate AHB reply to the processor.
- The slave has a base address of 0xC2800000 and range of 2 MB. At present, most of this address space is unused. The first 2KB [0xC2800000:0xC28007FC] are used for a DPRAM and the next 2KB [0xC2800800:0xC2800FFC] are used for registers.

- For a memory write, the processor performs an AHB write in the address range [0xC2800000-0xC2800FFC] and the data from HWDATA is written to the internal memory.
- For a read operation, the processor performs a standard AHB read in the address range [0xC2800000-0xC2800FFC]. The slave performs the required read from internal memory and returns the data on HRDATA.
- The registers are used to setup and initialize timers for automatically logging hardware performance statistics into the DPRAM.

7.1.6 AHB/CSR Bridge Slave

The function of the AHB-CSR bus bridge is to provide a sideband asynchronous bus interface between the AMBA AHB system bus connecting the motherboard, logic module and core module and the control status register bus connecting the logic tile to the RLDRAM registers. Physically the bus is bi-directional and multiplexes the data and address lines. The finite state machine for the controller is shown in Figure 7-9 and the state transitions are summarized in Table 7-5.

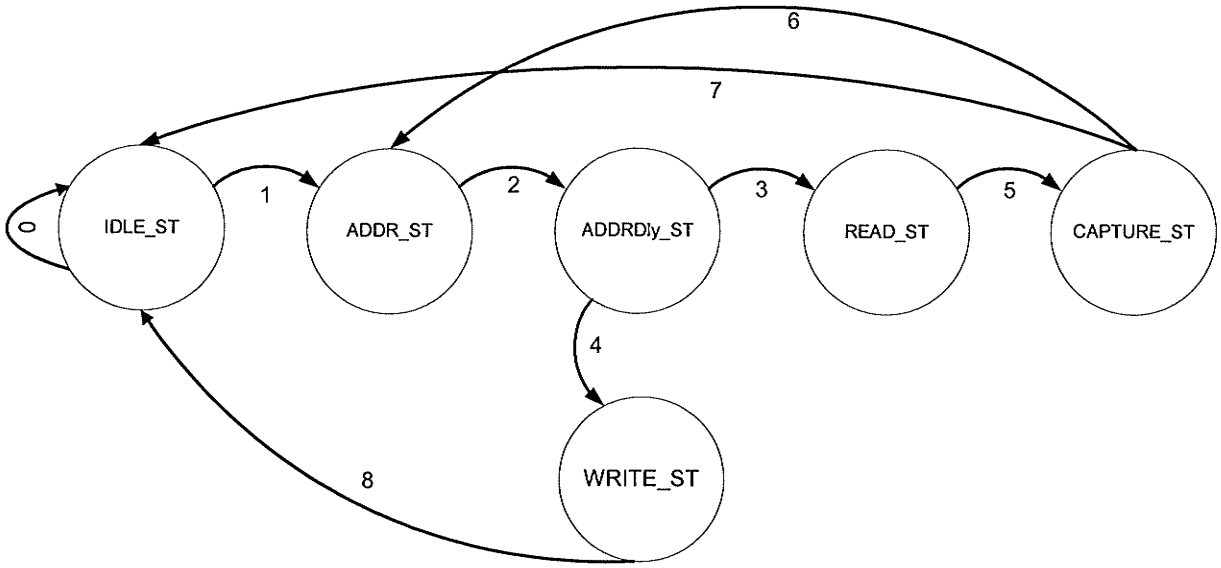


Figure 7-9: AHB/CSR Bridge State Machine

Table 7-5: AHB CSR State Machine Transitions

Arc	From State	To State	Reason
0,7,8	ANY	IDLE_ST	No valid AHB read or write transactions are pending for the slave. Valid AHB transfers only take place when a non-sequential or sequential transfer is shown on HTRANS and the AHB CSR slave is selected based on address decoding.
1	IDLE_ST	ADDR_ST	A valid AHB transfer is detected. HADDR, HTRANS and HWRITE values are registered on this transition.
2	ADDR_ST	ADDRDly_ST	Automatic transition.

Arc	From State	To State	Reason
3	ADDRDly_ST	READ_ST	The registered version of HWRITE is de-asserted.
4	ADDRDly_ST	WRITE_ST	The registered version of HWRITE is asserted.
5	READ_ST	CAPTURE_ST	Automatic transition. Control of the address/data bus is relinquished allowing the target to drive data on to the bus which is ouput on HRDATA.
6	CAPTURE_ST	ADDR_ST	Another valid data transfer is detected.

Note, to extend the transfer HREADYOut is de-asserted during the IDLE, ADDRESS and CAPTURE states. This allows the slave additional time to drive the separate address and data information onto the shared bus for writes as well as additional time to capture and sample data.

7.2 P160 RLDRAM Module FPGA

This section describes the features and implementation of the RLDRAM FPGA. The top-level block diagram illustrating the dataflow and main components within the FPGA is shown in Figure 7-10.

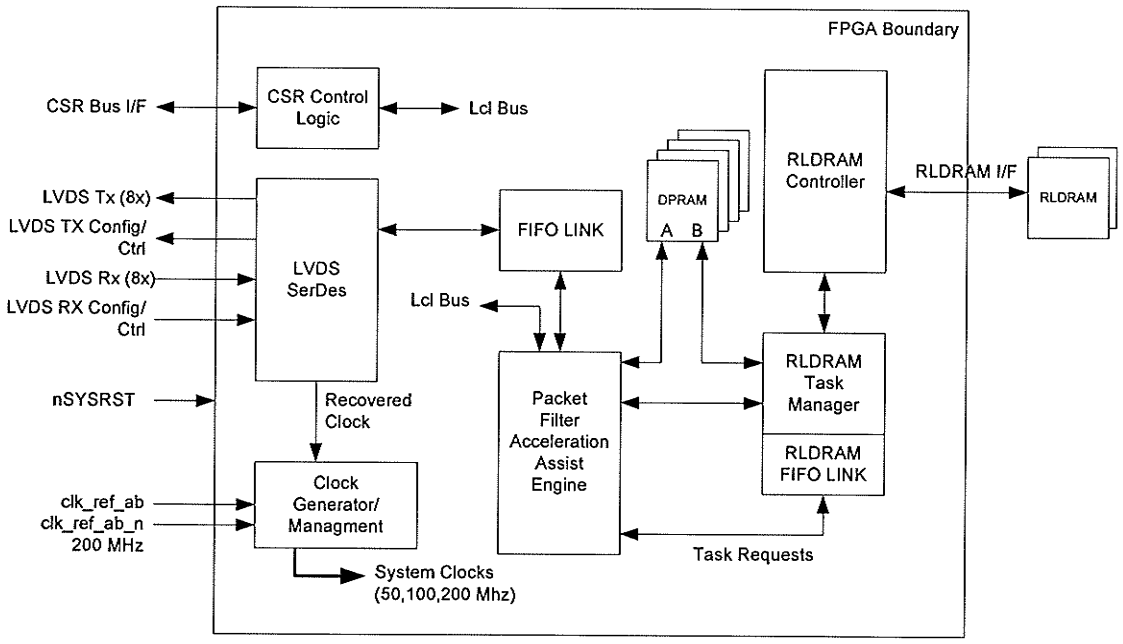


Figure 7-10: RLDRAM FPGA System Block Diagram

A brief summary of the operation of each of the functional blocks is provided in Table 7-6 while further detail on the blocks can be found in subsequent sections. The above diagram also illustrates the packet classifier design only utilized two RLDRAMs or 64 MB of the external

memory. FPGA resource limitations prevented more than one controller from being instantiated due to the size of the packet classifier application logic.

Table 7-6: RLDRAM FPGA Functional Block Summary

Block	Function
RLDRAM_fpga_top	Top-level HDL element for the RLDRAM FPGA that instantiates all of the components seen in Figure 7-10 as well as the support and glue logic to make a working system.
CSR Control Logic	Interface glue required for the CSR bus to access internal registers.
Clock Generator/ Management	Generates system clocks from an external oscillator input.
FIFO Link	Provides a pair of asynchronous FIFOs as an interface between the PFAAE and LVDS interface, which allows the PFAAE to operate at a slower speed. This block is very similar to the one used on the ARM FPGA, so further detail is omitted. The main difference is that the PFAAE side operates at 50 MHz where the AHB side uses 20 MHz. Refer to section 7.1.4.2.
LVDS SerDes	Controls the LVDS link to the LT FPGA. The RLDRAM LVDS SerDes logic is virtually identical to the LT LVDS SerDes logic. Refer to section 7.1.4.3.
RLDRAM Task Manager/ RLDRAM FIFO Link[1]	The RLDRAM Task Manager connects the PFAAE to the RLDRAM interface block. It accepts requests from the PFAAE and converts them into RLDRAM commands for transferring memory blocks between internal memory and the RLDRAM. This block is also responsible for refresh commands. A FIFO Link had to be added between the two interfaces to synchronize communication between the 200 MHz and 50 MHz clock domains.
RLDRAM Interface block[2]	This RLDRAM Interface block manages two RLDRAM devices where the two RLDRAMs share common address and control signals, and each RLDRAM has separate chip select, data, read data valid, and read data strobes. The module is a black box but is comprised of a controller, data path buffer, DCMs, and several instantiated IOB registers and buffers.
Packet Filter Acceleration Assist Engine (PFAAE)	The PFAAE contains the majority of the custom logic required for the packet classifier. It houses hardware for performing the various modes of operation outlined below: • FIFO read/write tasks • Filter Mode: • Build Mode • User Mode • Loopback Mode More detail on these modes is provided in Section 7.2.4.

[1]The RLDRAM Task Manager and RLDRAM FIFO Link contain some proprietary third party IP. Therefore, explicit details on its design and implementation are omitted.

[2]The RLDRAM interface logic is proprietary third party IP. Therefore, explicit details on its design and implementation are omitted.

7.2.1 RLDRAM FPGA Pin Descriptions

The following table provides the name, direction, I/O standard and a description for each port (i.e. pin) in the top-level entity of the RLDRAM FPGA.

Table 7-7: RLDRAM FPGA Pin Descriptions

Name	Type	Function
clk_ref_ab clk_ref_ab_n	LVDS_33	200 MHz reference CLK provided by on-board oscillator. 200 MHz reference CLK complement provided by on-board oscillator.
nSYSRST	IN LVCMOS25	This is a system wide reset (active low) from the platform board. It is used to reset all user logic, peripherals and ARM cores, and it is forwarded to the RLDRAM module by the logic tile.
LVDS Ingress Interface		
rxclkina_p rxclkina_n	IN LVDS_25_DCI	LVDS receive clock differential pair. Used to generate a recovered clock and its complement.
dataina_p[7:0] dataina_n[7:0]	IN LVDS_25_DCI	LVDS data inputs. The receive data is clocked into DDR registers within the VIRTEX II FPGA. One bit of data is captured on each edge of the recovered clock providing 400 Mbps throughput per lane.
frameina_p frameina_n	IN LVDS_25_DCI	LVDS frame in. One frame signal is used for the entire link and ensures that the serial receive data from each lane is reproduced in its original 8-bit format.
FIFO_ Status	IN LVCMOS25	FIFO_Status is a sideband signal providing an indication to the PFAAE when it should halt transmission of packets to prevent overflow. It is necessary because the AHB system bus is running at a much slower frequency than the system clock on the RLDRAM module.
clkin_locked	IN LVCMOS25	Clock synchronization input between FPGAs. Provides an indication when the LT FPGA has locked its DCM on the RLDRAM FPGA LVDS clock output.
LVDS Egress Interface		
dataouta_p[7:0] dataouta_n[7:0]	OUT LVDS_25	LVDS data outputs. For each lane, eight bits of data are registered by the 50 MHz clock, multiplexed and re-timed by the 200 MHz clock, and passed through a DDR register within the VIRTEX II IOB.
clkouta_p clkouta_n	OUT LVDS_25	LVDS clock output. Generated through a DDR register within the VIRTEX II IOB with inputs connected to logic-high and logic-low.
frameouta_p frameouta_n	OUT LVDS_25	LVDS frame output. Generated through a DDR register in a VIRTEX II IOB. The output is logic-high for two clock periods and low for the next two.
clkout_locked	OUT LVCMOS25	Clock synchronization output between FPGAs. Provides an indication to the LT FPGA when the RLDRAM FPGA has locked its DCM to its LVDS clock input.

Name	Type	Function
Test Configuration Status Bus Interface		
CSR_DATA[15:0]	BIDIR LVCMOS25	Multiplexed address and data bus. Carries a 16-bit address when CSR_ALE=1, read data when CSR_nRd = 0 and write data when CSR_nWR = 0. To/From AHB/CSR bridge slave on LT FPGA.
CSR_CS_b	IN LVCMOS25	Chip select (active low). The current bus cycle is targeting this FPGA when low. From ARM AHB/CSR Bridge in the LT FPGA.
CSR_RD_b	IN LVCMOS25	Read enable (active low). The current bus cycle is a read when low. The FPGA should drive data onto the CSR_DATA bus.
CSR_WR_b	IN LVCMOS25	Write enable (active low). The current bus cycle is a write when low. The data on the CSR_DATA bus is ready and should be latched by the FPGA.
CSR_ALE	IN LVCMOS25	Address latch enable. The CSR_DATA bus carries an address when this is high. It should be latched when asserted.
RLDRAM Interface		
RLDA_DQ[31:0]	BIDIR HSTL_I_18	RLDRAM A DDR data bus to/from RLDRAM.
RLDB_DQ[31:0]	BIDIR HSTL_I_18	RLDRAM B DDR data bus to/from RLDRAM.
RLDAB_AS_N	OUT HSTL_I_18	RLDRAM address strobe. It is shared between the two RLDRAMs.
RLDAB_WE_N	OUT HSTL_I_18	RLDRAM write enable. It is shared between the two RLDRAMs.
RLDAB_REF_N	OUT HSTL_I_18	RLDRAM auto refresh. It is shared between the two RLDRAMs.
RLDAB_BA[2:0]	OUT HSTL_I_18	RLDRAM bank address. It is shared between the two RLDRAMs.
RLDAB_A[18:0]	OUT HSTL_I_18	RLDRAM address. It is shared between the two RLDRAMs.
RLDAB_DM[1:0]	OUT HSTL_I_18	RLDRAM data mask. It is shared between the two RLDRAMs.
RLDA_CLK_P, RLDA_CLK_N	OUT HSTL_I_18	RLDRAM A differential clock.
RLDB_CLK_P RLDB_CLK_N	OUT HSTL_I_18	RLDRAM B differential clock.
RLDA_CS_N	OUT HSTL_I_18	RLDRAM A chip select.
RLDB_CS_N	OUT HSTL_I_18	RLDRAM B chip select.
RLDA_TRI_TEST	OUT HSTL_I_18	RLDRAM A TEST.

Name	Type	Function
RLDB_TRI_TEST	OUT HSTL_I_18	RLDRAM B TEST.
RLDA_DVLD	IN HSTL_I_18	RLDRAM A data valid from RLDRAM.
RLDB_DVLD	IN HSTL_I_18	RLDRAM B data valid from RLDRAM.
RLDA_DQS RLDA_DQS_N	IN LVDS_33	RLDRAM A data strobe from RLDRAM. RLDRAM A data strobe complement from RLDRAM.
RLDB_DQS RLDB_DQS_N	IN LVDS_33	RLDRAM B data strobe from RLDRAM. RLDRAM B data strobe complement from RLDRAM.
Miscellaneous		
SWITCH_IN_P[3:0]	IN HSTL_I_18	General-purpose switch inputs on the RLDRAM module. Currently not used in the design.
UI_LED[7:0]	OUT HSTL_I_18	General-purpose user LEDs on the RLDRAM module. These are used for diagnostic purposes, such as RLDRAM status.

7.2.2 RLDRAM Clocking

The following clocks are used within the RLDRAM FPGA[1]:

- clk_ref_ab – 200 MHz. Reference clock provided by external input.
- clk_ref_ab_n – 200 MHz. Reference clock complement provided by external input.
- rxclk – 200 MHz. Recovered clock and its complement are generated from rxclkina_n and rxclkina_p, which are forwarded from the LVDS transceiver on the LT FPGA. This clock is phase shifted to provide a sampling point in the center of the rxdata valid bit periods. It is also used for LVDS transmit logic on this FPGA.
- rxclk180 – 200 MHz. The clock is the complement of rxclk and is used for LVDS receive logic.
- clkouta_p, clkouta_n – 200 MHz. This is a regenerated differential version of the 200 MHz rxclk forwarded to the LVDS receiver on the LT FPGA.
- sysclk – 50 MHz. Used for the majority of logic within the RLDRAM FPGA including the PFAAE and LVDS SerDes logic. This clock is phase aligned with rxclk.
- sysclk2x – 100 MHz. Used for the FIFO link receive logic. This clock is phase aligned with rxclk.
- wclk0_ab – 200 MHz. This is an internal system clock that clocks most of the registers in the RLDRAM interface, and task manager.

[1] The RLDRAM interface logic is proprietary third party IP. Therefore, explicit details on its clocking structure have been omitted.

- wclk180_ab – 200 MHz. This clock is used with wclk0_ab to clock all of the DDR registers in the RLDRAM interface and task manager.
- wclk90_ab and wclk270_ab – 200 MHz. These clocks are used to generate RLDx_CLK_P and RLDx_CLK_N. The wclk270_ab is also used to clock the output registers for the command and control signals going to the RLDRAMs.

A block diagram for the clock generation and management logic is shown in Figure 7-11.

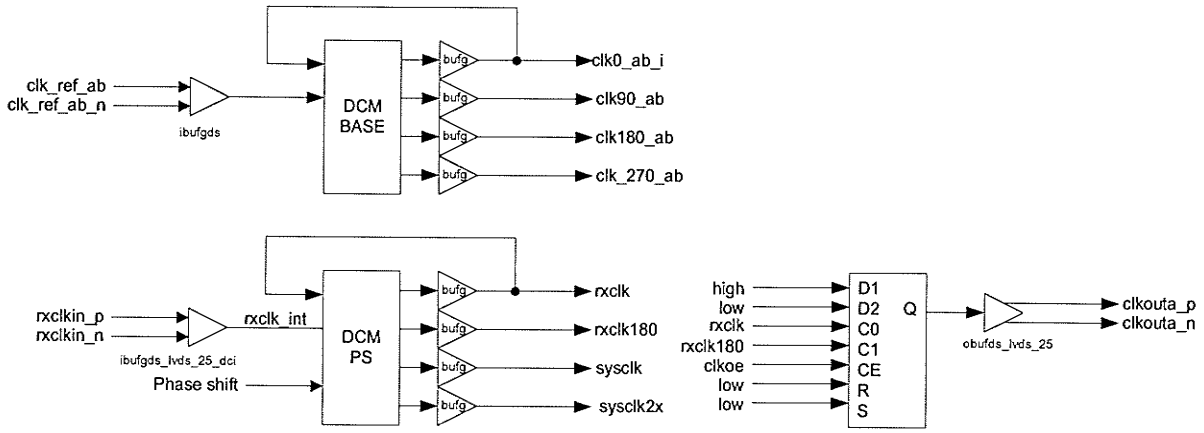


Figure 7-11: RLDRAM Clock Logic

The clock logic has the following features:

- FPGA input clocks are sent through global clock input buffers to minimize skew.
- Clocks generated for use within the RLDRAM FPGA are sent through BUFGs to ensure global routing is used for all clock resources.
- The design uses two digital clock managers. A basic DCM accepts a buffered 200 MHz reference clock and generates phase aligned 50, 100, and 200 MHz system clocks as well as an inverted version of the 200 MHz system clock. A second dedicated phase shifting DCM is used for LVDS receive clocks. This DCM generates a reference clock at the receive port based on the forwarded differential clock. A fixed phase shift is required to provide a sampling point near the center of the receivers valid bit periods.

7.2.3 Control Status Register Glue Logic

The purpose of this block is to interface the external CSR multiplexed address/data bus with the internal FPGA non-multiplexed bus. This block has the following features:

- Operates at 50 MHz in the system clock domain
- De-multiplexes the 16-bit address from the external CSR bus and distributes the address to the top-level and PFAAE. The address is latched from the external CSR data when the *address latch enable* (ALE) is asserted.

- Provides registers for version ID, DCM status, debug, general-purpose use and PFAAE configuration. Key configuration fields for the PFAAE include the mode of operation and filter response format.
- A distributed address decoding scheme is used in which each block connected to the internal CSR bus is responsible for decoding its own address space. The address map is shown in Table 7-8.

Table 7-8: Configuration Status Register Address Map

Name	Base Address
Top-Level Registers	0x0000
PFAAE Registers	0x8000

7.2.4 Packet Filter Acceleration Assist Engine

The Packet Filter Acceleration Assist Engine is the heart of the RLDRAM FPGA logic. It instantiates four packet classification assist agents or modes (build, filter, loop and user command) and multiplexes access between them and itself to the RLDRAM Task Manager, LVDS interface and internal DPRAMs. Figure 7-12 provides a block diagram of the PFAAE.

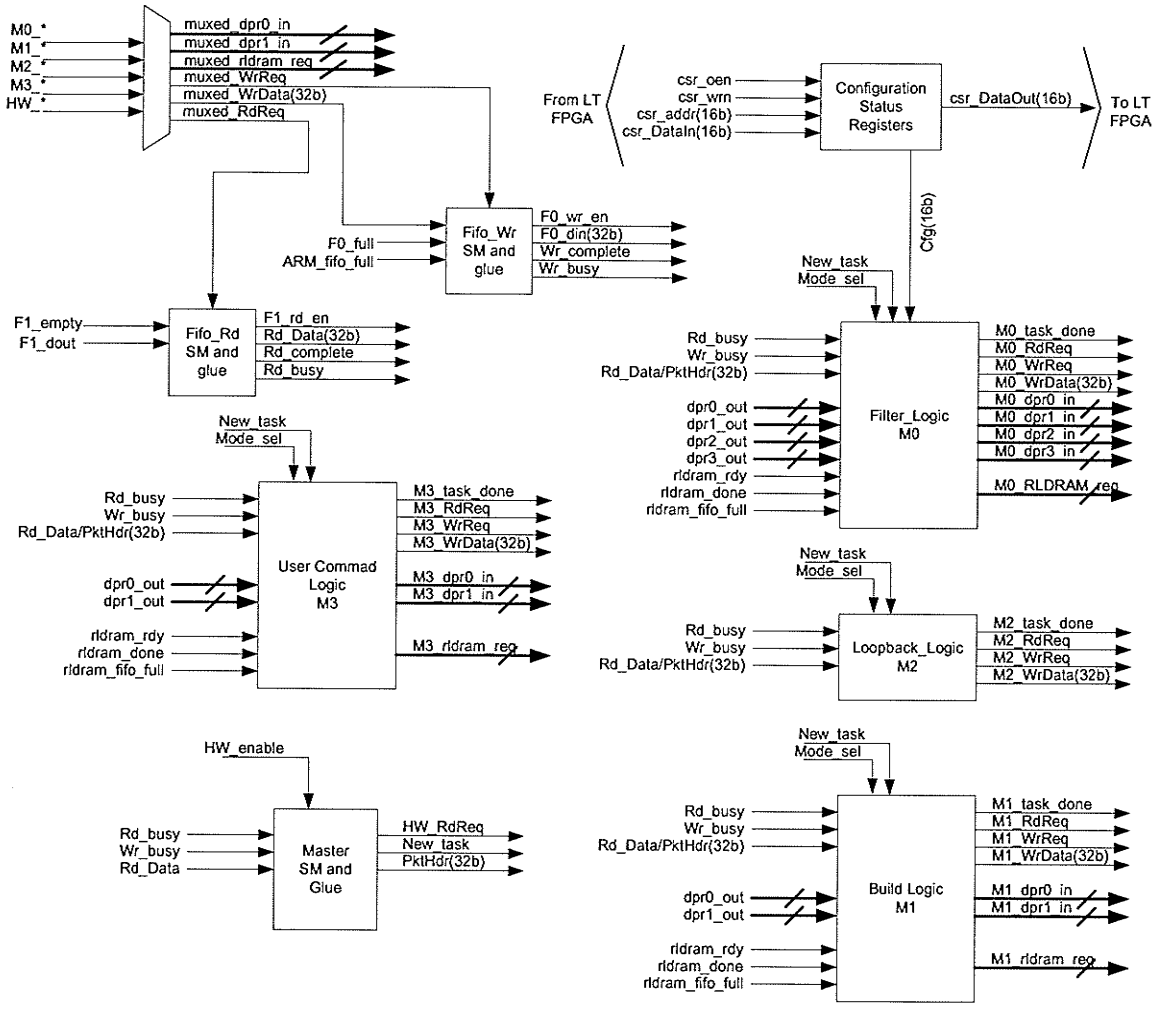


Figure 7-12: PFAAE Block Diagram

Table 7-9 provides a brief description of each of the functional blocks shown above. The key IP blocks that perform the build and filter operations, are described in more detail in subsequent sections.

Table 7-9: PFAAE Functional Block Summary

<table>
<tr><th>Block</th><th>Function</th></tr>
<tr><td>FIFO Read State Machine and Glue</td><td>Performs read operations from the ingress FIFO that interfaces with the SerDes logic.</td></tr>
<tr><td>FIFO Write State Machine and Glue</td><td>Performs write operations to the egress FIFO that interfaces with the SerDes logic.</td></tr>
<tr><td>Configuration Status Register Logic</td><td>Decodes the CSR bus address. CSR registers define the active packet filter assist agent and provide configuration control of the individual agents.
<table>
<tr><th>Mode</th><th>Mode Setting</th><th>Description</th></tr>
<tr><td>M0</td><td>'b00</td><td>Filter Mode</td></tr>
<tr><td>M1</td><td>'b01</td><td>Build Tree Mode</td></tr>
<tr><td>M2</td><td>'b10</td><td>Loopback Mode</td></tr>
<tr><td>M3</td><td>'b11</td><td>User Command Mode</td></tr>
</table>
They also provide an enable for a quick filter option. When this option is turned on, resulting CBVs are not returned to the software. Instead only headers are returned.</td></tr>
<tr><td>Master State Machine and Glue</td><td>When none of the agents are busy, this block is responsible for polling the asynchronous input FIFO for requests. Once a packet is received, control is passed to the active agent. Control is returned when the active agent finishes its task.</td></tr>
<tr><td>Build Logic</td><td>Performs hierarchical compression on a list of rules that make up a bit-vector for a key in the search tree. The CBV is written to RLDRAM and a pointer is returned to the search tree software.</td></tr>
<tr><td>Filter Logic</td><td>Accepts CBV pointers, which make up a search and retrieves them from RLDRAM. Filter mode also performs the OR operation directly on CBVs and returns a search result according to the filter configuration.
<table>
<tr><th>Filter Setting</th><th>Description</th></tr>
<tr><td>'b0</td><td>Normal Operation (entire CBV returned)</td></tr>
<tr><td>'b1</td><td>Quick Filter (only header returned)</td></tr>
</table></td></tr>
<tr><td>User Logic</td><td>The main function of this block is to provide the user with a diagnostic tool allowing software to perform basic tasks of reading and writing to DPRAM and transferring to RLDRAM.</td></tr>
<tr><td>Loopback Logic</td><td>The main function of this block is to act as a diagnostic tool to verify that the SerDes logic between the two FPGAs is functioning as expected. On the RLDRAM FPGA each packet that is received at the ingress FIFO is looped back directly to the egress FIFO.</td></tr>
</table>

7.2.4.1 Build Tree Logic

This module interconnects the main components required to convert lists of rules into hierarchical compressed bit-vectors and stores them in external RLDRAM. Figure 7-13 provides a block diagram of this agent.

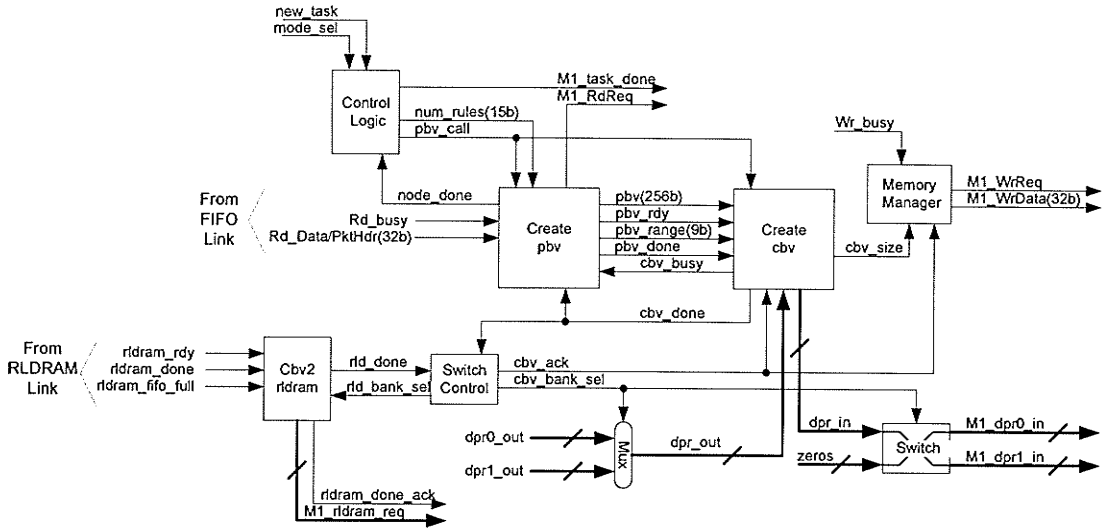
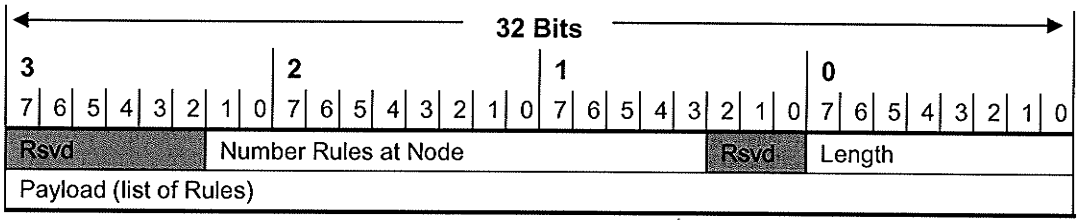


Figure 7-13: Build Mode Block Diagram

This block has the following features:

- Operates at 50 MHz in the system clock domain.
- This build agent is enabled when the mode bits in the CSR space are set to 'b01. When enabled, it accepts packets from the LT in the format defined in Figure 7-14.



Length: represents the total packet length in 32-bit words including the packet header.
Number of Rules at Node: indicates the total number of rules associated with a key in the search tree.
Payload: represents a list of rules for at a key in 32-bit format and forwarded in ascending order.
Rsvd: reserved for future enhancements.

Figure 7-14: Build Mode Receive Packet Format

- Since there may be greater than 254 rules associated with a key in a node, it may take multiple packets to build one CBV. In this case, the build logic is responsible for tracking the number of rules received against the total number of rules at the node.
- The create PBV block performs the first stage of the build operation. Its two main functions consist of reading the rules associated with a particular key and creating *partial bit-vectors* (PBVs). Each PBV is a block of 256 bits representing a small range of an uncompressed bit-vector. Its location within the uncompressed bit-vector is identified by its BV range or interval. For a rule-set of 32K, there are a maximum of 128 partial bit-vectors. However, only PBVs with asserted rules need be created.

- When the first rule is read, PBV control logic will initialize a PBV and use the upper half of the rule ID[15:8] to compute an initial BV range. The lower half of the rule ID[7:0] is forwarded to the set PBV logic. The set PBV logic determines where a rule ID should be set within the current PBV. This logic can set a bit in one clock cycle by performing parallel comparisons.
- To narrow the region of the PBV that needs to be updated with each incoming rule, the PBV is broken into 16 blocks. A greater-than comparison is performed on bits [7:4] of the incoming rule ID with each of the 16 possible range locations. An equality test is also performed between rule ID bits [3:0] and values [0:15]. The result is a compare vector and a one-hot encoded set-bit-vector. Figure 7-15 illustrates this process.
- **Example**: Given a rule ID of: 'b0001 0000 0111 0000 or (0x1070), a bit is set in the compare vector when the PBV range > rule ID [7:4]. The transition from 1 to 0 indicates in which 16-bit block the current rule ID [3:0] should be set. In the set-bit-vector, a bit is asserted only when an exact match is found.

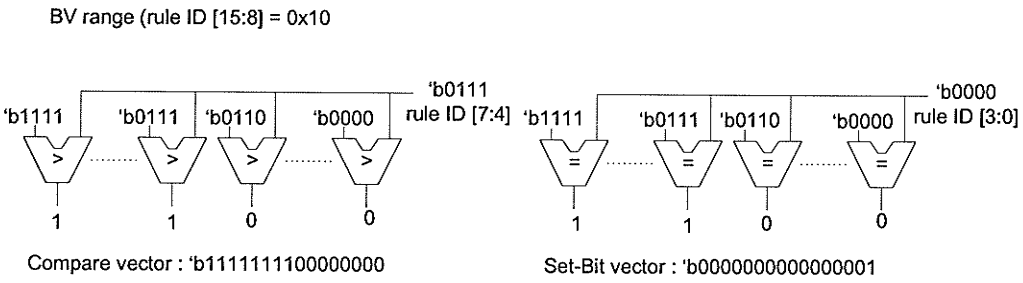


Figure 7-15: Partial Bit Vector Example: Part 1

- Together, the above vectors are used to update the PBV. Parallel comparisons are again performed to determine if each 16-bit block should retain its current value or if it should be updated. If a low to high transition is encountered in the compare vector a mux select line is asserted and a PBV block is updated by performing a parallel OR operation of the set-bit-vector with the current value. In cases where there are no transactions the PBV blocks retain their current value. The only exception is where there are no transitions because the end range is the match. In this case the end bank of 16 bits is updated with the OR operation. Figure 7-16 illustrates the above procedure.

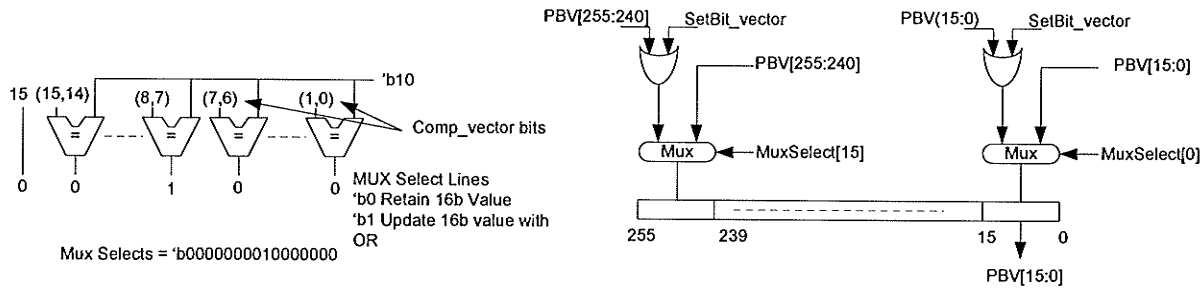


Figure 7-16: Partial Bit Vector Example: Part 2

- The process in the previous example is repeated until a new PBV range is detected or the last rule has been read by the PBV control logic. When this occurs, the current PBV is complete and is forwarded to the create CBV logic along with the BV range. Therefore, the CBV generation can be pipelined with the PBV generation.
- The create CBV block performs the second stage of the build procedure. This block creates the compressed bit-vector representation in 32-32-32 hierarchical format (with expansion capabilities to 64-32-32) from partial bit-vectors and stores them in internal dual-port memory 64w x 512d. To improve performance and pipeline operations, two dual-port memories are used to build the compressed bit-vectors. The create CBV operation and cbv2rldram logic alternate between the two (i.e. while the CBV module is writing to DPRAM 0 the cbv2rldram logic is writing DPRAM 1 to the RLDRAM). Each time a new key is encountered the memories are swapped.
- For a 32-32-32 CBV, a PBV of 256 bits will allow for up to 1 bit set in the root level, 8 bits set in level 2, and 8 complete 3rd level vectors. Figure 7-17 provides an example of the create CBV operation for a single PBV.
- **Example:**

```
BV Range: 0x04
PBV:
0x00000000 00000000 00000000 00000000 00000000 00010000 00000000 000F0000

 MSB            LSB
0x0000000000000002   Root (bit set based on range alone)
                 /\
                /  \
              0x00000005            Level 2 Count=1
                       /\
                      /  \
            0x00001000   0x000F0000       Level 3 Count=1
```

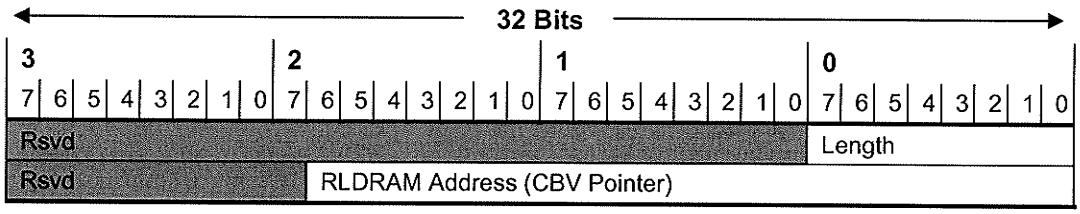
Figure 7-17: Create CBV Example

- The above CBV is stored in internal DPRAM in the format shown in Table 7-10. In the internal memory, space is reserved for all possible level 2 and level 3 vectors. To simplify the filter operation, counts for the number of level 2 and level 3 64-bit words are also stored. This adds a constant amount of overhead to each CBV but significantly reduces the complexity and time to perform filter tasks.

Table 7-10: CBV Representation in DPRAM

Address	Description	Contents
0x000	L1 Vector	0x00000000 0x00000002
0x001	L2 Count[63:32], L3 Count[31:0]	0x00000001 0x00000002
0x002	Start of L2 Vector(s)	0x00000005 0x00000000
:	:	0xXXXXXXXX 0xXXXXXXXX
:	:	0xXXXXXXXX 0xXXXXXXXX
0x023	End of L2 Vector(s) 32-32-32 format	0xXXXXXXXX 0xXXXXXXXX
0x024	Start of L3 Vector(s)	0x000F0000 0x00001000
:	:	0xXXXXXXXX 0xXXXXXXXX
0x1FF	End of L3 Vector(s)	0xXXXXXXXX 0xXXXXXXXX

- The memory manager is responsible for calculating the next available RLDRAM pointer for each CBV and returning it to the ARM processor to be stored in the search tree. The format of the return packet is shown in Figure 7-18.



Length: represents the total packet length in 32-bit words including the packet header. It is fixed to 0x02 for the CBV pointer return packet.
RLDRAM Address: indicates the RLDRAM start address for a CBV (i.e. pointer to a CBV).
Rsvd: reserved for future enhancements.

Figure 7-18: Build Mode Transmit CBV Pointer Packet Format

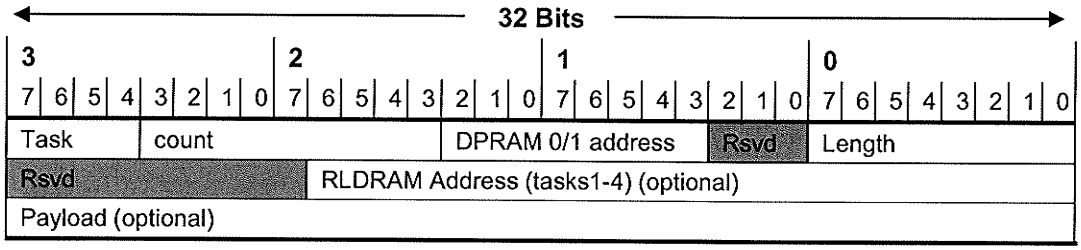
- Once a new CBV is created and the cbv2rldram logic is idle, the switch control logic is responsible for sending a request to the cbv2rldram block to transfer the internally generated CBV to the RLDRAM.
- The cbv2rldram block is responsible for transferring the CBV from internal memory while removing any unused memory locations for levels 2 and 3. Therefore, in the external memory, the CBV shown in Table 7-10 would represent a minimum-sized CBV occupying a total of 4 64-bit words.

7.2.4.2 User Command Mode

The function of this block is primarily diagnostic. It provides processor driven control over basic datapath operations and the ability to dump memory contents. This block has the following features:

- Operates at 50 MHz in the system clock domain.

- This agent is enabled when the mode bits in the CSR space are set to 'b11. When enabled it accepts packets in the format defined in Figure 7-19



Length: represents the total packet length in 32-bit words including the packet header.
DPRAM address: indicates the DPRAM 0 or 1 start address depending on the requested task.
Count: represents the number of 64-bit locations to read or write.
RLDRAM Address: indicates the RLDRAM start address depending on the requested task.
Payload: represents data for write requests.
Rsvd: reserved for future enhancements.
Task: provides an indication of the desired datapath operation. Currently, there are eight tasks defined for user mode with the capability to support up to 16.

Figure 7-19: User Mode Packet Format

Table 7-11 defines the tasks available in User mode.

Table 7-11: User Mode Tasks

Task Code	Description
'b0000	NOP
'b0001	Read from dual-port RAM 0 and write to external RLDRAM.
'b0010	Read from dual-port RAM 1 and write to RLDRAM.
'b0011	Read from RLDRAM and write to dual-port RAM 0.
'b0100	Read from RLDRAM and write to dual-port RAM 1.
'b0101	Read from ingress FIFO and write to dual-port RAM 0.
'b0110	Read from ingress FIFO and write to dual-port RAM 1.
'b0111	Read from dual-port RAM 0 and write to egress FIFO.
'b1000	Read from dual-port RAM 1 and write to egress FIFO.

7.2.4.3 Filter Mode

Filter mode provides the main hardware acceleration for the packet classifier. This block retrieves CBVs from the RLDRAM into local memory and performs the logical OR on the respective CBVs. Figure 7-20 provides a block diagram of the main components within this block.

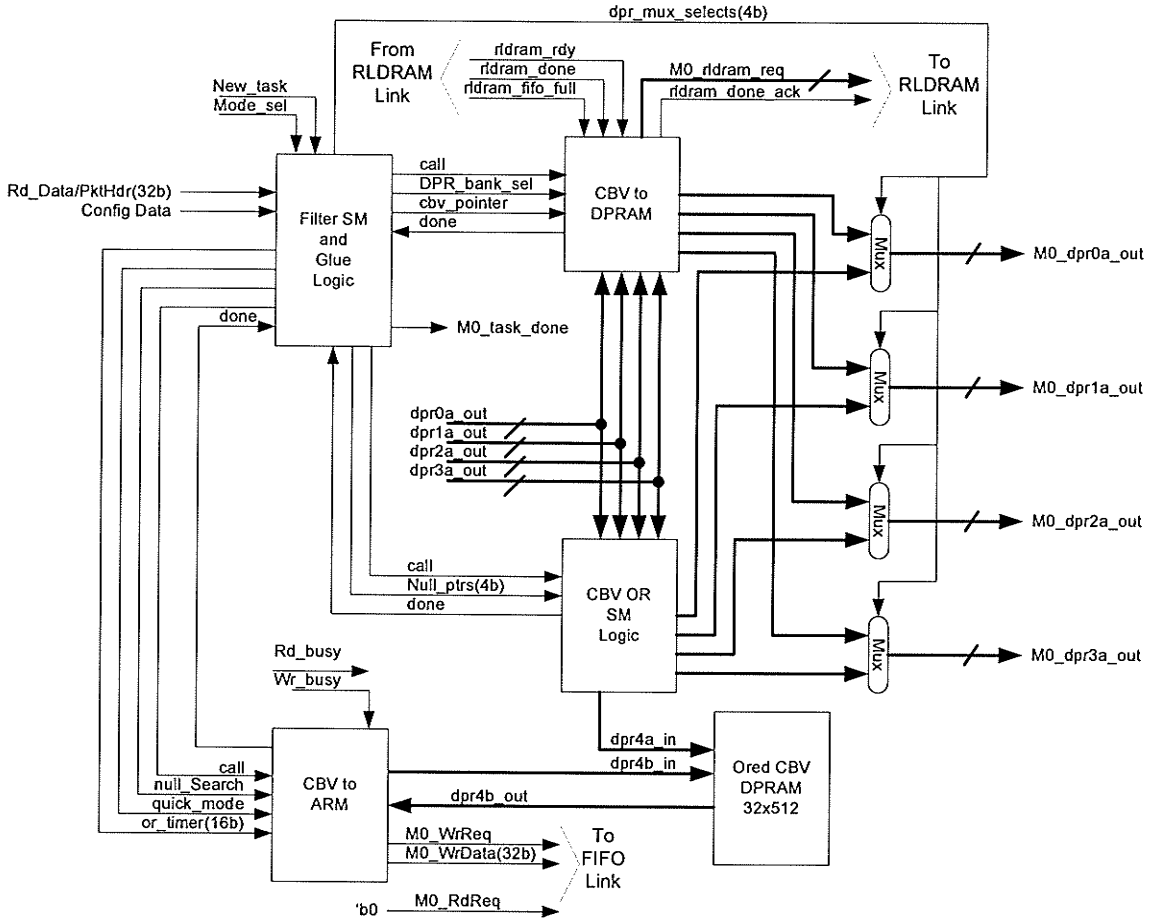
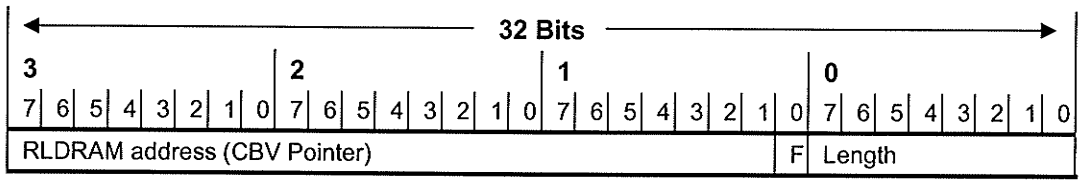


Figure 7-20: Filter Mode Block Diagram

This block has the following features:

- Operates at 50 MHz in the system clock domain.
- Contains all required registers for configuring, managing and monitoring the status of the filter operation.
- This agent is enabled when the mode bits in the CSR space are set to 'b00. When enabled, it accepts packets in the format defined in Figure 7-21.



Length: represents the total packet length in 32-bit words including the packet header.
F: represents a null flag, when there was no hit at a particular level in the search tree.
RLDRAM Address: represents a pointer to the start of a CBV in the external memory

Figure 7-21: Filter Mode Receive Packet Header Format

- The filter SM is responsible for parsing the packet header and determining if a NULL pointer or CBV pointer has been received. Up to a maximum of four CBV pointers can be accepted per search. For each CBV pointer received the cbv2dpram controller is called to retrieve the CBV.
- When called upon, the cbv2dpram block is provided a CBV pointer and dual-port RAM select lines. It is responsible for providing requests to the RLDRAM Task Manager to retrieve a CBV and return it to one of four internal dual-port memories. This is a one or two-step process depending on the size of the CBV. Initially 4 64-bit words will be read from the RLDRAM (size of a minimum CBV) beginning at the pointer address. Using the second 64-bit value the overall size of the CBV is determined using Equation 7-1 and the remainder is retrieved.

Equation 7-1: CBV Calculation for Remaining 64-bit Words

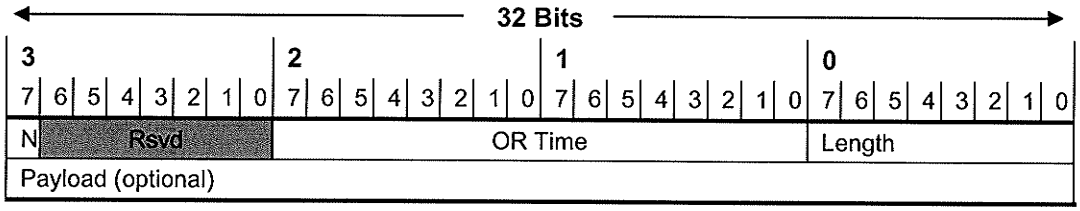
$$Remaining64bitWords = l2\_count + l3\_count - 4$$

- The RLDRAM task is determined based on the dual-port RAM select lines provided by the filter SM. The following table outlines the various tasks.

Table 7-12: RLDRAM Task Definitions

RLDRAM Task	DPRAM Select Lines	Description
'b0100	'b0001	Read from RLDRAM and write to Dual Port Memory 0
'b1000	'b0010	Read from RLDRAM and write to Dual Port Memory 1
'b0101	'b0100	Read from RLDRAM and write to Dual Port Memory 2
'b0110	'b1000	Read from RLDRAM and write to Dual Port Memory 3

- Once all of the CBVs are retrieved, the CBV OR logic is called upon to perform the OR operation, expanding the vector where necessary. Look-up tables and bit-level parallelism are used to quickly find asserted bits in the first two levels of the CBV. As the OR operation proceeds, the ORed result is stored in a 5th local DPRAM.
- The filter state machine is also responsible for timing the OR operation in 50 MHz clock cycles and sending the result to the cbv2arm logic.
- Once the OR operation is complete, the cbv2arm block is called and an interrupt is asserted indicating the task is complete. This re-enables the cbv2dpram and OR tasks for the next search operation.
- The cbv2arm block supports either quick or normal mode of operation based on configuration settings. In normal mode, it is responsible for writing the entire ORed CBV result back to the SW along with the time spent performing the OR operation. In quick mode, it foregoes writing the entire CBV and simply returns a single word packet indicating the time spent performing the OR operation. The format of the return packet(s) is illustrated in Figure 7-22.



Length: represents the total packet length in 32-bit words including the packet header.
OR Time: indicates the time the PFAAE spends performing the CBV OR task in 50 MHz clock cycles.
N: is a flag that indicates a null search was requested.
Payload: this field is only used in normal mode to return the resulting ORed CBV.
Rsvd: reserved for future enhancements.

Figure 7-22: Filter Mode Transmit Packet Header Format

7.2.5 RLDRAM FIFO Link, Task Manager and Controller

The RLDRAM FIFO link, Task Manager and controller combine to communicate with the external RLDRAMs. The FIFO allows the PFAAE to queue up several data transfer requests that are read out by the Task Manager. The Task Manager forwards these requests to the controller in addition to issuing regular memory refresh commands. The Task Manager also controls the datapath between the RLDRAM and internal memory. Finally, the controller formats the commands into the appropriate address, data, and control line signaling expected by the memory. Since these blocks contain some proprietary third party IP further details are omitted.

7.3 Xilinx Implementation

As System-on-Chip designs increase in complexity, the integration of IP blocks into the design becomes critical. As shown in Figure 7-23 and Table 7-13, this section addresses the procedures for implementing the design (place & route) in Xilinx FPGAs using Leonardo Spectrum and the Xilinx implementation tools.

Due to the 200 MHz requirements for the LVDS and RLDRAM memory buses, placement became a critical factor to meet timing constraints on both FPGAs. The placement was managed using a number of techniques. *Relationally placed macros* (RPMs) and *relative location constraints* (RLOC) are used for the most critical routes, *location constraints* (LOC) for locking down placement of I/O, clock buffers, RAMs and DCMs, and area_groups to force similar blocks close together. The next two sections illustrate the utilization and floorplans for the RLDRAM and LT FPGAs respectively.

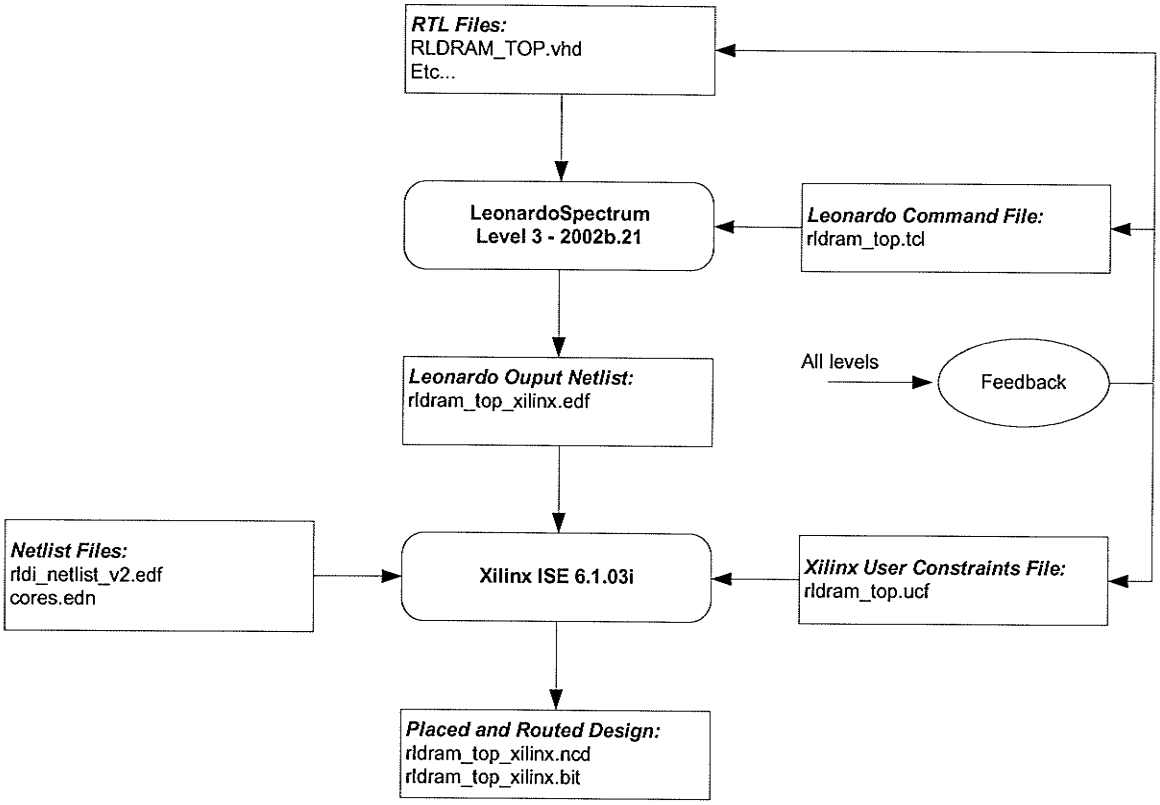


Figure 7-23: Implementation Process for the RLDRAM FPGA Design

Table 7-13: Implementation Files for the RLDRAM FPGA Design

File	Function
rldram_top.vhd etc.	Verilog and VHDL RTL source files.
rldram_top.tcl	User generated Leonardo Spectrum command file for creating a synthesis project, adding source files and generating an EDIF.
rldram_top.ucf	Xilinx user constraints file containing device, timing and placement constraints for the design.
rldi_netlist_v2.edf	An EDIF recognized netlist representation for the RLDRAM core.
rldram_top_Xilinx.edf	An EDIF recognized netlist for the user application logic that is produced by synthesizing the source files with Leonardo Spectrum.
cores.edn	An EDIF recognized netlist representation for Xilinx cores used in the design.
rdlram_top.Xilinx.ncd	A Xilinx placed and routed design file.
rdlram_top.Xilinx.par	A Xilinx report including summary information for all of the iterations of placement and routing.
rldram_top_Xilinx.bit	A bitstream for Xilinx device configuration, which is produced by Xilinx BitGen.

In the above, Figure 7-23 and Table 7-13 depict the process, files and flow used to implement the RLDRAM FPGA portion of the design. A similar approach was used for the LT FPGA.

7.3.1 RLDRAM FPGA Resource Utilization

The RLDRAM FPGA design has been placed and routed in an XC2V1000-5FF896 device, yielding the following utilization numbers and floorplan:

Table 7-14: RLDRAM FPGA Utilization Summary

Family	Device	Slices	GCLKs	Bonded IOBs	Block RAMs	DCMs
Virtex-II	XC2V1000-5FF896	5,015	11	182	18	4

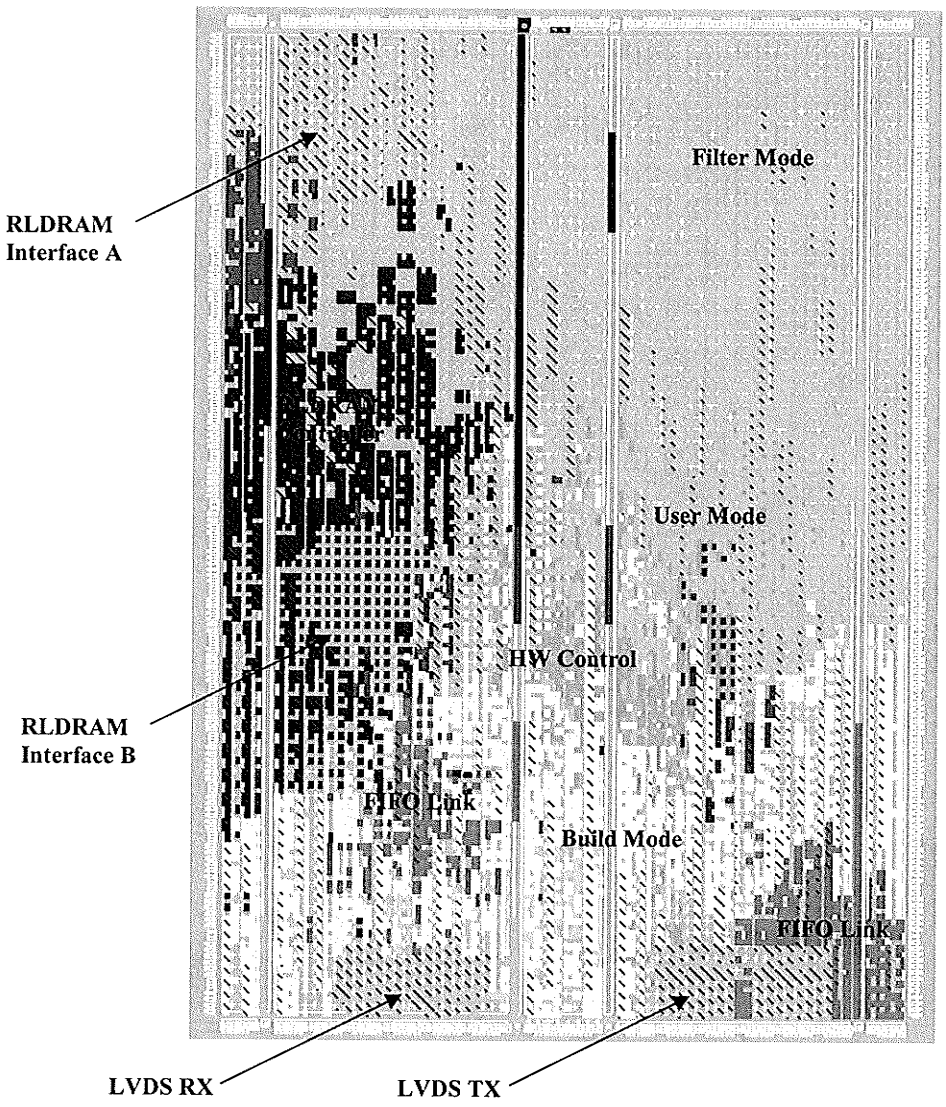


Figure 7-24: RLDRAM FPGA Floorplan

7.3.2 Logic Tile FPGA Resource Utilization

The LT design has been placed and routed in an XC2V6000-4FF1517 device, yielding the following utilization numbers and floorplan:

Table 7-15: LT FPGA Device Utilization Summary

Family	Device	Slices	GCLKs	Bonded IOBs	Block RAMs	DCMs
Virtex-II	xc2v6000-4ff1517	1,293	7	347	7	2

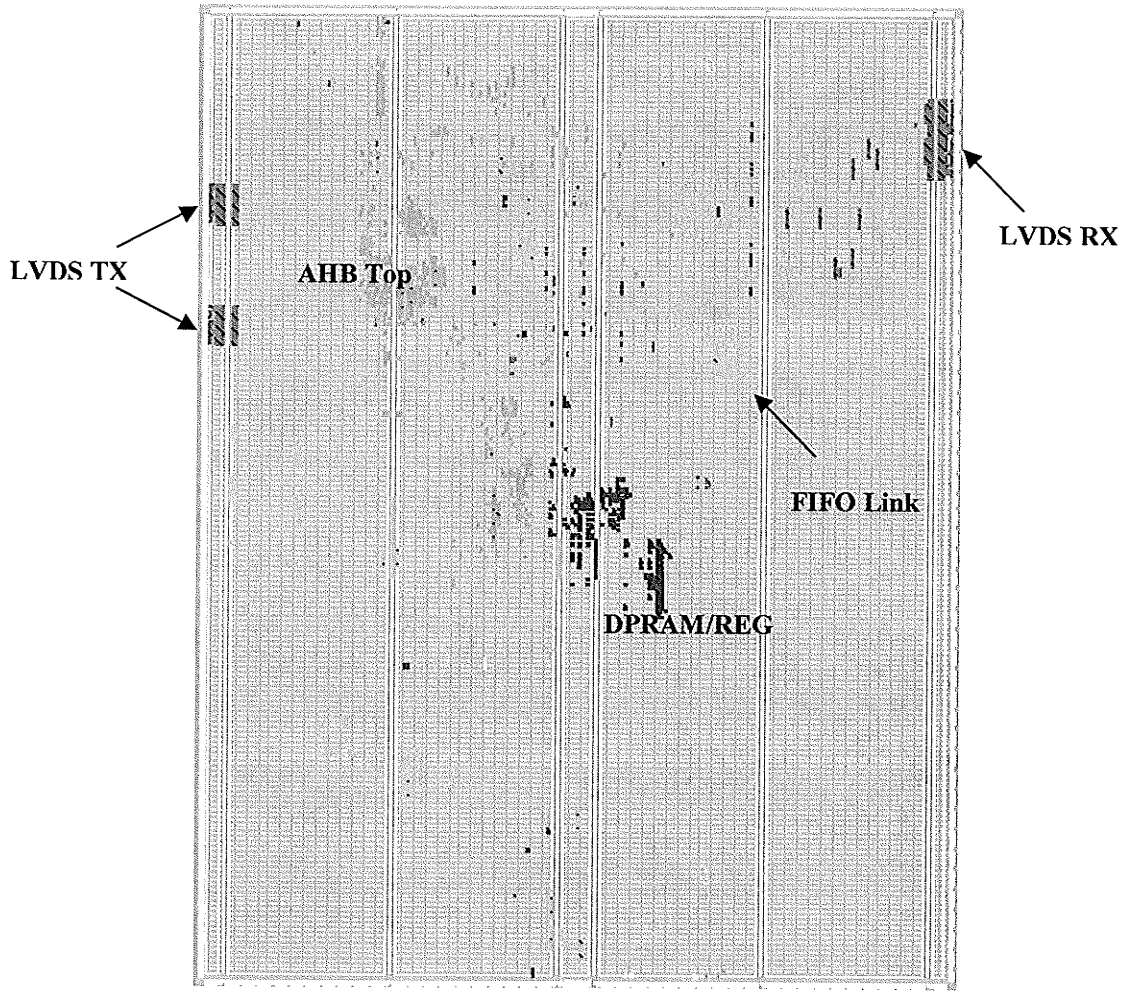


Figure 7-25: LT FPGA Floorplan

Chapter 8: Simulation

Chapter 8 explores the simulation approach used to verify design functionality from a hardware standpoint. The progression to platform-based SoC designs has resulted in a dramatic increase in design complexity and challenges. Companies using SoC techniques are facing extremely long simulation times and many verification cycles to validate their products. In fact, verification has been identified as the key obstacle in reducing overall design cycle time and meeting aggressive time-to-market goals [43]. Intelligent simulation techniques are required to overcome these challenges.

8.1 Functional Simulation

Prior to hardware and software integration on the ARM platform, the functional behavior of the entire hardware design was verified. Figure 8-1 and Table 8-1 depict the process, flow and files used to simulate the design using Mentor Graphic's ModelSim tool.

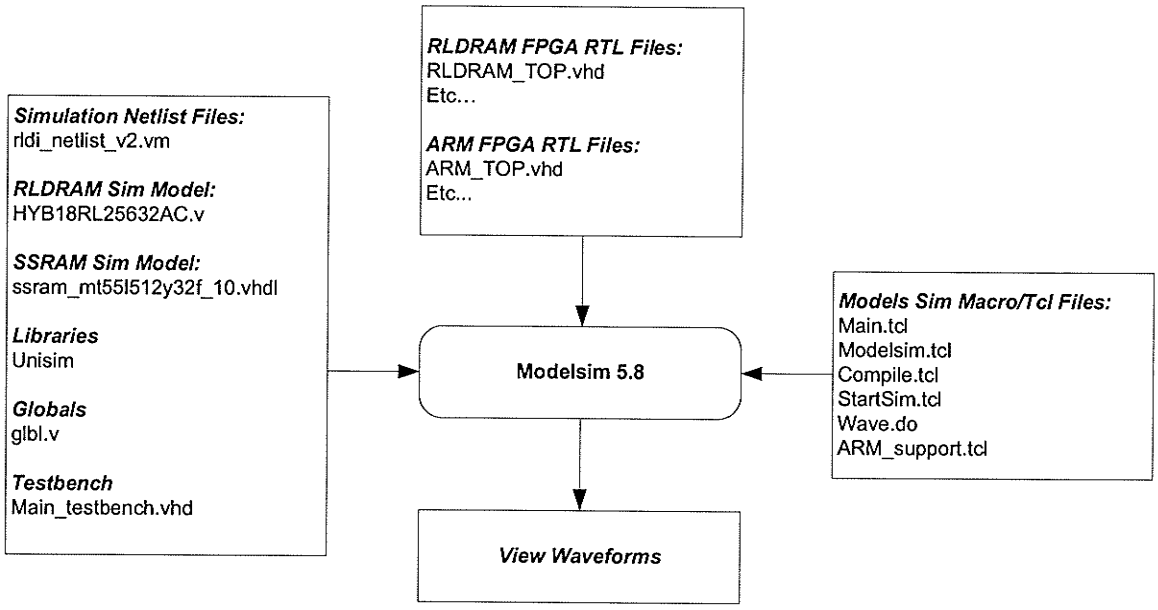


Figure 8-1: Functional Simulation Process for the CBV Packet Classification Design

Table 8-1: Functional Simulation Files for the CBV Packet Classification Design

File	Function
Main.tcl	Is executed to source all of the files required to start a simulation.
Modelsim.tcl	Contains preferences for the ModelSim GUI.
Compile.tcl	Compiles all design files, simulation models and the main testbench.
StartSim.tcl	Opens the simulator's signal and wave windows, maps signals and ports to Tcl variables, performs a reset of the system and runs the simulation for 1 us.
wave.do	ModelSim macro file to load waveform with signals of interest.
ARM_support.tcl	Contains bus functional models to perform AMBA bus transactions.
Main_testbench.vhd	Instantiates the top-level entities for the logic tile and RLDRAM FPGAs, models for the RLDRAM and ZBT memory and clock generators.
Unsims, Glbl.v	Xilinx behavioral simulation libraries. These need to be pre-compiled before running simulations.
HYB18RL25632AC.v	Simulation netlist for RLDRAM.
ssram_mt551512y32f_10.v hdl	Simulation netlist for SSRAM.

For this testing the focus is on the system bus interface. For the hardware to function correctly the logic connected directly or indirectly to the microprocessor or bus must obey the AMBA AHB bus protocol. If the rules of the bus are obeyed, then the details of the software tasks being performed by the microprocessor are not important. In simulation, the tasks performed by the microprocessor can be viewed simply as a series of memory reads and writes at the device pins. Traditionally, hardware engineers have used bus functional models to abstract the role of the microprocessor into a model of its bus. This type of verification is commonly referred to as transaction-based verification since it interprets the microprocessor as a bus transaction generator [30], [37].

The major advantage of this approach is the level of abstraction is raised to the transaction level as opposed to signal or pin. This eliminates the need for detailed testbenches with large test vectors and increases the efficiency of the verification process. In addition, if bugs are found, the signal and pin level are still accessible.

As a starting point for transaction-based verification, the *bus functional model* (BFM) needs to be developed which provides a method of running transactions on hardware design interfaces by driving signals on the interconnects according to the requirements of the interface protocol. Typically, this can be done in HDL languages or at a higher level using languages such as C++ or Tcl. ModelSim lends itself to Tcl, which is the approach taken here. Below is an example of a

model to perform an AMBA AHB read where each of the signals is tied to the ModelSim simulator.

```
proc AMBARead {addr} {

        set addr [format %08X [expr $addr]]
        force $::HWRITE_sig         0       $::d ns
        force $::HTRANS_sig         16#3    $::d ns
        force $::HADDR_sig          16#$addr $::d ns
        force $::HREADY_sig         1       $::d ns
        force $::HSIZE_sig          16#2    $::d ns
        force $::HDRID_IN_sig       16#E    $::d ns
        set ::d [expr $::d + $::clk_period]

        force $::HWRITE_sig         0       $::d ns
        force $::HADDR_sig          16#0    $::d ns
        force $::HREADY_sig         Z       $::d ns
        force $::HTRANS_sig         16#0  $::d ns
        set ::d [expr $::d + $::clk_period]
        if {$::d != 0} {
            run $::d ns
            set ::d 0
        };# if

        #examine the hready line
        while { [ expr ([examine $::HREADY_sig] == 0)]  } {
              run $::clk_period ns
        }
        set result "[ format 0x%08X 0x[examine -hex $::HDATA_sig]]"
        return $result

};# AMBARead
```

Figure 8-2: AMBA AHB Bus Functional Model for Single Reads

Using this approach, test scenarios were developed and implemented in Tcl to perform all of the basic system operations described in Chapter 7. For example, Figure 8-3 illustrates a simple test scenario for writing and reading the first location of DPRAM inside the logic tile FPGA.

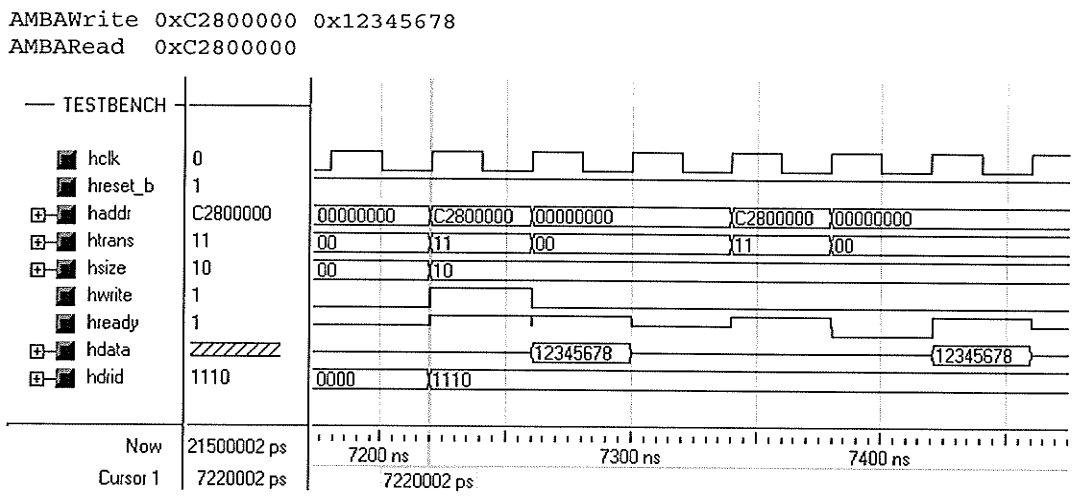


Figure 8-3: AMBA AHB Write and Read Timing Diagram

8.2 Timing Simulation

The design does not support timing simulation, primarily because the RLDRAM interface core does not support timing simulation.

8.3 Summary

Although the transaction-based simulations provide a reasonable level of confidence in the design, it is still software based and ultimately too slow to perform exhaustive verification, particularly with large rule-sets. Thankfully, once the design is implemented on the RPP a high degree of coverage can be achieved in addition to a realistic performance analysis. The next two chapters detail the hardware/software integration process, verification and benchmarking on the platform.

Chapter 9: Hardware and Software Integration

Chapter 9 addresses the integration of hardware and software. The progression to SoC design has led to the complete integration of microprocessor subsystems with custom hardware and software onto a single silicon device. This advancement has raised the complexity, risk and cost involved in developing such devices, bringing about change in traditional design and verification methodologies. Many companies now require that the hardware and its associated software work correctly before chip fabrication. This is a driving force behind platform-based design and can lead to a reduction in late design errors and improve overall efficiency in time-to-market.

The SoC design flow outlined in Chapter 2 meets the above requirement while allowing firmware development to proceed in parallel with the hardware design. To ensure tight integration, the flow relies not only on well-defined specifications and protocols, but also on early integration and testing capabilities provided by the platform. The platform enables:

- Early creation of a hardware model of the chip.
- User detection and correction of bugs early in the design cycle.
- The ability to develop, debug and benchmark application software running at speeds comparable to the target system.
- The ability to connect in-circuit emulators.

Essentially, the advantage of the platform design is to aid in the development and debug of application software as the hardware design becomes available. As the design progresses, tests can be ramped up.

Initially, a user interface driven test mode was developed to verify all of the datapaths through the system. This testing was performed in an incremental fashion and included verification of the hardware and software required to send data transfers along the AMBA, LVDS and CSR buses. It also included complete memory testing for the DPRAMs and RLDRAMs involved in PFAAE operations. Once all of the basic functional datapaths were proven to be reliable a complete system-level verification flow was established for the CBV packet classification algorithm.

9.1 System-Level Verification

To verify the complete functionality and performance of the integrated design, a collaborative system-level test plan was developed to exercise the key functional requirements from the specification. Collectively, the set of figures in Section 9.1 illustrate key aspects of the system-level verification flow including rule generation, pre-processing and search operations. When describing the flow, the following conventions were adopted:

- Asterisks (*) are used to delimit the location of variables inserted into file names.
- Most of the files are appended with a mode, seed and rule size to organize the test results. Mode refers to the two-dimensional pair of rule fields used in the test, seed refers to the random seed used to generate the trial rule-set, and rule size indicates the number of rules used for the test.

9.1.1 Verification Flow

The first major step to verify and benchmark the packet classifier was to acquire or develop realistic rule-sets. It was found to be extremely difficult to acquire rule-sets to test against since corporations are reluctant to publish these and there is no corpus available providing commercial firewall rule-sets. However, as part of the project the software team developed a Microsoft Visual C++ rule generator to create synthetic rule-sets of varying size and distribution statistics. Figure 9-1 depicts the role of the rule generator in relation to the overall flow.

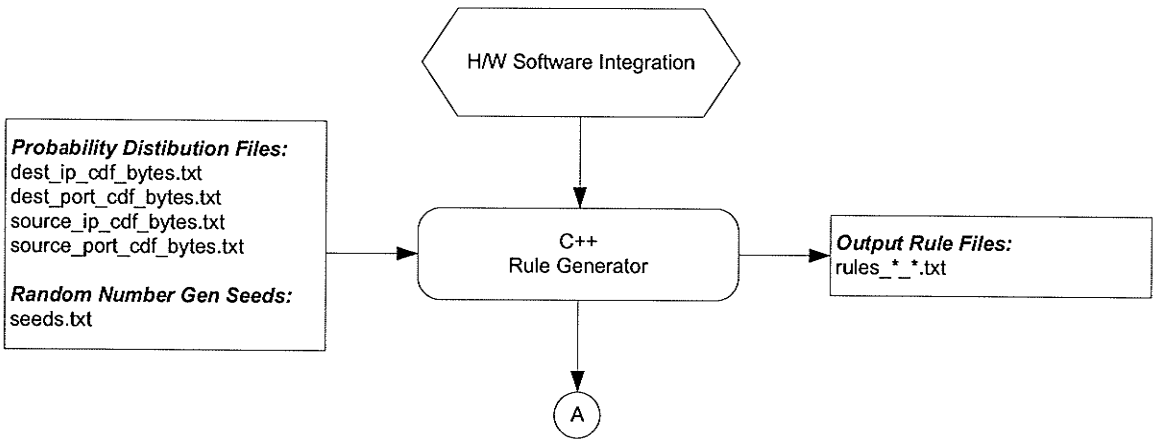


Figure 9-1: System Verification Flow Part 1: Rule Generation

The rule generator is a *graphical user interface* (GUI) based development tool. It accepts user arguments for random seeds, the number of rules to create, *cumulative distribution functions* (CDF) for each field and the path and prefix names to store the rules. In total it creates ten

different inbound and outbound rule-sets for a given CDF, each based on a different seed. Table 9-1 details the files associated with this portion of the flow.

Table 9-1: Verification Flow Part 1 Files

File	Function
seeds.txt	Holds ten different seeds for the random number generator. It is required to perform 10 trials per rule-set distribution to establish a 90% confidence interval in the algorithm performance results.
dest_ip_cdf.txt[1]	Defines the destination IP field cumulative distribution function.
source_ip_cdf.txt	Defines the source IP field cumulative distribution function.
dest_port_cdf.txt	Defines the destination port field cumulative distribution function.
source_port_cdf.txt	Defines the source port field cumulative distribution function.
rules_*_*.txt (model, seed)	A set of rules for a particular seed and rule model, where a rule model consists of the set of four CDF files enumerated above.

[1]The cumulative distribution functions were modeled after statistics published by Rovniagin and Wool in the GEM paper [15]. Details on these distributions are provided in Sections 10.1 and 10.2.

A generated rule is stored in five words as shown below:

```
dd07496b : Source IP
b389185b : Destination IP
81b00000 : Bits 31:26 mask length for source IP (32 in this case)
           Bits 25:20 mask length for destination IP (27 case)
           Bits 19:0
0000ffff : Source Port start is the most significant 16 bits and the end is
           the least significant 16 bits.
006a006a : Destination port start is the most significant 16 bits and the
           end is the least significant 16 bits.
```

The software is expected to apply the masks and interpret the necessary fields when building the search tree. For example, once the masks have been applied, the above rule takes the following format:

```
Source IP: (start,end) (0x00000000,0xFFFFFFFF)
Destination IP: (start,end)  (0xB0000000,0xB7FFFFFF)
Source Port: (start,end) (0x0000,0xffff)
Destination Port: (start,end) (0x006a,0x006a)
```

Using this arrangement, arbitrary ranges are allowed in the port field while prefix ranges are allowed in the IP fields. However, it would not be much of a leap to add arbitrary ranges to the address fields.

9.1.2 Build Tree and Search Operations

Once the rule-sets are available, the next step in the verification flow is to build and verify the search data structure. Figure 9-2 depicts the process and flow used to build the search trees and compressed bit-vectors.

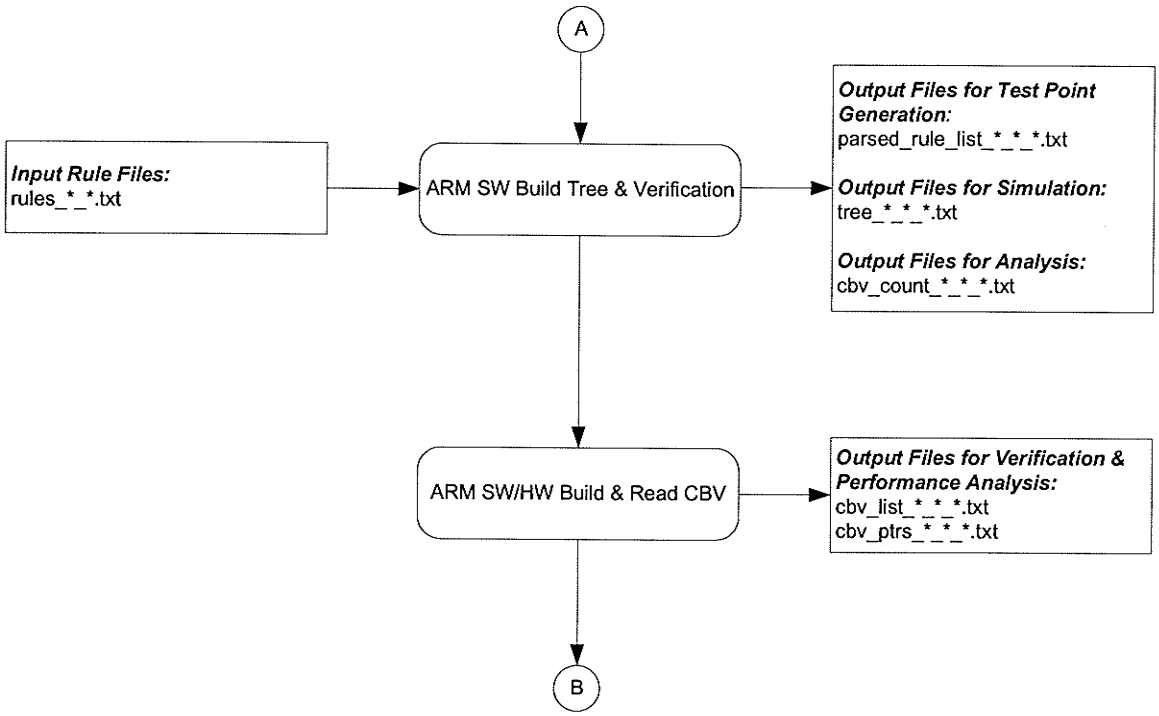


Figure 9-2: System Verification Flow Part 2: Build Tree and CBVs

To build the search trees, the ARM software is first configured for a particular rule size and mode of operation, where the mode indicates which header fields are used to build the tree. Based on this configuration, the software will read in the appropriate rules file, parse out the desired fields and build the trees. Initially, a list of rules is stored at each key in the second dimension B-trees, but in the next stage, these rules are passed to the hardware in exchange for CBV pointers. Table 9-2 details the file I/O associated with this portion of the flow.

Table 9-2: Verification Flow Part 2 Files

File	Function
parsed_rule_list_*_*_*.txt (mode, seed, rule size)	The purpose of this file is to store parsed rules indicating the start and end intervals for each field. This file is later used as a Tcl script input to produce random testpoints. See Appendix A for a sample file. Example: 0x00000000 0x00000000 0xFFFFFFFF 0xB3891800 0xB38918FF

File	Function
tree_*_*_*.txt (mode, seed, rule size)	Contains a text-based representation of the search data structure. It provides the start and end count and list of the rules at each non-overlapping interval in the B-trees. See Appendix A for an example. Note: this file is optionally generated and is used for simulation/tcl validation.
cbv_count_*_*_*.txt: (mode, seed, rule size)	Contains a list of the number of rules encompassed within each CBV. See Appendix A for an example.
cbv_list_*_*_*.txt (mode, seed, rule size)	Contains a list of the CBVs for each non-overlapping interval in the tree. See Appendix A for an example.
cbv_ptrs_*_*_*.txt (mode, seed, rule size)	Contains a list of the RLDRAM pointers for each CBV. See Appendix A for an example.

The next stage of the test plan is performed offline using a Tcl script shell and is illustrated in Figure 9-3. Scripts were developed to generate a set of stimulae and exercise the packet filter. To maximize coverage, multiple testpoints were selected within each rule.

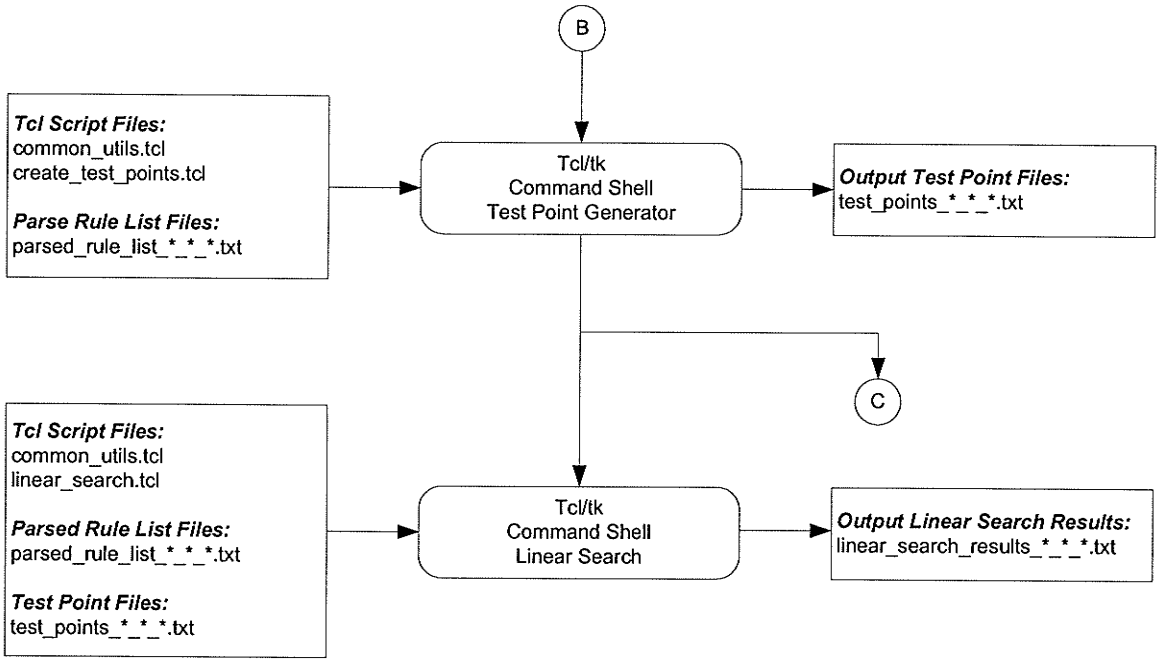


Figure 9-3: System Verification Flow Part 3: Tcl Operations and Linear Search Files

Once the rules and testpoints are available, a basic linear search is performed. The results of this test provide a means to validate the final results from the same testpoints applied to the CBV algorithm implementation. This indirectly validates the entire system including the embedded software and hardware operations. Table 9-3 details the file I/O associated with this portion of the flow.

Table 9-3: Verification Flow Part 3 Files

File	Function
common_utils.tcl	Contains common utilities and file I/O procedures used in Tcl packet filter operations.
create_test_points.tcl	Creates a file of testpoints for each mode, seed and rules size of a given rule model. Random test points were selected in the specified field ranges of each of the rules to ensure that testpoints were created for each rule.
test_points_*_*_*.txt (mode, seed, rule size)	Contains random testpoints for each of the parsed rules in a parsed rule list file (parsed_rule_list). See the Appendix A for an example file.
linear_search.tcl	Performs a linear search of the parsed rules list file for each testpoint and returns a list of matching rules. The list of rules is converted into CBV format and compared against the CBV algorithm solution.
linear_search_results_*_*_*.txt (mode, seed, rule size)	Contains results from linear search operations converted into CBV format.

The CBV algorithm search operations may be performed in unison with the linear search. This portion of the flow is illustrated in Figure 9-4. As B-trees are searched, the software forwards the resulting pointers from each tree to the hardware Packet Filter Assist Acceleration Engine. The PFAAE retrieves the corresponding CBVs, ORs them together and returns the resulting CBV. During this phase, critical information is logged for performance and analysis, including a list of search pointers. Since the ARM7TDMI processor is inherently slow, it was deemed necessary to test the PFAAE OR time and retrieve time independently. To accomplish this task, the software search is bypassed and the logged CBV pointers are burst directly to the PFAAE.

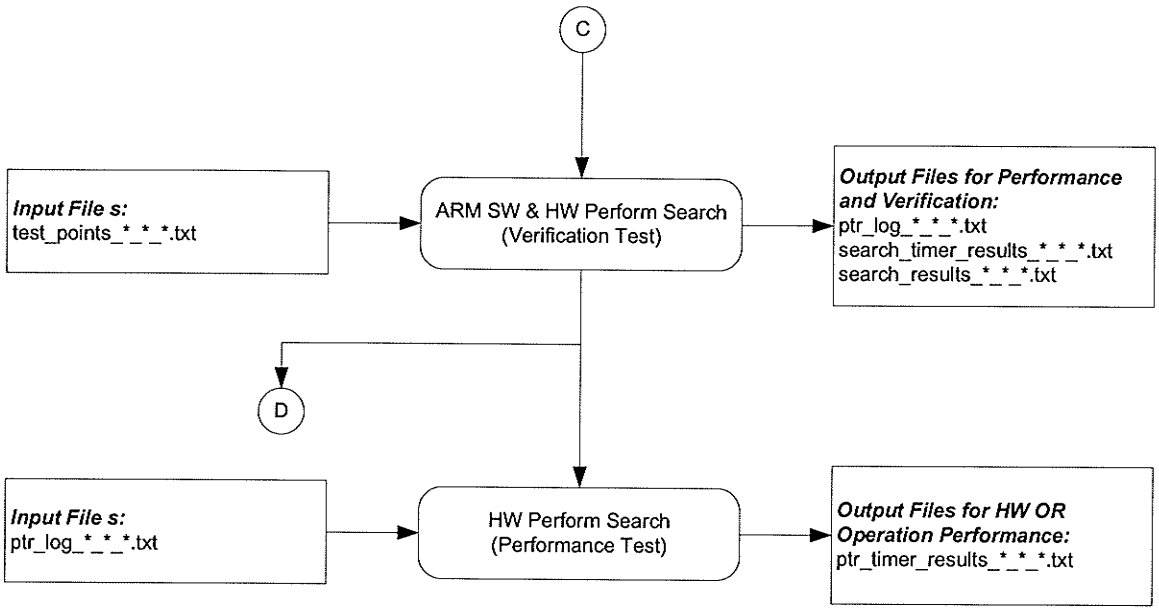


Figure 9-4: System Verification Flow Part 4: ARM Files and Process

Table 9-4 details the additional file I/O associated with this portion of the flow.

Table 9-4: Verification Flow Part 4 Files

File	Function
ptr_log_*_*_*.txt (mode, seed, rule size)	This is a log of the CBV pointers retrieved from the B-trees during each testpoint search. If no pointer is returned from a B-tree then a null pointer 0xFFFFFFFF is returned.
search_timer_results_*_*_*.txt (mode, seed, rule size)	The purpose of this file is to log statistics during search operations. The total time for a search operation is logged along with a count of the number of B-tree nodes that were accessed. Note: the search timer begins when the ARM receives a testpoint and ends when the PFAAE returns the resulting bit-vector. It is returned in 20 MHz clock cycles (50ns period). See Appendix A for an example of this log file.
search_results_*_*_*.txt (mode, seed, rule size)	The search_results file stores the resulting CBVs from two-dimensional searches as well as the time taken for each PFAAE OR operation. The OR time is returned as a count of 50 MHz clock cycles (20ns period). See Appendix A for an example of this file.
ptr_timer_results_*_*_*.txt (mode, seed, rule size)	The purpose of this file is to store the results from a performance test in which pre-calculated B-tree node pointers are read from SRAM and burst to the PFAAE for retrieval and ORing. This test assumes the command queue is always full and measures the time for the PFAAE to complete a filter operation including return communication to the ARM. It is returned in 20 MHz clock cycles (50ns period). See Appendix A for an example of this file.

The next section of the flow illustrated in Figure 9-5 is once again performed offline from the platform. It re-uses the ModelSim environment created to verify the hardware functional behavior in order to validate and debug the platform results. Simple Tcl procedures were developed to translate the tree_*_*_*.txt file into AMBA AHB transactions to build up the entire set of CBVs in RLDRAM. Likewise a simple script was developed to translate the ptr_log_*_*_*.txt into the corresponding AMBA transactions to perform PFAAE CBV OR operations. The results obtained from these operations should match the results obtained on the platform and linear search. Although this type of simulation is limited to small rule-sets, it was found to be invaluable in debugging the system.

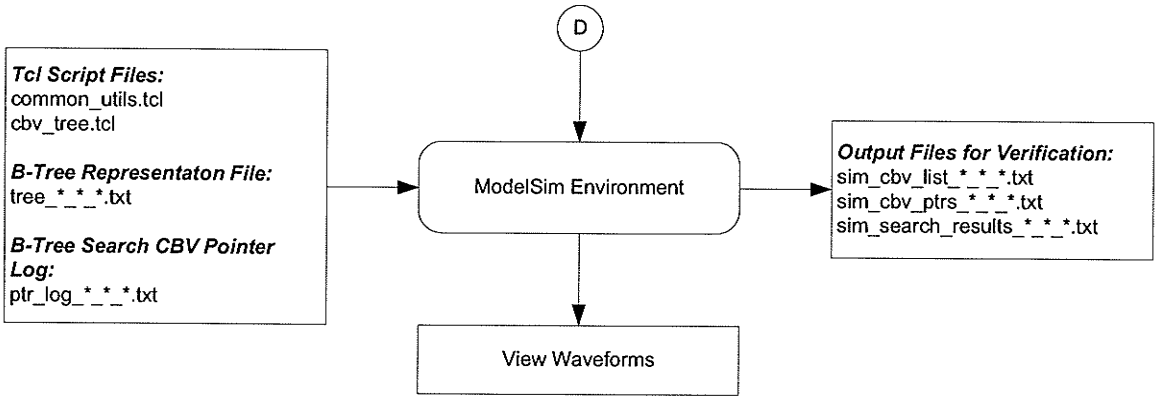


Figure 9-5: System Verification Flow Part 5: Simulation Files and Process

Table 9-5 details the additional file I/O associated with this portion of the flow. For details on the ModelSim environment and configuration files, please refer to Figure 8-1 and Table 8-1 in this document.

Table 9-5: Verification Flow Part 5 Files

File	Function
cbv_tree.tcl	Contains procedures for: • manipulating tree.txt into AHB transfers • building the CBVs and logging CBV pointers and CBV lists • performing OR operations from a list of pointers
sim_cbv_list_*_*_*.txt (mode, seed, rule size)	Contains a list of the CBVs for each non-overlapping interval in the tree. Similar to cbv_list_*_*_*.txt.
sim_cbv_ptrs _*_*_*.txt (mode, seed, rule size)	Contains a list of the RLDRAM pointers for each CBV. Similar to cbv_ptrs_*_*_*.txt.
sim_search_results_*_*_*.txt (mode, seed, rule size)	The sim_search_results file stores the resulting CBVs from two-dimensional searches as well as the time taken for each PFAAE OR operation. The OR time is returned as a count of 50 MHz clock cycles (20ns period). Similar to search_results_*_*_*.txt.

The final section in the system-level verification flow is to compare the results acquired from the platform, simulation and linear searches. This step is shown in Figure 9-6. To accomplish this, a Tcl procedure compares the list of matching rules from the linear search to the rules represented by the compressed bit-vectors returned by the PFAAE in simulation and on the platform.

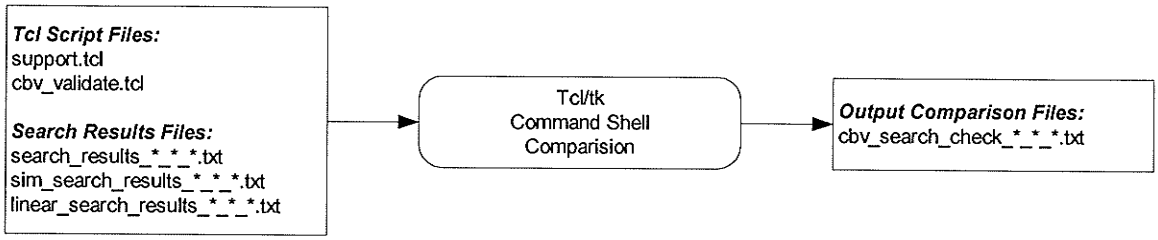


Figure 9-6: System Verification Flow Part 6: Final Verification Files and Process

Table 9-6 details the additional file I/O associated with this portion of the flow.

Table 9-6: Verification Flow Part 6 Files

File	Function
cbv_validate.tcl	Compares the results from the simulation search, linear search and platform search operations.
cbv_search_check_*_*_*.txt (mode, seed, rule size)	Logs the results of the CBV validate operation, flagging any differences between the results.

While this comparison does not guarantee the system is entirely free of bugs, it does indicate a fairly robust design.

Chapter 10: Performance Testing and Analysis

Chapter 10 presents the performance testing of the Compressed Bit-Vector PC algorithm in the rapid prototyping platform. The chapter begins with Section 10.1 presenting detail on real-life firewall database statistics. It is shown that 75% of rules match TCP. Therefore, the performance analysis focuses on the TCP protocol, and the four fields shown in Table 10-1. However, it should be noted that the same discussions could be applied to other protocols (UDP, ICMP) and fields (IP options, TCP flags).

Table 10-1: 4-Dimensional Firewall Match Fields

ID	Field	Space (bits)
0	Source IP	32b
1	Destination IP	32b
2	Source Port	16b
3	Destination Port	16b

The chapter continues in Section 10.2 by outlining a Perimeter Rule Model for generating synthetic structured rule-sets based on common firewall statistics. The results of the Perimeter Rule Model are found in Section 10.3 and demonstrate that the algorithm's space requirements scale far better than the theoretical limits under these conditions. In Section 10.4 the algorithm is tested against a Random Rule Model. While the results of this test indicate fast searching is possible, the algorithm's growth rate tends to approach the theoretical predictions. Finally, the chapter concludes with Section 10.5 making suggestions to improve the CBV algorithm, and an estimation of its performance when implemented in an ASIC.

10.1 Rule-Set Statistics

To gain an understanding of real-life firewall rule-set characteristics, this thesis relied on information and statistics gathered from literature. In the GEM paper [15] Rovniagin and Wool perform an examination of statistics from 19 corporate firewall rule-sets (CISCO PIX and Check Point Firewall-1) across a variety of industry sectors. The analysis revealed a large degree of structure within the rule-sets as shown in Table 10-2. First off, the distribution of protocols was heavily weighted towards TCP. In fact, 75% of the rules matched TCP and 93% matched one of TCP, UDP or ICMP. The analysis also showed that source ports are almost entirely wildcards,

while the destination ports are usually specified exactly. These findings make complete sense for the TCP protocol. For stateful packet filters, source port filtering is not required for return traffic through the firewall. It is also untrustworthy because it can easily be under control by hackers. For the destination port 96% of rules were found to contain single ports spread amongst nearly 200 numbers, while 4% were ranges. The destination port distribution in Table 10-2 also shows the average range size was quite large. Common ranges included all high ports 1024-65535 or X11 ports 6000-6003 [15].

Table 10-2: Protocol and Port Number Distributions in Rule-Sets [15]

Source Port Distribution	
*	98%
ranges	1%
single port	1%

Protocol Distribution	
*	6%
TCP	75%
UDP	14%
ICMP	4%
Other	1%

Destination Port Distribution		
*	0%	
ranges	4%	
average range size	27030	
single ports	96%	
avg number of single ports per rule-base	50	
most used ports	80	6.89%
	21	5.65%
	23	4.87%
	443	3.90%
	8080	2.25%

Due to the large amount of structure observed within real firewall rule-sets, it is probable that trials on these rule-sets will not result in worst-case behavior of the CBV algorithm. To verify this theory, the algorithm's space and search time behavior are quantified against rule-sets generated with distributions from a Perimeter Rule Model developed by GEM and described in Section 10.2 [15].

10.2 Perimeter Rule Model

The Perimeter Rule Model assumes a firewall on the network edge similar to the setup shown in Figure 1-1 in Section 1.3. On the inside is a protected network and on the outside is the Internet. Outbound rules govern traffic flow from the protected network out to the Internet, while inbound rules govern traffic from the Internet into the protected network. The internal network is made up of 10 Class B networks, and the Internet consists of all other IP addresses. Although, it is uncommon for a company to own 10 Class B networks, a larger internal subnet will further

stress the algorithm. This is because the rule generator assigns random ranges from internal ranges. Rovniagin and Wool also suggest this is a reasonable approach because:

"Many organizations use private (RFC 1918) IP addresses internally, and export them via network address translation (NAT) on outbound traffic. Such organizations often use large subnets liberally. e.g. assign a 172.x.*.* class B subnet to each department." [15]

10.2.1 Rule and Traffic Generation

The rule generator created during hardware and software integration testing was used to produce the inbound and outbound rule-sets from files containing cumulative distributions. The distributions were developed to closely match those provided in the GEM paper and re-stated in Table 10-3 [15]. In some areas where Rovniagin and Wool's model is ambiguous or specific details are missing, reasonable assumptions were made to complete the missing pieces.

Table 10-3: Statistical Distribution for the Perimeter Rule Model Rule-Sets [15]

Field	Match Type	Inbound	Outbound
source address	*	95%	5%
	range	5%	15%[1]
	Class B		10%
	Class C		25%
	single IP		45%
destination address	*		90%
	range	15%	5%
	Class B	10%	
	Class C	30%	
	single IP	45%	5%
destination port (Service)	from list of 100 services	96%	96%
	dst port is random range	2%	2%
	dst port is single port	2%	2%
source port	*	98%	98%
	src port is random range	1%	1%
	src port is from used port list	0.5%	0.5%
	src port is random 0-65535	0.5%	0.5%

[1]Range completely lies inside one of the internal class C networks.

The remainder of Section 10.2.1 provides further detail on inbound and outbound rule generation for each of the fields in Table 10-1.

10.2.1.1 Inbound Source IP Address

When generating the *source IP* (SIP) address field for rules applied to inbound traffic, Table 10-3 shows that the IP address is rarely specified. Up to 95% of the rules have a wildcard as their source address and the remaining 5% have a range. For the non-wildcard rules, the rule generator was designed to assign the SIP field one octet at a time followed by a prefix length. The SIP address was chosen at random from a discrete uniform distribution of all the Internet's IP addresses minus the internal network range 179.127.0.0-179.136.255.255 and specific addresses reserved by the *Internet Assigned Numbers Authority* (IANA) for private internets such as 10.0.0.0-10.255.255.255. The associated prefix mask length was then selected based on the distribution shown in Figure 10-1.

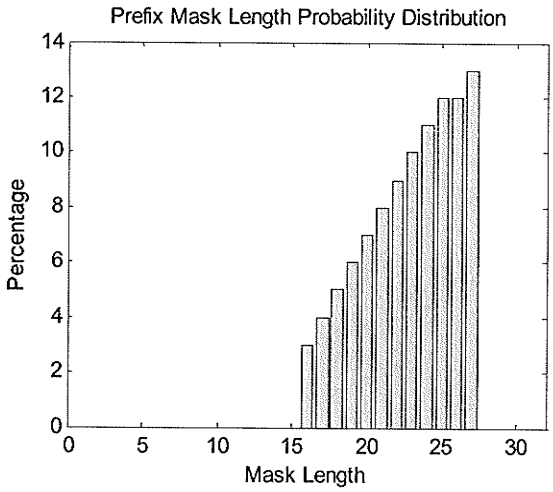


Figure 10-1: Inbound Source IP Prefix Mask Length Probability Distribution

This distribution shows increasing probability from Class B to slightly smaller than Class C. It is noted that this is a deviation from the uniform distribution of ranges applied in the GEM paper. However, it was felt this distribution is more representative of the actual Internet due to the introduction of *Classless Inter-Domain Routing* (CIDR).

10.2.1.2 Inbound Destination IP Address

When generating the *destination IP* (DIP) address field for inbound traffic rules, the address will always belong to one of the 10 internal Class B subnets defined in the Perimeter Rule Model. For a single Class B network, the first two octets define the network ID while the third and fourth octets define the host ID as shown in Table 10-4.

Table 10-4: Class B Network and Host ID's

	Network ID		Host ID	
Octet	First	Second	Third	Fourth
binary	10110011	10000000	00000000	10001111
decimal	179	128	0	143

Similar to the SIP field, the rule generator was designed to assign the non-wildcard DIP addresses one octet at a time followed by a prefix length. The following criteria were followed when assigning each of the octets:

- For a valid Class B network the first octet is in the range [128:191] inclusive. For test purposes, 179 was selected.
- The second octet was randomly selected from a discrete uniform distribution of 10 consecutive pre-selected Class B subnets: [128:137].
- The third and fourth octets defined the host ID and were randomly selected based on discrete uniform distributions.

Once the DIP address was generated, a prefix mask length was attached. For single DIPs, a prefix length of 32 was automatically attached. For each of the range-based rules, a prefix mask length was selected based on the distribution shown in Figure 10-2.

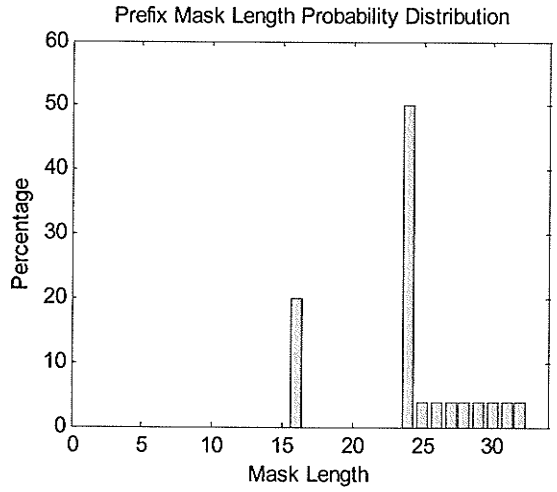


Figure 10-2: Inbound Destination IP Prefix Mask Length Probability Distribution

The above distribution shows large spikes at mask lengths of 16 and 24 as well as a uniform distribution from 25 to 32. This is intended to reflect reality, where the first two spikes represent full Class B and Class C masks respectively, and the uniform distribution allows for ranges smaller than a Class C network.

10.2.1.3 Inbound/Outbound Source Port

The *source port* (SP) field was selected similarly for both inbound and outbound rules. From the analysis in Table 10-3, it is apparent that a single port number is rarely specified, with 98% of rules using a wildcard in this field. This is consistent with stateful firewalls that do not need to perform source-port filtering to allow return traffic through the firewall. In addition, the SP field is generally unreliable since it is can easily be under the control of an attacker [15]. In generating this field it was necessary to make some assumptions regarding the non-wildcard rules:

- 1% of rules contained a range in the source port field. The range was selected according to Table 10-5 providing an average range size of:

$$60000 \times 0.45 + 10000 \times 0.05 + \frac{(3+4+\cdots+30)}{28} \times 0.45 + \frac{(100+101\cdots+1000)}{901} \times 0.05 \approx 27535$$

- 0.5% of the rules contained a SP field from a list of 100 commonly used ports with the weighting detailed in Table 10-6.
- 0.5% of the rules contained a random single port number selected from 0:65535. As the number of rules increase, this allows for a small rate of growth in the number of services.

Table 10-5: Source Port Range Probabilities

Range	Probability	Description
3-30	45%	Allows small ranges between 3 and 30 in size.
100-1000	5%	Allows any range between 100 and 1000.
10000	5%	Allows range of size 10000.
60000	45%	Allows range of 60000 only.
Total	100%	

Table 10-6: Most Frequently used Ports

Most Used Ports	Service	Probability
80	HTTP	7.08%
21	FTP	5.65%
23	TELNET	4.87%
443	SSL (Secure Socket Layer)	3.9%
8080	HTTP	2.5%
list of 95 individual ports 100-194		0.8% per port
Total		100%

10.2.1.4 Inbound/Outbound Destination Port

Contrary to the large degree of wildcards in the source port field, the *destination port* (DP) field is usually precisely specified for TCP. The majority of rules specified a single destination port number, while only a small portion allowed a range of ports. In general, the ranges were found to be quite large, with common ranges being all high ports (1024–65535) or X11 ports (6000-6003). The following list contains the criteria employed by the rule generator:

- 96% of the rules contained a single DP number distributed amongst 100 port numbers including the most popular services. The most common are shown in Table 10-6.
- 2% of rules contained a range in the DP field. The ranges were selected identically to the source port field. Again, most of the weighting was placed in extremely low and extremely high ranges.
- 2% contained a random single DP number, where the port was randomly selected from 0:65535. As the number of rules increase, this allows for a small rate of growth in the number of services.

10.2.1.5 Outbound Source IP Address

The SIP addresses for outbound rules are selected from the internal addresses with the frequencies shown in Table 10-3. For the non-wildcard rules, the rule generator assigned IP addresses one octet at a time followed by a mask length. The following criteria were followed when assigning each of the octets:

- For a valid Class B network, the first octet is in the range [128:191] inclusive. For test purposes, 179 was selected.
- The second octet was randomly selected from a discrete uniform distribution of 10 consecutive pre-selected Class B subnets: [128:137].
- The third and fourth octets defined the host ID and were randomly selected based on discrete uniform distributions.

Once the SIP address was generated, a prefix mask length was attached. For single IPs the mask length is 32 and for range-based rules a prefix was selected from a distribution similar to that shown in Figure 10-2.

10.2.1.6 Outbound Destination IP Address

When modeling the DIP address field for rules applied to outbound traffic, the address was found to be a wildcard 90% of the time. The remaining 10% was split between single addresses and ranges. This produces rules modeling the access policies to specific servers or networks. For the non-wildcard rules, the rule generator assigned the DIP field one octet at a time followed by a

prefix length. The DIP address was chosen at random from a discrete uniform distribution of the entire IP address space minus the internal network and specific addresses reserved by the IANA. For single IP addresses, a 32-bit mask was added and for ranges, a mask was selected from a distribution similar to the one shown in Figure 10-1.

10.2.1.7 Traffic Generation

To test the efficiency of the algorithm, inbound and outbound traffic needs to be passed through the packet classifier and the resulting performance logged. To accomplish this, packet header information was generated for TCP traffic (source IP address, destination IP address, source port number and destination port number). It is only necessary to generate the header fields since the search algorithm would ignore the rest of the packet. The headers were created by randomly picking testpoints within each rule to maximize coverage and ensure that inbound packets will only contain destination addresses behind the firewall.

10.3 Perimeter Rule Model Results

In the CBV algorithm, there are four fields, but 2-dimensional search operations are performed in parallel and the results combined. Therefore, this provides the potential of twelve different 2-D pairs as outlined in Table 10-7. The identifiers for these pairs are used in the results plots in the remainder of this document.

Table 10-7: 2-Dimensional Field Combinations

File Numbering	2-D Pair Identifier	Field 1 (Dimension 1)	Field 2 (Dimension 2)
1	P01	Source IP	Destination IP
2	P02	Source IP	Source Port
3	P03	Source IP	Destination Port
4	P10	Destination IP	Source IP
5	P12	Destination IP	Source Port
6	P13	Destination IP	Destination Port
7	P20	Source Port	Source IP
8	P21	Source Port	Destination IP
9	P23	Source Port	Destination Port
10	P30	Destination Port	Source IP
11	P31	Destination Port	Destination IP
12	P32	Destination Port	Source Port

To obtain insight into the 4-dimensional growth rate and search performance, various combinations of the two-dimensional searches were combined. The growth rate in four dimensions becomes the addition of the space requirements while the hardware search time is limited by the worst-case search time of the two 2-D pairs. Since the searches are performed in parallel, the order of the pair is inconsequential (e.g. P01_P23 is the same as P23_P01). Thus, there are twelve valid groupings of two-dimensional pairs to consider as outlined in Table 10-8.

Table 10-8: 4-Dimensional Field Combinations

File Numbering	4D Pair Identifier	Pair1	Pair 2
1,9	P01_P23	Source IP, Destination IP	Source Port, Destination Port
1,12	P01_P32	Source IP, Destination IP	Destination Port, Source Port
2,6	P02_P13	Source IP, Source Port	Destination IP, Destination Port
2,11	P02_P31	Source IP, Source Port	Destination Port, Destination IP
3,5	P03_P12	Source IP, Destination Port	Destination IP, Source Port
3,8	P03_P21	Source IP, Destination Port	Source Port, Destination IP
4,9	P10_P23	Destination IP, Source IP	Source Port, Destination Port
4,12	P10_P32	Destination IP, Source IP	Destination Port, Source Port
5,10	P12_P30	Destination IP, Source Port	Destination Port, Source IP
6,7	P13_P20	Destination IP, Destination Port	Source Port, Source IP
7,11	P20_P31	Source Port, Source IP	Destination Port, Destination IP
8,10	P21_P30	Source Port, Destination IP	Destination Port, Source IP

It should also be noted that ten rule-sets were generated for each set of distributions described in Section 10.2.1, where each rule-set was initialized from a different random seed. Thus, every point in the result plots reflects the mean value from 10 independent runs, providing a 90 % confidence interval. If a confidence interval is not visible, then it is because the interval was too small to show.

10.3.1 Best Field Order Analysis

During development and initial testing, it became apparent that the field order greatly impacted the size of the data structure (CBV and B-tree memory usage) produced with the Perimeter Rule Model. It became an objective to compare the performance and data structure size of different field orders. It is fair to consider only the 2-D pairs in this analysis since the four-dimensional search is composed of two 2-D searches run in parallel and the worst two-dimensional pairs will limit its space and search performance.

During this examination, the differences in data structure size became so great, the memory space required to build the tree became prohibitive and large rule-set trials could not be performed on the worst choices. Therefore, in an effort to weed out the worst field orders and identify the best ones, an iterative elimination process was applied. The main rationale for dropping pairs was data structure size, but each pair was also exercised with traffic and search performance logged. The hardware search time (retrieval and ORing time) provides a second criterion for dropping pairs. It should also be noted that pairs were dropped when no 4-D groupings were left for analysis.

In the first step, small rule-sets (1024 rules) were generated and the data structure was built for each of the 12 field orders. The following bar graphs show the results from this stage with a 90% confidence interval for outbound rules. Figure 10-3 (A) indicates the data structure size for each pair while Figure 10-3 (B) illustrates the average search time when traffic was passed. The line graphs in Figure 10-3 (B) provide further insight by indicating the maximum and minimum hardware search times.

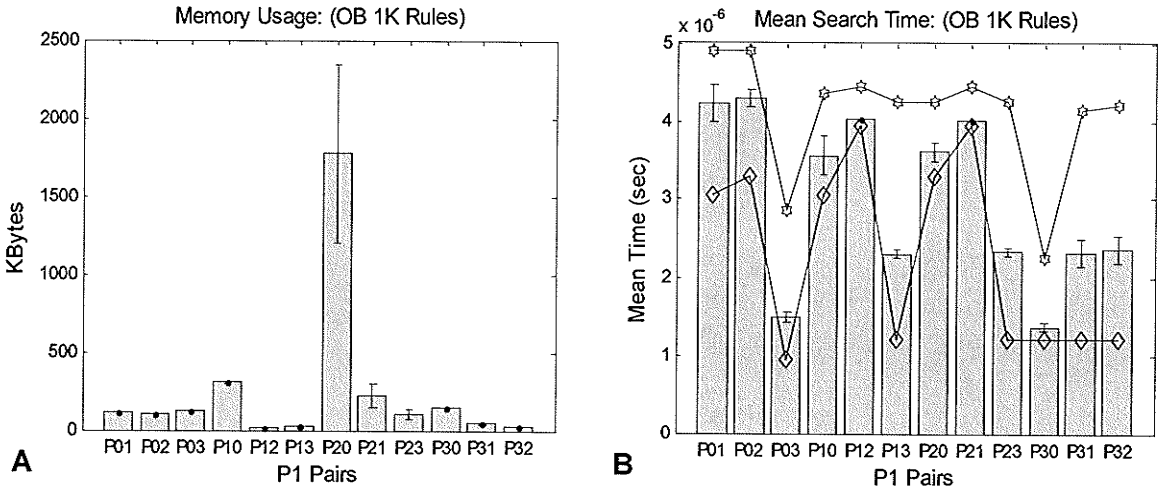


Figure 10-3: Finding the Best 2-D Outbound Field Orders (1K rules)

The results indicate pairs P20, P10, and P21 were the worst by size while P02, P01, P12 and P21 were the worst by speed. Therefore, pairs P20, P21, and P10 were dropped and the elimination process continued on rule-sets of size 2K with the remaining pairs. Figure 10-4 shows the results from the second stage.

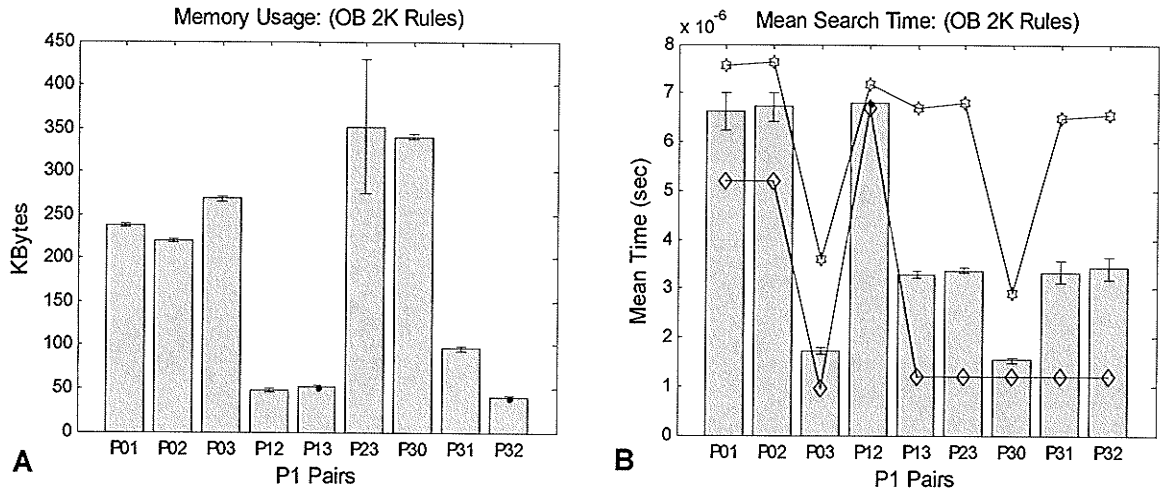


Figure 10-4: Finding the Best 2-D Outbound Field Orders (2K rules)

The results in Figure 10-4 indicate pairs P23, P30 and P03 were the worst by size while P02, P12 and P01 were the worst by speed. Pairs P23, P30 and P03 were dropped and the elimination process was continued on rule-sets of size 4K with the remaining pairs. The P12 pair was also removed because it no longer had any partnering pairs (P03 or P30). Figure 10-5 shows the results from the third stage.

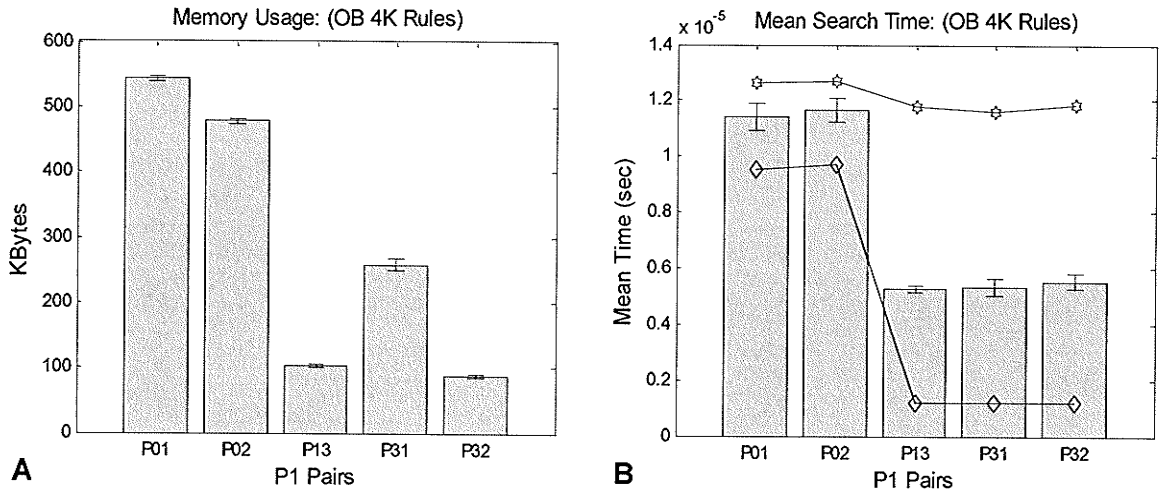


Figure 10-5: Finding the Best 2-D Outbound Field Orders (4K rules)

The results in Figure 10-5 indicate pairs P01 and P02 were the worst by size while P01 and P02 were the worst by speed. However, no pairs were dropped since these pairs are required to estimate the performance of a 4-D implementation. Together, the remaining pairs (P01, P02, P13, P31 and P32) combine for three P2 groupings P01_P32, P02_P13 and P02_P31. From these

combinations the natural pairing is likely P01_P32 (i.e. <source IP, destination IP>, <destination port, source port>). However, it appears the most efficient implementation may be P02_P13 (i.e. <source IP, source port>, <destination IP, destination ports>).

With the outbound field order analysis complete, the iterative procedure was repeated for the inbound case. Figure 10-6 shows the results from the first stage.

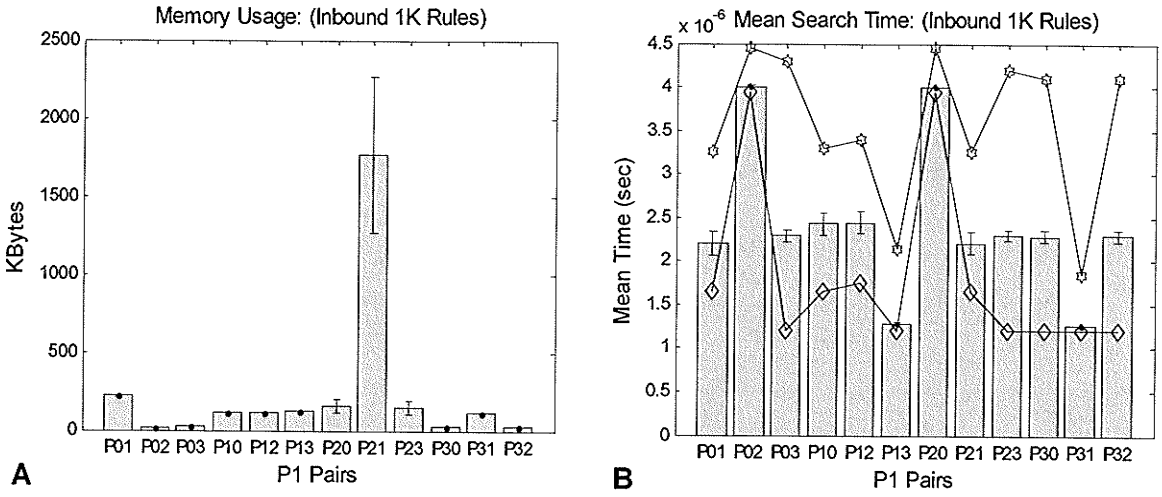


Figure 10-6: Finding the Best 2-D Inbound Field Orders (1K rules)

The results in Figure 10-6 indicate pairs P21, P20 and P01 were the worst by size while P02, P20 and P23 were the worst by speed. Pairs P21 and P20 were dropped and the iteration process continued on rule-sets of size 2K with the remaining pairs. Figure 10-7 shows the results from the second stage.

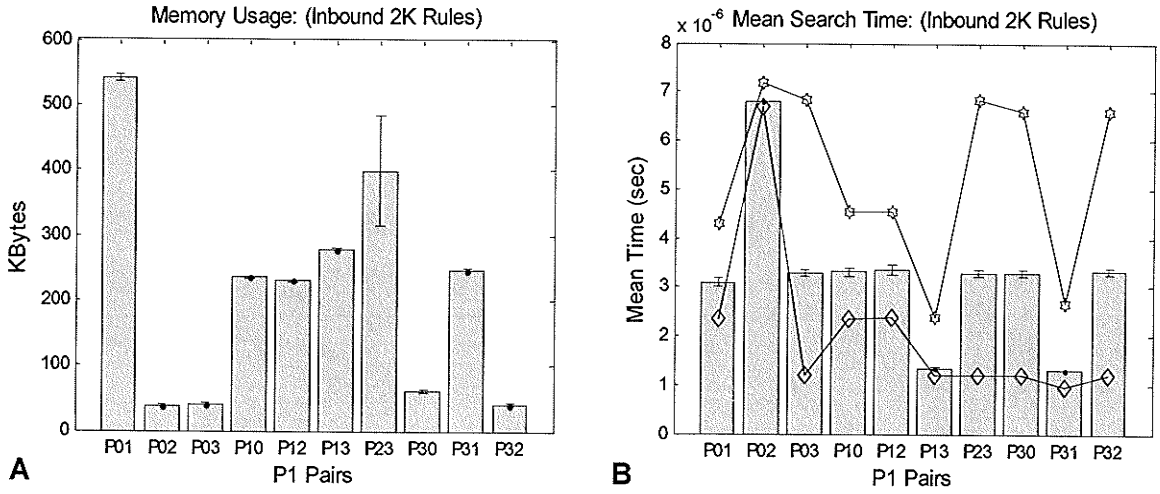


Figure 10-7: Finding the Best 2-D Inbound Field Orders (2K rules)

The results in Figure 10-7 indicate that pairs P01, P23 and P13 were the worst by size while P02 was the worst by speed. Pairs P01, P23 and P13 were dropped and the iteration process continued on rule-sets of size 4K with the remaining pairs. Figure 10-8 shows the results from the second stage.

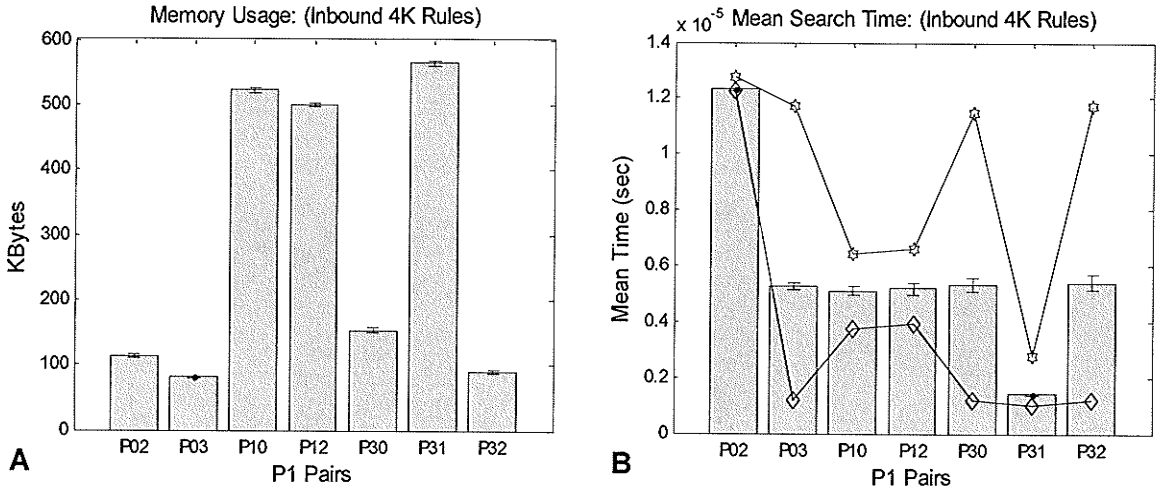


Figure 10-8: Finding the Best 2-D Inbound Field Orders (4K rules)

The results in Figure 10-8 indicate pairs P31, P10 and P21 were the worst by size while P02 was the worst by speed. Pair P31 was dropped due to size performance and P02 dropped since it no longer had any partnering pairs for 4-D grouping analysis. This leaves five P1 pairs P03, P10, P12, P30 and P32, which can form up to three P2 pairs P10_P32, P03_P12 and P30_P12 for investigation.

10.3.2 Growth Rate

After establishing the best field orders, rule-sets as large as 16K were investigated to provide a better understanding of the CBV algorithm's performance and properties. First off, to gain more insight into the memory savings achieved by using compressed bit-vectors, the compression ratios were calculated for each of the best P1 pairs, as shown in Figure 10-9. The compression ratios reveal that the use of CBVs resulted in significant memory savings particularly for pairs where the primary search field was an IP address.

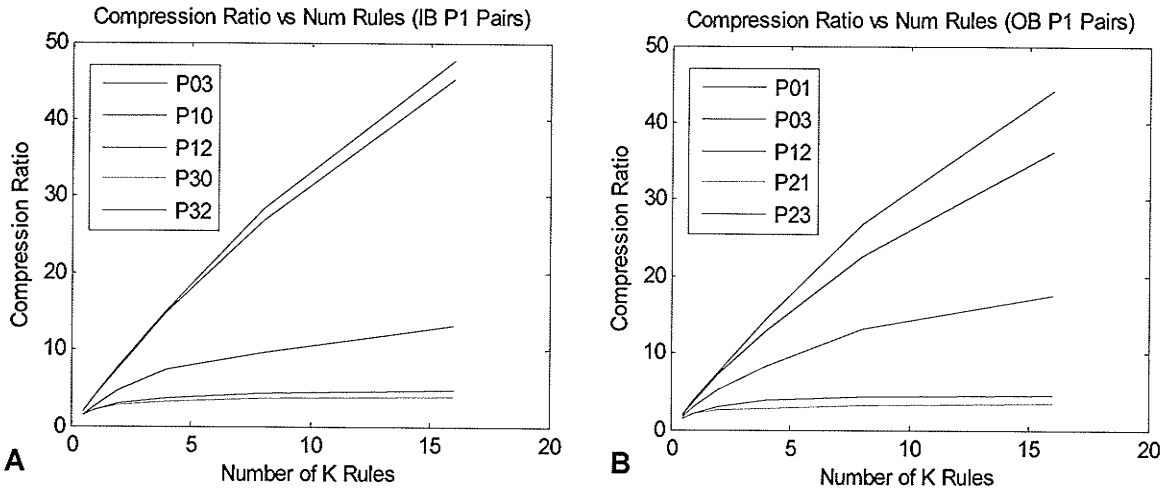


Figure 10-9: P1 Pairs Compression Ratio vs. Number of Rules: (A) Inbound (B) Outbound

Next, the final outbound and inbound CBV data structure sizes were plotted in Figure 10-10 and Figure 10-11 for the best field orders as a function of the rule-set size. In this case, the size is a combination of the CBVs and B-tree memory usage. However, these results do not reflect the peak memory usage during the build tree operation. Very little optimization was done during the build operation, thus the required memory usage could exceed 256 MB. This constraint in combination with the processor and bus speeds were the major limiting factors in testing larger rule-sets.

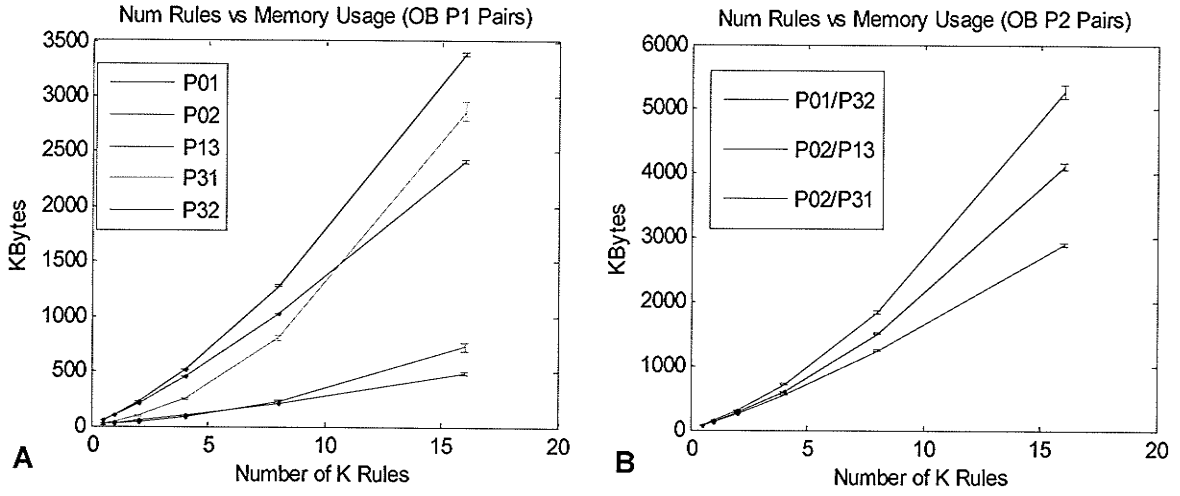


Figure 10-10: Outbound Size vs. Number of Rules: (A) P1 Pairs (B) P2 Pairs

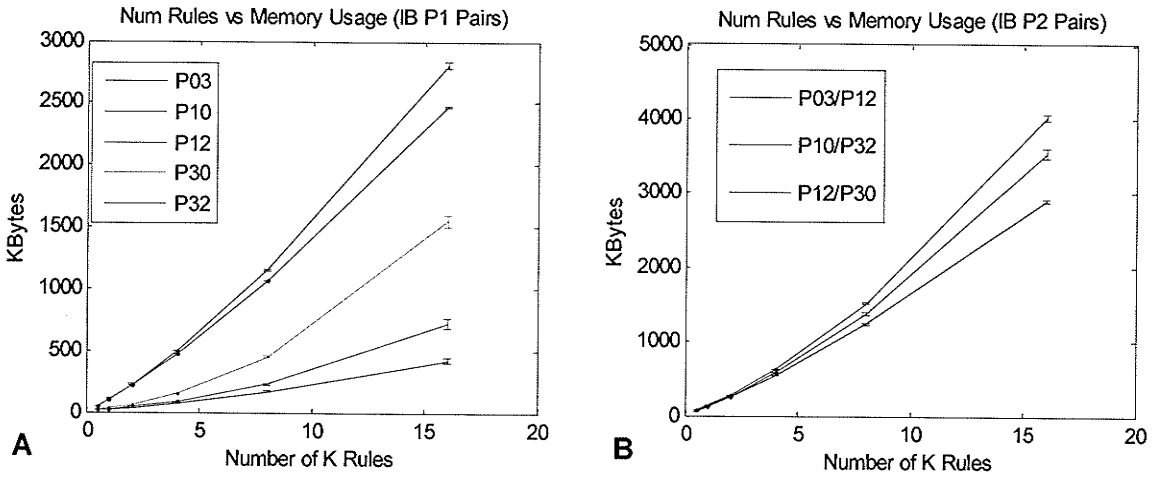


Figure 10-11: Inbound Size vs. Number of Rules: (A) P1 Pairs (B) P2 Pairs

From the above figures it is clear that the data structure size is quite small even for rule-sets as large as 16K. For the outbound case, the P1 pairs consumed less than 3.5 MB and the resulting P2 pairs less than 6.0 MB. As expected, the best outbound combination in terms of space was P02_P13. For the inbound case, the individual P1 pairs required less that 3.0 MB of memory while the resulting P2 pairs were within 5.0 MB. Overall, the best inbound and outbound combinations were P03_P12 and P02_P13 respectively.

By plotting the best P2 pairs on a log-log plot, one can deduce if the relationship between the two variables follows the power law behavior in Equation 10-1.

Equation 10-1: Power Law Behavior

$$y = 10^b x^a$$

When $y = 10^b x^a$, a plot of $\log_{10} y = a \log_{10} x + b$ forms a straight line with slope equal to a and an intercept equal to b. In this case a represents the observed growth rate for rule-sets based on the Perimeter Rule Model. Figure 10-12 shows the outcome of plotting the outbound combination P02_P13 on a log-log scale. A fitted line and plot of $y = 10^b x^2$ are also shown.

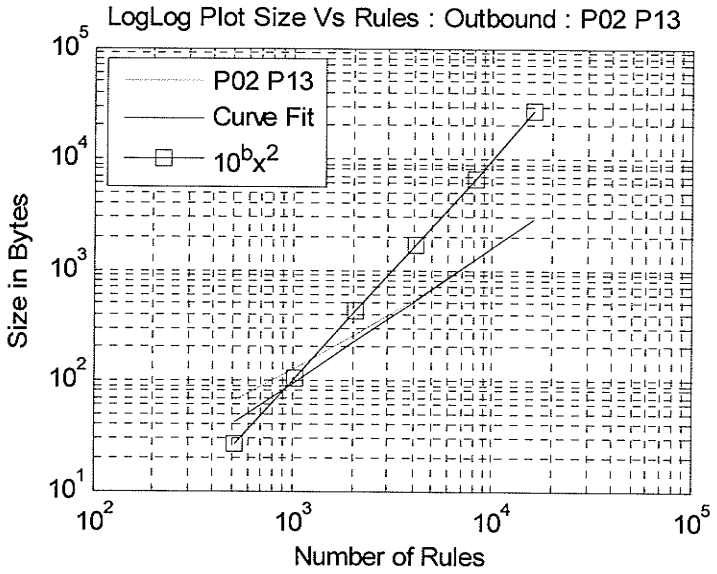


Figure 10-12: Log-Log Growth Plot (Outbound P02_P13)

From the log-log plot it is clear that the set of points form a nearly straight line. Thus, by performing a curve fit, the slope and intercept can be found within experimental error. Using a Matlab polyfit function on the steepest part of the curve (last 2 data points), the slope and intercept are found as follows:

```
my_fit = polyfit(log10(x),log10(y),1) (1 means a linear fit)
a = my_fit(1) = 1.2276 (slope)
b = my_fit(2) = -1.7117 (intercept)
```

$$y = 10^{-1.7117} x^{1.2276}$$

Therefore, the slope indicates that the data structure size is growing at a rate of slightly less than $5/4$ with respect to the rule-set size. This is much slower than the theoretical upper bound of $O(n^3)$.

10.3.3 Search Time and Throughput

In this section, the search time for the CBV algorithm is evaluated, based on the traffic generation outlined in Section 10.2.1.7. Early on it was discovered that the major bottleneck in the packet classifier's throughput was the speed of the ARM processor. Therefore, this analysis is split into software operations and hardware operations.

10.3.3.1 Software Operations Performance

Since the software performance is severely limited by the processor speed, it did not make sense to analyze its performance based on search time measurements. Instead, software analysis focuses on the number of node accesses required for a search operation. Based on the number of node accesses and node size, it is then possible to predict the number of memory accesses. In practice, this is a good technique for measuring performance since hardware and software architectures are limited by memory bandwidth.

Generally speaking, it would be a good idea for the node size to fall within a single cache line so that one node access maps to one memory access. Figure 10-13 illustrates the total average number of nodes accessed for inbound and outbound P1 pairs versus the number of rules.

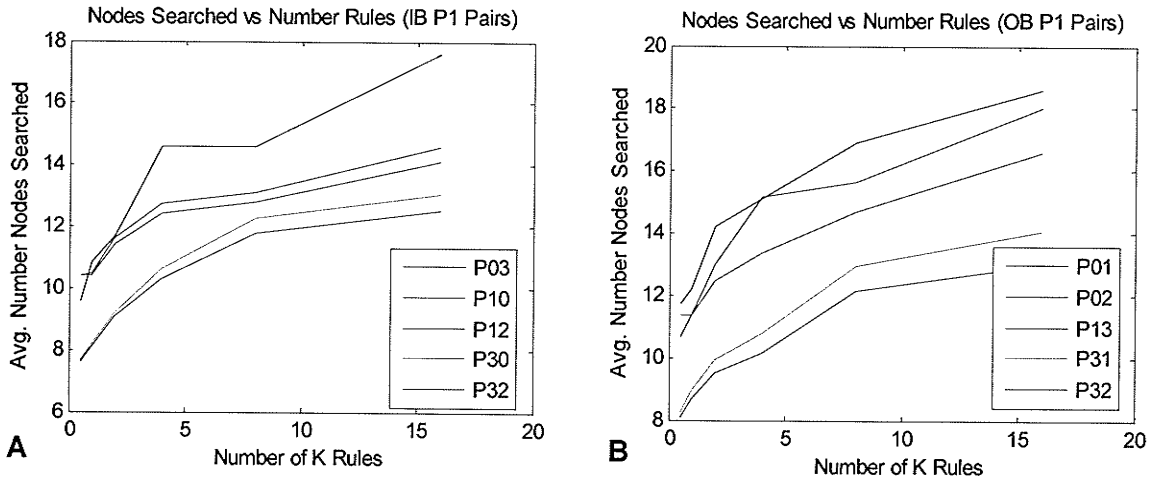


Figure 10-13: P1 Pairs Number of Nodes vs. Number of Rules: (A) Inbound (B) Outbound

In addition to the above figures the distributions for B-tree node accesses are included in Appendix B. These plots are provided as additional information with no analysis provided.

10.3.3.2 Hardware Operations Performance

For the hardware operations, the performance is measured using the actual time required to retrieve the CBVs and perform the OR operation. The final AND operation between the two P1 search results is ignored since it would be pipelined with the OR. It is expected that the CBV AND function would also take far less time then the OR operation since it could be terminated on first match.

Given that the software could not keep pace with the hardware, the search and data transfer between the ARM processor and the PFAAE had to be eliminated from this test. To accomplish this, the CBV search pointers were logged during the software speed analysis. The logged pointers were stored in internal memory and burst to the PFAAE. Then, by utilizing performance-measuring logic built into the PFAAE and logic tile, the OR and overall hardware times were recorded for each search operation. Figure 10-14 and Figure 10-15 illustrate the total time spent performing hardware operations for inbound and outbound packet classification respectively. In each figure, the P1 pair performance is shown on a linear plot while the P2 performance is provided on a log-log plot. In each case, the results show a linear relationship between search time and the number of rules.

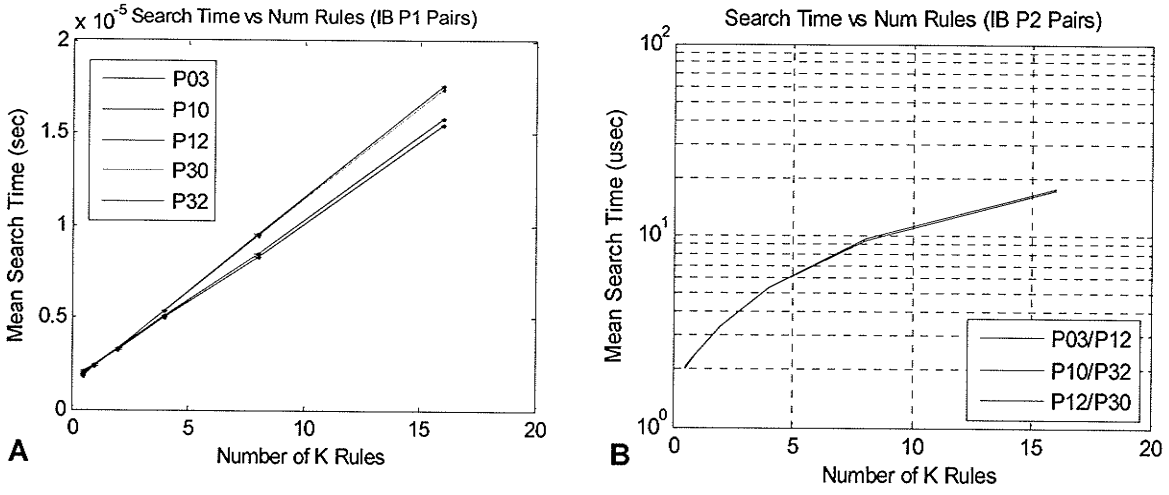


Figure 10-14: Inbound Search Time vs. Number of Rules: (A) P1 Pairs (B) P2 Pairs

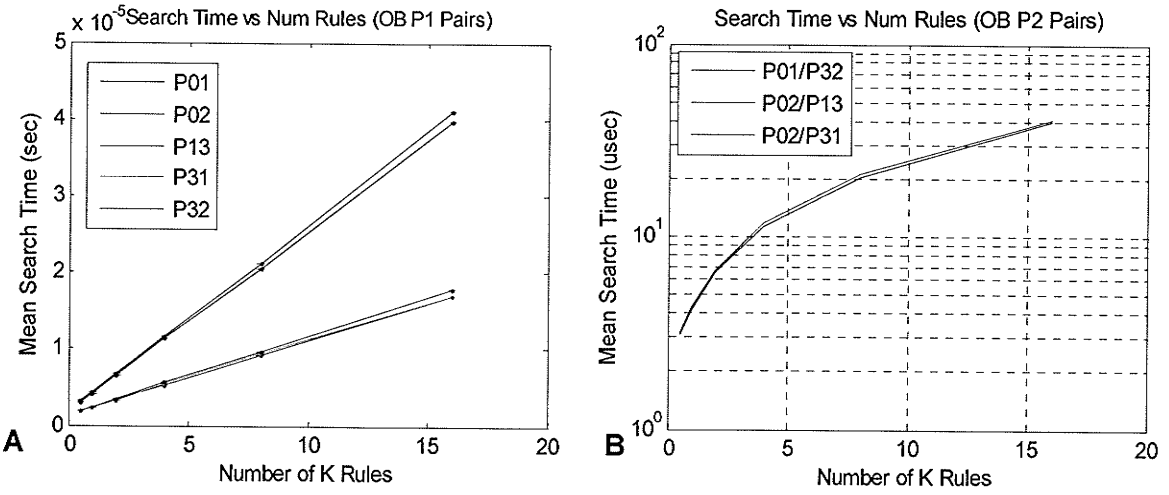


Figure 10-15: Outbound Search Time vs. Number of Rules: (A) P1 Pairs (B) P2 Pairs

Nonetheless, for inbound traffic, the implementation of the CBV algorithm was able to sustain a throughput of approximately 18 us per packet or 56,000 pps. For outbound traffic, it was able to sustain a throughput of approximately 42 us per packet or 24,000 pps. This is far from a GE target, but the platform is also running at reduced speeds compared with an ASIC implementation. In Section 10.5, a case study is performed to illustrate how the algorithm could sustain a Gigabit Ethernet line-rate.

It is hypothesized that the linear growth in Figure 10-14 and Figure 10-15 is attributed to the characteristics of the rule-sets from the Perimeter Rule Model. Both the initial and resultant CBVs were much denser than originally expected as shown in Figure 10-16.

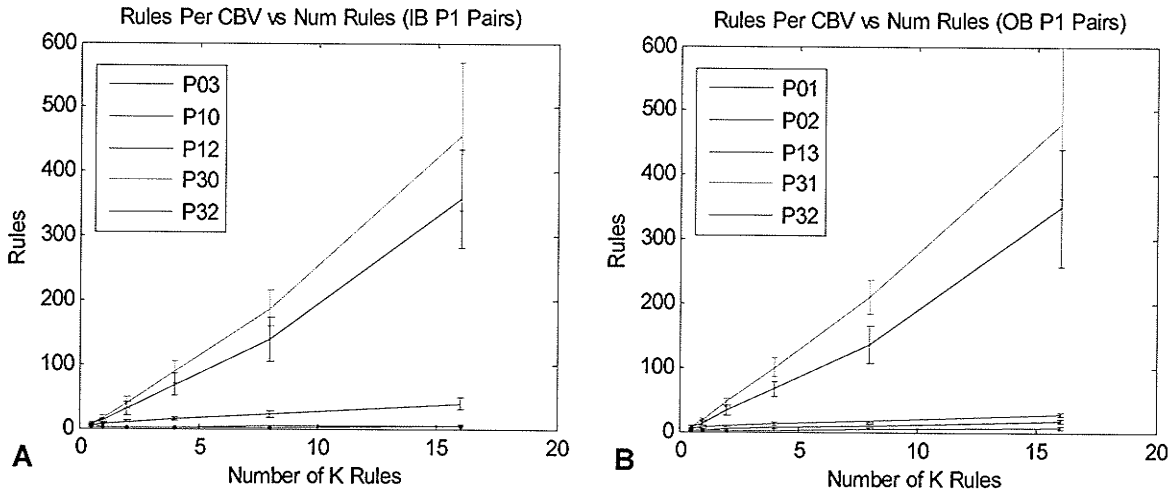


Figure 10-16: Rules Stored per CBV vs. Number of Rules: (A) Inbound P1 Pairs (B) Outbound P1 Pairs

This is in high contrast to the ABV paper whose experimental rule-sets, based on industrial firewalls contained at most four matching rules per packet [2]. It is left for future work to perform trials based on additional rule-set models.

To obtain a deeper understanding of where the bottlenecks are in the hardware portion of the algorithm, the time spent strictly retrieving CBVs was also captured. Figure 10-17 and Figure 10-18 illustrate the amount of time spent performing CBV retrieval from the RLDRAM to the internal FPGA memory for the inbound and outbound searches respectively.

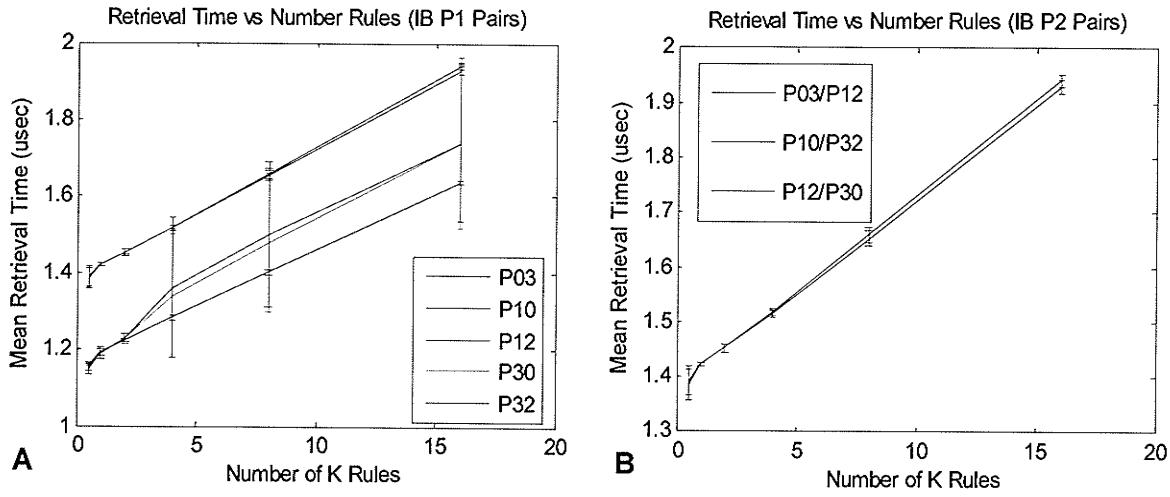


Figure 10-17: Inbound Retrieval Time vs. Number of Rules: (A) P1 Pairs (B) P2 Pairs

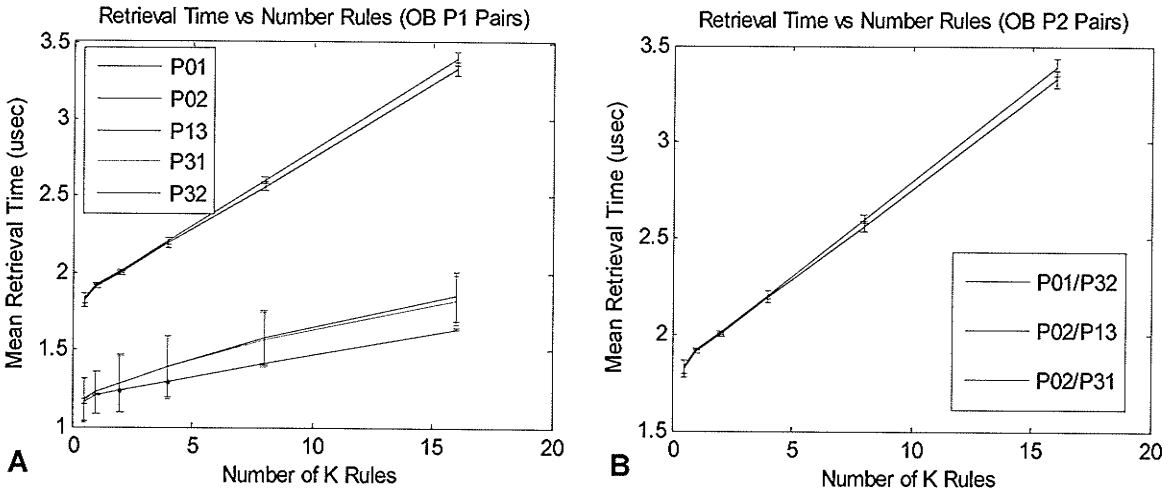


Figure 10-18: Outbound Retrieval Time vs. Number of Rules: (A) P1 Pairs (B) P2 Pairs

Clearly the CBV retrieval time is not the bottleneck in the PFAAE throughput. For 16K rules, the average time spent retrieving CBVs based on rule-sets from the Perimeter Rule Model was less than 2 us for inbound and less than 3.5 us for outbound classification.

One distinct observation from the retrieval plots is the large confidence intervals on P32 and P30 for inbound and P31 and P32 for outbound. In all cases, this occurs when the destination port is the primary search field. The large confidence interval is an artifact of the probability distribution used to generate the destination port field as well as the criteria for splitting the rule-sets into buckets. The 16-bit DP field is almost always precisely specified with only a few ranges and the rule-set is split into 2 buckets ([0x0000:0x00FF], [0x0100:0xFFFF]). The first B-tree

structure contains small ranges and single ports creating both large and small CBVs since many rules will apply to a common port number. The second B-tree structure is based on large ranges and the wildcard port of which there are very few, creating mostly small CBVs. Therefore, when a search is performed, the hardware ends up retrieving either two small CBVs or one large and one small CBV, which tends to produce large confidence intervals.

10.4 Random Rule Model

The Random Rule Model analysis is strictly intended to demonstrate how the algorithm will react towards discrete uniformly distributed random rule-sets. For this test case, there is no concept of inbound or outbound or source versus destination. Instead, only two field types are considered, a 32-bit field to mimic IPs and a 16-bit field to mimic ports.

The following criteria characterize the probability distribution for 32-bit fields in the Random Rule Model:

- 50% of the rules contain a random value selected based on a discrete uniform distribution in the IP space.
- 50% contain a random range based on a uniform distribution of prefix lengths.

The following criteria characterize the probability distribution for 16-bit fields in the Random Rule Model:

- 50% of the rules contain a random single 16-bit number selected based on a discrete uniform distribution from 0:65535.
- 50% contain a range according to Table 10-9.

Table 10-9: Random Rule-Set 16-bit Field Range Distribution

range	Probability	Description
3-30	25%	Allows small ranges between 3 and 30 in size.
100-1000	25%	Allows any range between 100 and 1000.
1000-10000	25%	Allows any range between 1000 and 10000.
10000-60000	25%	Allows any range between 10000 and 60000.
Total	100%	

Since there is far less structure in the Random Rule Model, the rules will be much more spread out. It is expected that this will tend to produce larger data structures, sparser bit-vectors and overall very fast search operations. Unfortunately, the search data structure quickly grew to

prohibitive sizes due to the limitations of the platform memory and the implementation of the software build operation. Therefore, only one test case (32-bit field, 16-bit field) was examined for rule sizes up to 8K. Figure 10-19 illustrates the search time and memory use for this case while Figure 10-20 demonstrates the software performance.

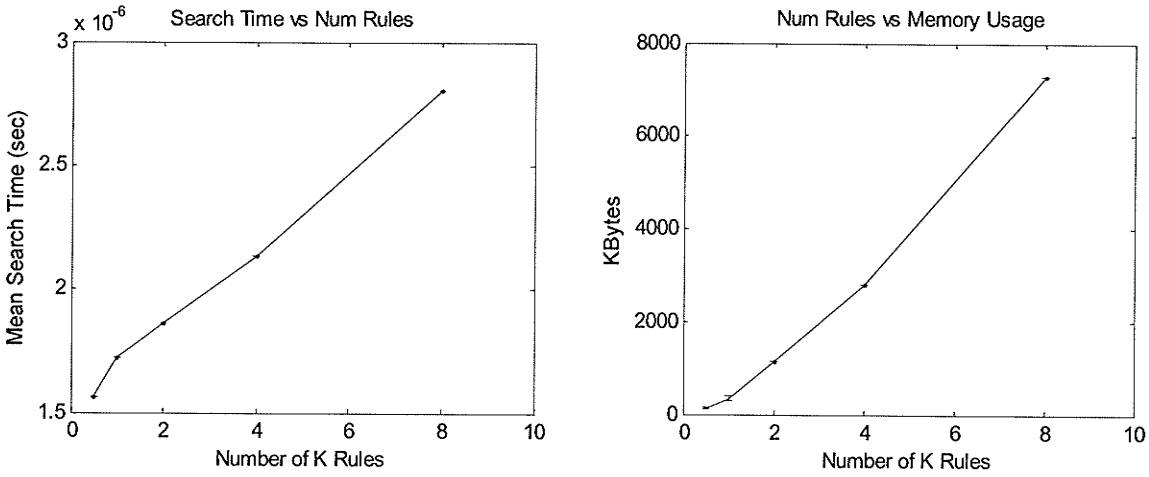


Figure 10-19: Random: (A) Search Time vs. Num. Rules (B) Memory vs. Num. Rules

The above figure for memory use shows that even for 8K rule-sets, the size of the final data structure was more than double that of the Perimeter Rule Model while the search time was nearly an order of magnitude faster. This is likely due to the reduction in wildcards in the Random Rule Model and emphasizes the point that the performance is highly influenced by the rule-set. However, for the Random Model it was somewhat expected that the search time plot would have revealed a more logarithmic relationship with respect to number of rules. This probably did not occur because the OR task is an $O(n)$ operation and the number of bits set per CBV grows linearly with the rule-set size.

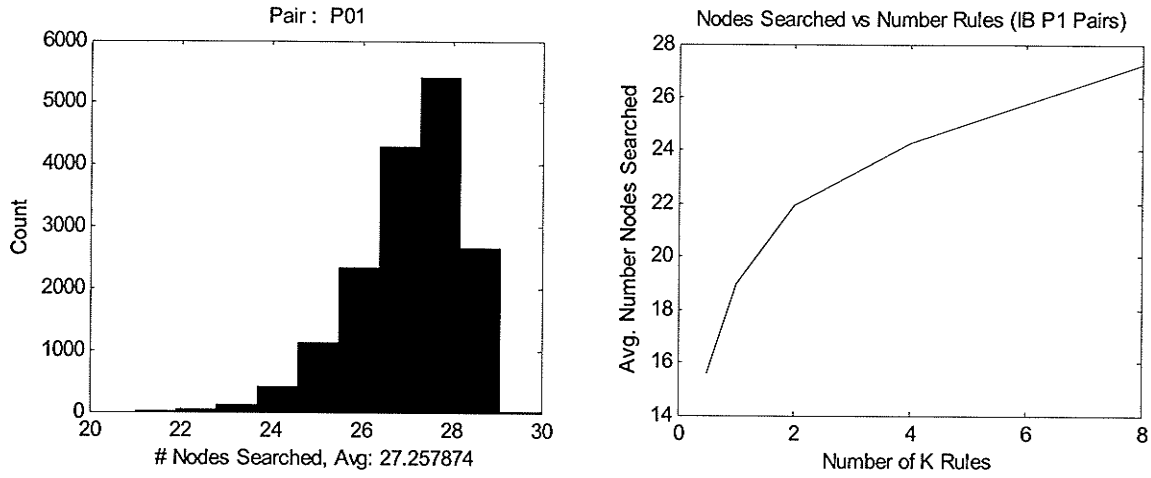


Figure 10-20 Random: (A) Histogram of Nodes Searched vs. Num. Rules (B) Nodes Searched vs. Num. Rules

In the above, Figure 10-20 indicates that the average number of nodes searched in the Random Rule Model was significantly higher than in the Perimeter Rule Model. This is to be expected due to the increased size of the search data structure.

10.5 Estimated ASIC Performance

The previous analysis was an emulation effort intended to validate the algorithm functionality and prove that the embedded system software runs correctly on the hardware design. While the prototyping platform imposes numerous limitations including the type of processor, bus speeds and bus widths between modules, the performance results obtained can also be used to help produce realistic ASIC performance estimates. The following case study suggests how the CBV algorithm could sustain a 1 Gigabit Ethernet line-rate for average sized packets by incorporating a few design enhancements based on available technologies. The known parameters for this analysis are specified in Table 10-10.

Table 10-10: 1 GE Case Study Parameters

Parameter	Value	Description
Rule-Set Size	8K	4K inbound and 4K outbound. Therefore, *n* will equal 4K
Bucket Factor	4	For this case study, each of the 4K rule-sets are split into 4 equal subsets.
d	4	Number of dimensions. (2-dimensions per search structure)
t	3	This is known as the minimum degree of the B-tree.
Node Size	96 bytes	Assumes a B-tree node size requires 96 bytes to store the ranges, CBV pointers etc.

Given an average packet size of 256 bytes, an average throughput of 0.45 Mpps or 2.2 us of processing time per packet must be maintained. Therefore, each stage of a pipelined implementation needs to complete within 2.2 us. For the CBV algorithm design, there are 3 stages of the pipe to consider: the software B-tree search, the PFAAE CBV retrieval and OR operation, and the system controller's final intersection. Analysis of the final intersection operation is omitted since it is a first match operation and presumed to take up less time than the OR. Therefore, the focus falls on the software and hardware operations of a two-dimensional search.

10.5.1 Software Analysis

Recalling from Section 5.4.3.1 the search complexity for the software portion of the algorithm is the time taken at each level of the tree, $O(th)$, where h is the height of the tree. By substituting for *h,* this can be re-written as $O(d \times t \log_t n)$, where the worst-case depth is really $d \times \log_t n$. Next, if a constant factor, *b,* is introduced splitting the rule-set into *b* equal groups of size $\frac{n}{b}$, then *b* two-dimensional data structures are created. This leads to Equation 10-2 for the worst-case number of nodes searched.

Equation 10-2: Worst-Case Number of Nodes Searched

$$\#NodesSearched = b \times d \times \log_t \frac{n}{b}$$

Substituting the known values from Table 10-10 into Equation 10-2 yields:

$$b \times d \times \log_t \frac{n}{b} = 4 \times 2 \times \log_3 \frac{4K}{4} \approx 15.25$$

In other words, the software searches a maximum of 16 nodes for each packet. To achieve a GE line-rate, the throughput calculation in Equation 10-3 must be maintained by the software search.

Equation 10-3: Software Memory Throughput

$$\#NodesSearched \times GEpps \times NodeSizeBytes = SW\ Memory\ Throughput$$

Substituting the known values in Table 10-10 into Equation 10-3 yields:

$$16 \times 0.4Mpps \times 96bytes \approx 0.7 \text{ GBps}$$

If DDR2-800 memory is used to store the B-trees with a clock speed of 400 MHz, the maximum theoretical throughput would be 6.4 GBps, which more than exceeds the requirement. Once the search is complete, the software also needs to send *b* pointers per packet to the PFAAE. To achieve GE throughput with a *b* factor of 4, this would require roughly 2 million pointers per second. Furthermore, for a 4 GB memory space and 32-bit pointers, this would require a sustainable throughput of 8 MB per second. A full-duplex, on-chip 32-bit wide ARM AHB bus running at 250 MHz would more than exceed the requirement, providing a throughput of up to 1 GB per second.

10.5.2 Hardware Analysis

The combined PFAAE CBV retrieval and OR operations need to come in under 2.2 us processing time. To meet this target, recommendations are made to improve the design based on statistics gathered in Section 10.3. If the CBV memory was upgraded to use RLDRAM II (single common I/O) with a 400 MHz clock, then a throughput of 800 Mbps per pin can be achieved. Furthermore, increasing the width of the data bus to 128-bits would provide a total throughput of 102.4 GBps. This is an eight-times improvement over the platform setup, which achieves 12.8 GBps running 32-bits wide with 200 MHz DDR. Based on a worst-case retrieval time of 2.3 us for a 4K rule-set in the Perimeter Rule Model, the total retrieval time could be reduced to less than 0.3 us as shown by carrying out Equation 10-4.

Equation 10-4: Estimated ASIC Retrieval Time

$$PlatformRetrievalTime \times \frac{PlatformMemoryThroughput}{ASICMemoryThroughput} = EstimatedASICRetrievalTime$$

$$2.3us \times \frac{12.8Gbps}{102.4Gbps} = 0.3us$$

This leaves 1.9 us for the OR operation. If a realistic ASIC clock rate of 300 MHz is assumed instead of the 50 MHz platform rate, then a six-times improvement is immediately achieved. Based on the Perimeter Rule Model worst-case statistics for 4K rule-sets, this leads to an OR time of 1.3 us as shown by completing Equation 10-5.

Equation 10-5: Estimated ASIC Hardware ORing Time

$$(TotalHWTime - RetrievalTime)\frac{PlatformClockSpeed}{ASICClockSpeed} = estimatedORTime$$

$$(10.3us - 2.3us) \times \frac{50MHz}{300MHz} = 1.3us$$

Cumulatively, the retrieval and OR time fall within the 2.2 us requirement. Therefore, it is reasonable to state that if the CBV packet classification algorithm is implemented in an ASIC with the aforementioned improvements, it could sustain a Gigabit Ethernet line rate on average-sized packets.

Chapter 11: Conclusions and Future Work

This thesis has developed a novel packet classification algorithm, which builds upon the Lucent and Aggregated Bit-Vector algorithms [1], [2]. Both of these schemes decompose the general PC problem into one-dimensional problems that return bit-vector solutions. The latter adds the notion of aggregation and rule rearrangement to make the Lucent BV scheme more scalable. Two new ideas were introduced, creating what is called the Compressed Bit-Vector scheme. Since it is well known that many efficient solutions exist for 2-D packet classification, the proposed solution breaks the general multi-dimensional problem into more manageable two-dimensional problems, that each produce compressed bit-vector solutions. This provides a possible framework that will allow for the incorporation of various two-dimensional search algorithms into one system for performing multi-dimensional classification. By strictly storing and operating on hierarchically compressed bit-vectors, it is also shown that the algorithm's memory usage is scalable to large rule-sets.

As with the ABV and Lucent BV schemes, the CBV algorithm is equally amenable to a hardware and software implementation. In particular, the initial searches in the two-dimensional B-tree query can be pipelined with the PFAAE operations on the compressed bit-vectors. Likewise, the final intersection can be pipelined with the PFAAE OR operation.

As a proof-of-concept, a portion of the CBV algorithm framework, namely one 2-D search, was implemented on a System-on-Chip rapid prototyping platform containing processors, custom computing hardware and embedded memory. Results from functional and performance tests performed on the prototyping platform have lead to the following conclusions regarding the CBV algorithm and on platform-based design in general.

11.1 Conclusions

The platform-based design methodology was found to provide many benefits without a commitment to silicon. It allowed for embedded software development early in the design cycle, the resolution of hardware and software integration problems, the ability to validate 3rd party IP blocks in a system environment (e.g. the RLDRAM interface) and the capability to experiment with the implementation.

Using the platform-based prototype, the algorithm's functionality and its performance were verified. All of the functional tests performed were successful, indicating a fairly healthy design. This analysis also revealed the major system limitation to be the B-tree SDRAM, and the performance bottlenecks as the processor speed followed by the PFAAE OR operation. This was not a complete surprise since the platform imposed many limitations including the processor, clock speeds, memory resources and bus width requirements for bit-vectoring. The B-tree SDRAM imposed a hard limit on the size of rule-sets that could be tested while the bus limitations increased the depth cycles for memory accesses.

Nonetheless, the performance was evaluated on synthetic rule-sets modeled after real firewall rule-set statistics. To obtain a more accurate picture of the CBV algorithm properties, space optimization techniques were employed to maximize the size of trial rule-sets. Separate data structures were built and tested for inbound and outbound rule-sets and a space-time trade-off called the bucket factor was introduced to further partition the rule-sets into b groups.

During investigation and analysis, the best field orders were identified based on size and time. It was observed that the total data structure size of the best 2-D pair combinations grew at slightly less than $5/4$ times with respect to the number of rules (Section 10.3.2). This is much slower than the theoretical upper bound, $O(n^3)$, and indicates that the CBV algorithm's actual space requirements fall well within the capabilities of modern hardware. Using the platform, a throughput of roughly 18 us per packet or 56,000 packets per second was sustained for inbound traffic and a throughput of approximately 42 us per packet or 24,000 packets per second was sustained for outbound traffic with 16K rules (Section 10.3.3). This is far from GE speeds, but the platform has many limitations and is running at reduced clock speeds compared to an ASIC design. To that end, a simple case study was performed to illustrate how an ASIC implementation of the algorithm could sustain a Gigabit Ethernet line-rate for an 8K rule-set (Section 10.5). This example showed that the CBV algorithm can, indeed, be an efficient hardware-amenable algorithm for firewall packet filtering and classification.

11.2 Future Work

The SoC platform implementation of the CBV packet classification algorithm leaves several directions for future work. It is anticipated that the execution of additional what-if-analysis on the system can lead to further improvements in the design. The following are a few recommendations for future work:

a) Implement pre-processing steps for rule rearrangement as described in the ABV scheme [2]. This was found to have a much larger impact than aggregation alone. Therefore, it is equally likely to have a strong impact on the CBV scheme performance.

b) Implement a variety of 2-D search algorithms in place of the B-tree search. For the IP address fields, a search based on tries would likely lead to sparser bit-vectors and a significant improvement in performance.

c) Quantify the performance gains for various implementations of the bucket factor. In the current scheme the rule-sets were split into b buckets, according to range settings. It would be interesting to quantify the impact for various settings of b as well as different splitting criteria. For example, if the rules were split into equal groups based on priority, then the OR operation would become a concatenation of vectors.

d) Explore new ways of generating rule-sets that will stress the CBV algorithm. It would be interesting to obtain actual commercial firewall rule-sets and also to explore the impact of various degrees of wildcard rules.

e) Implement additional logic to perform multiple 2-D searches in parallel along with the final intersection operation. This could be achieved by attaching additional cores and logic tiles to the platform.

f) Port the system over to work on a processor, which is more suited to classification. A MIPS processor might be a suitable alternative

References

[1] T.V. Lakshman, D. Stiliadis, "High Speed Policy Based Packet Forwarding Using Efficient Multi Dimensional Range Matching," *ACM SIGCOMM Computer Communication Review,* vol.28, no. 4, pp. 203-214, Oct. 1998.

[2] F. Baboescu, G. Varghese, "Aggregated Bit Vector Search Algorithms for Packet Filter Lookups," *UCSD Technical Report cs2001-0673*, pp.1-27, June 2001.

[3] F. Baboescu, G. Varghese, "Scalable Packet Classification," *ACM SIGCOMM Computer Communication Review,* vol. 31, no. 4 pp. 199-210, Oct. 2001.

[4] A. Feldmann, S. Muthukrishnan, "Tradeoffs for Packet Classification," *Proceedings of the Conference on Computer Communications (IEEE INFOCOM 2000)*, vol. 3, pp. 1193-1202, Mar. 2000.

[5] C.J. Stuart, D.J. Cornelsen, "A Decomposition Approach to the Multi-Dimensional IP Packet Classification Problem," *Proceedings of the IASTED International Conference on Computer Science And Technology*, pp. 342-347, May 2003.

[6] C. Macian, R. Finthammer, "An Evaluation Of The Key Design Criteria To Achieve High Update Rates In Packet Classifiers," *IEEE Network*, vol. 15, no.6, pp. 24-29, Nov. 2001.

[7] V. Sahasranaman, M. Buddhikot, "Comparative Evaluation of Software Implementations of Layer-4 Packet Classification Schemes," *Proceedings of the Ninth International Conference on Network Protocols (ICNP'01),* pp. 220-228, Nov. 2001.

[8] V. Srinivasan, G. Varghese, S. Suri, M. Waldvogel, "Fast and Scalable Layer Four Switching," *ACM SIGCOMM Computer Communication Review*, vol.28, no. 4, pp. 191-202, Oct. 1998.

[9] M.M. Buddhikot, S. Suri, M. Waldvogel, "Space decomposition for Fast layer 4 switching," *Proceedings of the IFIP TC6 WG6.1 & WG6.4 / IEEE ComSoc TC on Gigabit Networking Sixth International Workshop on Protocols for High Speed Networks VI*, pp. 25-42, Aug. 1999.

[10] B. Lampson, V. Srinivasan, G. Varghese, "IP Lookups using Multiway and Multicolumn Search," *IEEE/ACM Transactions on Networking (TON)*, vol. 7, no. 3, pp. 324-334, June 1999.

[11] M. Waldvogel, G. Varghese, J. Turner, B. Plattner, "Scalable high-speed IP routing lookups," *Proceedings of the ACM SIGCOMM '97 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pp. 25-36, Sept. 1997.

[12] P. Gupta, N. McKeown, "Classifying packets using hierarchical intelligent cuttings," *IEEE Micro,* vol. 20, no. 1, pp. 34-41, Jan 2000.

[13] V. Srinivasan, S. Suri, G. Varghese, and M. Waldvogel, "Packet classification using tuple space search," *Proceedings of the ACM SIGCOMM '99 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pp. 135-146, Sept. 1999.

[14] P. Gupta, S. Lin, N. McKeown, "Routing Lookups In Hardware At Memory Access Speeds," *Proceedings of the Conference on Computer Communications (IEEE INFOCOM '98)*, vol. 3, pp. 1241-1248, Mar. 1998.

[15] D. Rovniagin, A. Wool, "The Geometric Matching Algorithm for Firewalls," *Tel Aviv University Technical Report Ees2003-6*, pp. 1-17, July 2003.

[16] S. Iyer, R. Rao Kompella, A. Shelat, "ClassiPI: An architecture for fast and flexible packet classification," *IEEE Network*, vol. 15, no. 2, pp. 33-41, Mar. 2001.

[17] F. Baboescu, S. Singh, G. Varghese, "Packet classification for core routers: Is there an alternative to CAMs?," *Proceedings of the Conference on Computer Communications (IEEE INFOCOM '03)*, vol. 1, no. 30, pp. 53-63, Mar. 2003.

[18] T. H Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, The MIT Press, Second Edition, 2001, 1180 pp.

[19] A. S. Tanenbaum, *Computer Networks,* Upper Saddle River, NJ: Prentice Hall Inc., Third Edition, 1996, 813 pp.

[20] J. McHugh, A. Christie, J. Allen, "Defending Yourself: The Role of Intrusion Detection Systems," *IEEE Software*, vol. 17, no. 5, pp. 42-51, Sept. 2000.

[21] E. D. Zwicky, S. Cooper, D.B. Chapman. *Building Internet Firewalls,* Sebastpol, CA: O-Reilly & Associates Inc., 2nd Edition, June 2000, 869 pp.

[22] PMC-Sierra Inc., *Toward Content-Based Classification,* PMC-2002233 White Paper, Issue 1, Feb. 2001.

[23] R.J. Ellison, D.A. Fisher, R.C. Linger, H.F. Lipson, T.A. Longstaff, N.R. Mead, "Survivability: protecting your critical systems," *IEEE Internet Computing*, vol. 3, no. 6, pp 55-63, Nov. 1999.

[24] A. Kuznetsov, Bitmagic Hierarchical Compression. [Online], http://bmagic.sourceforge.net/hCompression.html (available as of June 2003).

[25] ARM Limited, *ARM Integrator™/AP: User Guide*, 2001.

[26] ARM Limited, *AMBA™ Specification (Rev. 2)*, 1999.

[27] ARM Limited, *ARM Integrator™/CM7TDMI: User Guide*, 1999.

[28] ARM Limited, *ARM Integrator™/LM-XC2V4000+: User Guide*, 2002.

[29] H. Chang, L. Cooke, M. Hunt, G. Martin, A. McNelly, L. Todd, *Surviving the SoC Revolution A Guide to Plaform-Based Design.*. Norwell, MA: Kluwer Academic Publishers, 1999, 235 pp.

[30] P. Rashinkar, P. Paterson, L. Singh, *System-on-a-Chip Verification Methodology and Techniques*. Norwell, MA: Kluwer Academic Publishers, 2001, 372 pp.

[31] Memec Design, *MC-XIL-RLDRAM RLDRAM Controller User Guide*, Version 0.7, Nov. 2002, 58 pp.

[32] N. Sawyer, *High-Speed Data Serialization and Deserialization (840 Mb/s LVDS),* Xilinx Inc., Application Note Virtex II Family, XAPP265, Version 1.3, pp. 1-13, June 2002.

[33] J. Zaidi, Different platform types are needed for SoC design, EE Times [Online], http://www.eetimes.com/showArticle.jhtml?articleID=17408109 (available as of Jan. 31 2003).

[34] Xilinx Inc., *Xilinx Development System Reference Guide*, UG000 (v3.5.1) , April 30, 2003, 476 pp.

[35] Infineon Technologies, *Graphics & Speciality DRAMs 256 Mbit DDR Reduced Latency DRAM*, Marketing-Communication, Version 1.60, July 2003.

[36] Xilinx Inc., *Xilinx LogiCore, Asynchronous FIFO v5.1*, Xilinx Product Specification DS232 (v0.2) March 28, 2003.

[37] J. Andrews, "ARM SoC Verification Matrix Improves HW/SW Co-Verification," *Electronic Design Processes 2004,* pp. 1-6, April 2004.

[38] Canadian Microelectronics Corporation, *CMC Rapid-Prototyping Platform: Design Flow Guide*, Version 1.0, Feb. 8 2002.

[39] EZChip Technologies, *7-Layer packet processing: A Performance Analysis*, White paper, pp. 1-4, July 2000.

[40] EZChip Technologies, *The Role of Memory in NPU System Design*, White paper, pp. 1-6, http://www.ezchip.com/html/in_emp.html , July 24 2003

[41] Memec Design, *P160 Module Specification*, Version 1.3, PN# DS-MANUAL-P160-SPEC, Nov. 2002.

[42] Canadian Microelectronics Corporation, CMC-CMP-MOSIS 2001, [Online]. http://www.mseconference.org/mse_03_archive/mse03_5_cmc_cmp_mosis_v2.pdf (available as of Nov. 2001).

[43] R. Abrishmani, "Design Strategies and Methodologies, Emulation Emerges as a Centerpiece in SoC Verification," *ARM Information Quarterly*, vol. 3, no. 3, pp. 64-66, 2004.

[44] Wikipedia, *System-on-a-chip*, [Online], http://en.wikipedia.org/wiki/System-on-a-chip, (available as of May 1 2007).

[45] Wikipedia, *Big O notation,* [Online], http://en.wikipedia.org/wiki/Big_O_notation, (available as of Jan 15 2006).

[46] Wikipedia, *Trie*, [Online], http://en.wikipedia.org/wiki/Trie, (available as of Jan 15 2006).

[47] Netlogic MicroSystems, *NSE5512 Press Release,*[Online], http://www.netlogicmicro.com/4-news/pr/2003/03-02-26.htm, (available as of Feb. 3 2003).

[48] IEEE, *Part 3: Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications*, IEEE Std. 802.-2000 Edition, 2000.

[49] Wikipedia, *Content Addressable Memory*, [Online], http://en.wikipedia.org/wiki/Content-addressable_memory (available as of Feb. 8 2007).

Appendix A: File I/O Listing

A.1 CBV List File

```
#-------------------------------------------------------------------------
#    Starting cbv_list LogFile
#
#    Seed: 0
#    Mode: 0
#    Rule Size: 512
#    Direction: Inbound
#    Description:  This file contains all of the CBVs generated for a
#    particular rule-set.
#    Example:
#
#    0x00000000 0x00000001 0x00000001 0x00000001 0x00000200 0x00000000
#    0x00000010 0x00000000
#
#    {Level 1 or root} {L2 Count, L3 Count} {L2 Vectors} {L3 Vectors}
#
#    {0x00000000 0x00000001} {0x00000001 0x00000001} {0x00000200 0x00000000}
#    {0x00000010 0x00000000}
#
#-------------------------------------------------------------------------
0x00000000 0x00000001 0x00000001 0x00000001 0x00000200 0x00000000 0x00000010
0x00000000
0x00000000 0x00000001 0x00000001 0x00000001 0x00000001 0x00000000 0x00040000
0x00000000
0x00000000 0x00000001 0x00000001 0x00000001 0x00002000 0x00000000 0x01000000
0x00000000
0x00000000 0x00000001 0x00000001 0x00000001 0x00000020 0x00000000 0x00000400
0x00000000
0x00000000 0x00000001 0x00000001 0x00000001 0x00000010 0x00000000 0x00400000
0x00000000
0x00000000 0x00000001 0x00000001 0x00000001 0x00001000 0x00000000 0x00000400
0x00000000
0x00000000 0x00000001 0x00000001 0x00000001 0x00008000 0x00000000 0x00000800
0x00000000
0x00000000 0x00000001 0x00000001 0x00000001 0x00000800 0x00000000 0x00002000
0x00000000
0x00000000 0x00000001 0x00000001 0x00000001 0x00001000 0x00000000 0x00200000
0x00000000
0x00000000 0x00000001 0x00000001 0x00000002 0x0000408A 0x00000000 0x00010002
0x00000020 0x00010000 0x00080000
0x00000000 0x00000001 0x00000001 0x00000003 0x0000408B 0x00000000 0x00400000
0x00010002 0x00000020 0x00010000 0x00080000 0x00000000
0x00000000 0x00000001 0x00000001 0x00000004 0x0000C156 0x00000000 0x00004020
0x09008000 0x04001000 0x00400000 0x00080000 0x00000001 0x00002000 0x00000000
0x00000000 0x00000001 0x00000001 0x00000005 0x000016ED 0x00000000 0x10000000
0x00200000 0x00200000 0x00010000 0x00000050 0x00440400 0x04000000 0x00010000
0x00000040 0x00000000
#-------------------------------------------------------------------------
#    Completed cbv_list LogFile
#
#-------------------------------------------------------------------------
```

A.2 CBV Pointers File

```
#-------------------------------------------------------------------------
#    Starting cbv_ptrs LogFile
#
#    Seed: 0
#    Mode: 0
#    Rule Size: 512
#    Direction: Inbound
#    Description: Each line represents a pointer into RLDRAM, which corresponds
#    to the start of a CBV.
#
#
#-------------------------------------------------------------------------
0x00000000
0x00000004
0x00000008
0x0000000C
0x00000010
0x00000014
0x00000018
0x0000001C
0x00000020
0x00000024
0x00000028
0x0000002C
0x00000030
0x00000034
0x00000038
0x0000003C
0x00000040
0x00000044
0x00000048
0x0000004C
0x00000050
0x00000054
0x00000058
0x0000005C
0x00000060
0x00000064
0x00000068
0x0000006C
0x00000070
0x00000074
0x00000078
0x0000007C
0x00000080
0x00000087
  :
0x000013E7
#-------------------------------------------------------------------------
#    Completed cbv_ptrs LogFile
#
#-------------------------------------------------------------------------
```

A.3 CBV Count File

```
#-------------------------------------------------------------------------
#    Starting cbv_count LogFile
#
#    Seed: 0
#    Mode: 0
#    Rule Size: 512
#    Direction: Inbound
#    Description: Each line represents a count of the number of rules set in a
#    CBV.
#-------------------------------------------------------------------------
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
:
:
:
10
10
3
2
11
5
6
11
12
11
11
#-------------------------------------------------------------------------
#    Completed cbv_count LogFile
#
#-------------------------------------------------------------------------
```

A.4 Parsed Rule List File

```
#-------------------------------------------------------------------------
#    Starting parsed Rule List LogFile
#
#    Seed: 0
#    Mode: 0
#    Rule Size: 512
#    Direction: Inbound
#    Description: This file contains the start and end ranges for a pair
#    of fields of a two-dimensional search.
#        Example:
#    {Rule ID} {Start Field 1} {End Field 1} {Start Field 2} {End Field 2}
#    0x00000000 0x00000000 0xFFFFFFFF 0xB3891800 0xB38918FF
#
#-------------------------------------------------------------------------

0x00000000 0x00000000 0xFFFFFFFF 0xB3891800 0xB38918FF
0x00000001 0x00000000 0xFFFFFFFF 0xB3834FC7 0xB3834FC7
0x00000002 0x00000000 0xFFFFFFFF 0xB3895EBA 0xB3895EBA
0x00000003 0x00000000 0xFFFFFFFF 0xB3879D1A 0xB3879D1A
0x00000004 0x00000000 0xFFFFFFFF 0xB3842C00 0xB3842CFF
0x00000005 0x00000000 0xFFFFFFFF 0xB388EA79 0xB388EA79
0x00000006 0x00000000 0xFFFFFFFF 0xB38813F2 0xB38813F2
0x00000007 0x00000000 0xFFFFFFFF 0xB3894B08 0xB3894B08
0x00000008 0x00000000 0xFFFFFFFF 0xB3850000 0xB385FFFF
0x00000009 0x00000000 0xFFFFFFFF 0xB388F730 0xB388F730
0x0000000A 0x00000000 0xFFFFFFFF 0xB389EC00 0xB389ECFF
0x0000000B 0x1C388000 0x1C38FFFF 0xB388721B 0xB388721B
0x0000000C 0x00000000 0xFFFFFFFF 0xB388B5A9 0xB388B5A9
0x0000000D 0x00000000 0xFFFFFFFF 0xB3876000 0xB38760FF
0x0000000E 0x00000000 0xFFFFFFFF 0xB3860000 0xB386FFFF
0x0000000F 0x00000000 0xFFFFFFFF 0xB3812F86 0xB3812F86
0x00000010 0x00000000 0xFFFFFFFF 0xB388FFD1 0xB388FFD1
0x00000011 0x00000000 0xFFFFFFFF 0xB3870000 0xB387FFFF
0x00000012 0x7C806F80 0x7C806FBF 0xB384E900 0xB384E9FF
0x00000013 0x00000000 0xFFFFFFFF 0xB380EB00 0xB380EBFF
0x00000014 0x00000000 0xFFFFFFFF 0xB389B700 0xB389B7FF
0x00000015 0x00000000 0xFFFFFFFF 0xB3808900 0xB38089FF
0x00000016 0x00000000 0xFFFFFFFF 0xB381FA70 0xB381FA7F
0x00000017 0x00000000 0xFFFFFFFF 0xB3859AE0 0xB3859AFF
0x00000018 0x00000000 0xFFFFFFFF 0xB3865A20 0xB3865A2F
0x00000019 0x00000000 0xFFFFFFFF 0xB3813510 0xB381351F
0x0000001A 0x05551000 0x05551FFF 0xB3867748 0xB386774F
0x0000001B 0x00000000 0xFFFFFFFF 0xB3865F0B 0xB3865F0B
0x0000001C 0x00000000 0xFFFFFFFF 0xB3800000 0xB380FFFF
0x0000001D 0x00000000 0xFFFFFFFF 0xB386134B 0xB386134B
0x0000001E 0x00000000 0xFFFFFFFF 0xB384EAA3 0xB384EAA3
     :          :          :          :          :
     :          :          :          :          :
     :          :          :          :          :
0x000001FE 0x00000000 0xFFFFFFFF 0xB3897200 0xB38972FF
0x000001FF 0x00000000 0xFFFFFFFF 0xB386DF00 0xB386DFFF
#-------------------------------------------------------------------------
#    Completed parsed Rule List LogFile
#
#-------------------------------------------------------------------------
```

A.5 Search Results File

```
#------------------------------------------------------------------------
#     Starting Search Results LogFile
#
#     Seed: 0
#     Mode: 0
#     Rule Size: 512
#     Direction: Inbound
#     Note: The resultant packet includes the CBV and time for the PFAAE to perform the OR
#     {Header} {Level 1} {L2 Count, L3 Count} {L2 Vectors} {L3 Vectors}
#     Example:
#     {0x00001E0D}: OR time = 0x1E, Length = 0xD
#     {0x00000000 0x00000001} {0x00000001 0x00000003} {0x00000073 0xXXXXXXXX}
#     {0x00000001 0x00000001 0x02000000 0x00000010 0x00000080 0xXXXXXXXX}
#------------------------------------------------------------------------
0x00001E0D  0x00000000  0x00000001  0x00000001  0x00000003  0x00000073  0xBB89F7E7  0x00000001  0x00000001  0x02000000
0x00000010 0x00000080 0x08000080
0x00001E0D  0x00000000  0x00000001  0x00000001  0x00000003  0x00000073  0xBB89F7E7  0x00000001  0x00000001  0x02000000
0x00000010 0x00000080 0x08000080
0x00001E0D  0x00000000  0x00000001  0x00000001  0x00000003  0x00000073  0xBB89F7E7  0x00000001  0x00000001  0x02000000
0x00000010 0x00000080 0x08000080
0x00001E0D  0x00000000  0x00000001  0x00000001  0x00000003  0x00000073  0xBB89F7E7  0x00000001  0x00000001  0x02000000
0x00000010 0x00000080 0x08000080
0x00002A0F  0x00000000  0x00000001  0x00000001  0x00000004  0x0000692D  0xBB89F7E7  0x00000002  0x20000000  0x08000040
0x00800000 0x80000000 0x00000100 0x02000000 0x80000100
    :           :           :           :           :           :           :           :           :           :
    :           :           :           :           :           :           :           :           :           :
    :           :           :           :           :           :           :           :           :           :
0x0000160B  0x00000000  0x00000001  0x00000001  0x00000002  0x00008201  0xBB89F7E7  0x00004000  0x00000400  0x80000040
0x00000080
#------------------------------------------------------------------------
#     Completed Search Results LogFile
#
#------------------------------------------------------------------------
```

A.6 Search Timer Results File

```
#-------------------------------------------------------------------------
#    Starting Search Timer Results LogFile
#
#    Seed: 0
#    Mode: 0
#    Rule Size: 512
#    Direction: Inbound
#    Description:
#    0x00000CA9 : Total Search Time measured in 20 MHz clock cycles.
#    0x0000000C : Count of the number of keys examined in the software search.
#    0x0000000A : Count of the number of nodes accessed during the software
#    search.
#-------------------------------------------------------------------------
0x00000CA9
0x0000000C
0x0000000A
0x00000CE1
0x0000000D
0x0000000A
0x00000CBE
0x00000015
0x0000000A
0x00000E5B
0x0000000F
0x0000000A
0x00000C75
0x0000000D
0x0000000A
0x00000D39
0x0000000F
0x0000000A
0x00000D13
0x0000000E
0x0000000A
0x00000D2D
0x00000011
0x0000000A
0x00000E87
0x0000000B
0x0000000A
0x00000CD3
0x0000000F
0x0000000A
0x00000EAC
0x0000000F
    :
    :
    :
0x00000BCF
0x00000013
0x00000008
#-------------------------------------------------------------------------
#    Completed Search Timer Results LogFile
#
#-------------------------------------------------------------------------
```

A.7 Pointer Timer Results File

```
#-------------------------------------------------------------------------
#    Starting Pointer Timer Results LogFile
#
#    Seed: 0
#    Mode: 0
#    Rule Size: 512
#    Direction: Inbound
#    Description: Each line represents the time, measured in 20 MHz clock
#    cycles for the hardware to retrieve and OR the CBVs from a search.
#-------------------------------------------------------------------------
0x00000026
0x00000023
0x00000026
0x00000023
0x00000024
0x00000028
0x00000028
0x00000028
0x0000002B
0x00000028
0x00000023
0x00000023
0x00000026
0x00000023
0x00000027
0x00000023
0x00000024
0x00000023
0x00000024
0x00000023
0x00000023
0x00000024
0x00000023
0x00000024
0x00000023
0x00000020
0x00000020
0x00000022
0x00000024
0x00000023
0x00000020
0x00000020
0x00000022
0x00000024
0x00000023
    :
    :
    :
0x00000023
0x00000020
0x0000001F
#-------------------------------------------------------------------------
#    Completed Pointer Timer Results LogFile
#
#-------------------------------------------------------------------------
```

A.8 Tree LogFile

```
#-------------------------------------------------------------------------
#    Starting Tree LogFile
#
#    Seed: 0
#    Mode: 0
#    Rule Size: 512
#    Direction: Inbound
#    Description: This file is used for examining trees and reproducing them in
#    simulation.  It indicates the start and end ranges for each key in up to 4
#    B-trees along with the rules stored at each key.
#-------------------------------------------------------------------------
# Level 0
# Level:  0 Start: B3844700 End: B38447FF
0x00000001 0x00000124
# Level:  0 Start: B384E900 End: B384E9FF
0x00000001 0x00000012
# Level:  1 Start: B3860000 End: B386FFFF
0x00000001 0x000001B8
# Level:  1 Start: B3886CE4 End: B3886CE4
0x00000001 0x000000AA
# Level 2
# Level 3
# Level:  0 Start: B3832500 End: B38325FF
0x0000000A 0x0000005D 0x00000066 0x0000007B 0x000000A6 0x000000B7 0x0000011F
0x00000168 0x000001B9 0x000001C8 0x000001DF
# Level:  0 Start: B3860100 End: B3860EFF
0x00000003 0x0000000E 0x0000012A 0x000001E6
# Level:  0 Start: B38834CE End: B38834FF
0x00000002 0x00000072 0x00000151
# Level:  1 Start: B3808A00 End: B3808AFF
0x0000000B 0x0000001C 0x00000055 0x00000075 0x000000B0 0x000000C6 0x000000EA
0x000000F2 0x000000F6 0x0000013A 0x00000150 0x00000186
# Level:  1 Start: B381FA70 End: B381FA7F
0x00000006 0x00000016 0x00000021 0x00000030 0x00000065 0x000000F0 0x000001D3
# Level:  1 Start: B3828800 End: B38288FF
0x0000000B 0x00000025 0x0000002E 0x0000004F 0x00000058 0x0000005B 0x0000008C
0x0000009A 0x000000D6 0x00000113 0x000001C0 0x000001ED
# Level:  2 Start: B3804000 End: B38040FF
0x0000000C 0x0000001C 0x00000055 0x00000075 0x000000B0 0x000000C4 0x000000C6
0x000000EA 0x000000F2 0x000000F6 0x0000013A 0x00000150 0x00000186
# Level:  3 Start: B380284B End: B3802B82
0x0000000B 0x0000001C 0x00000055 0x00000075 0x000000B0 0x000000C6 0x000000EA
0x000000F2 0x000000F6 0x0000013A 0x00000150 0x00000186
      :       :
      :       :
      :
# Level:  4 Start: B3800000 End: B3800BFF
0x0000000B 0x0000001C 0x00000055 0x00000075 0x000000B0 0x000000C6 0x000000EA
0x000000F2 0x000000F6 0x0000013A 0x00000150 0x00000186
#-------------------------------------------------------------------------
#    Completed Tree LogFile
#
#-------------------------------------------------------------------------
```

A.9 Testpoints File

```
#-------------------------------------------------------------------------
#    Starting Testpoints LogFile
#
#    Seed: 0
#    Mode: 0
#    Rule Size: 512
#    Direction: Inbound
#    Description:  This file contains test points for performing search
#    operations.  Each test point is selected from within a generated rule-set
#    and represents packet header fields for the TCP protocol according to the
#    selected mode.
#    {Testpoint Dimension 1} {Testpoint Dimension 2}
#    Example: SIP,DIP
#    {0x145E5426} {0xB389183C}
#-------------------------------------------------------------------------
0x145E5426 0xB389183C
0x464FB560 0xB389181B
0xE7A123AF 0xB3891804
0x62B3D354 0xB389180B
0x9E5EA697 0xB3891858
0x3F11B334 0xB3834FC7
0x780740FC 0xB3834FC7
0xC5FF10C1 0xB3834FC7
0x49758874 0xB3834FC7
0x23316096 0xB3834FC7
0x7362FC0A 0xB3895EBA
0x6CD3078A 0xB3895EBA
0x3E0BE0C8 0xB3895EBA
0x897540CF 0xB3895EBA
0x1659655C 0xB3895EBA
0xC8F7D807 0xB3879D1A
0x67B70F8E 0xB3879D1A
0x0B344870 0xB3879D1A
0xF3BADB0F 0xB3879D1A
0xE64CAC31 0xB3879D1A
0x3B9C00F2 0xB3842C7F
0x6836FC34 0xB3842CF1
0xB2148065 0xB3842C5F
0xF5682277 0xB3842C87
0xEDC2EC7F 0xB3842C98
0xAEAF7B8D 0xB388EA79
0x53335410 0xB388EA79
0x44AF6DB2 0xB388EA79
0x43D57B34 0xB388EA79
0x8046FB81 0xB388EA79
0x25AD54AE 0xB38813F2
     :          :
     :          :
     :          :
0xBD4BEEAF 0xB386DFC4
0xE6E96DC1 0xB386DFE7
#-------------------------------------------------------------------------
#    Completed Testpoints LogFile
#
#-------------------------------------------------------------------------
```

Appendix B: Software Nodes Searched

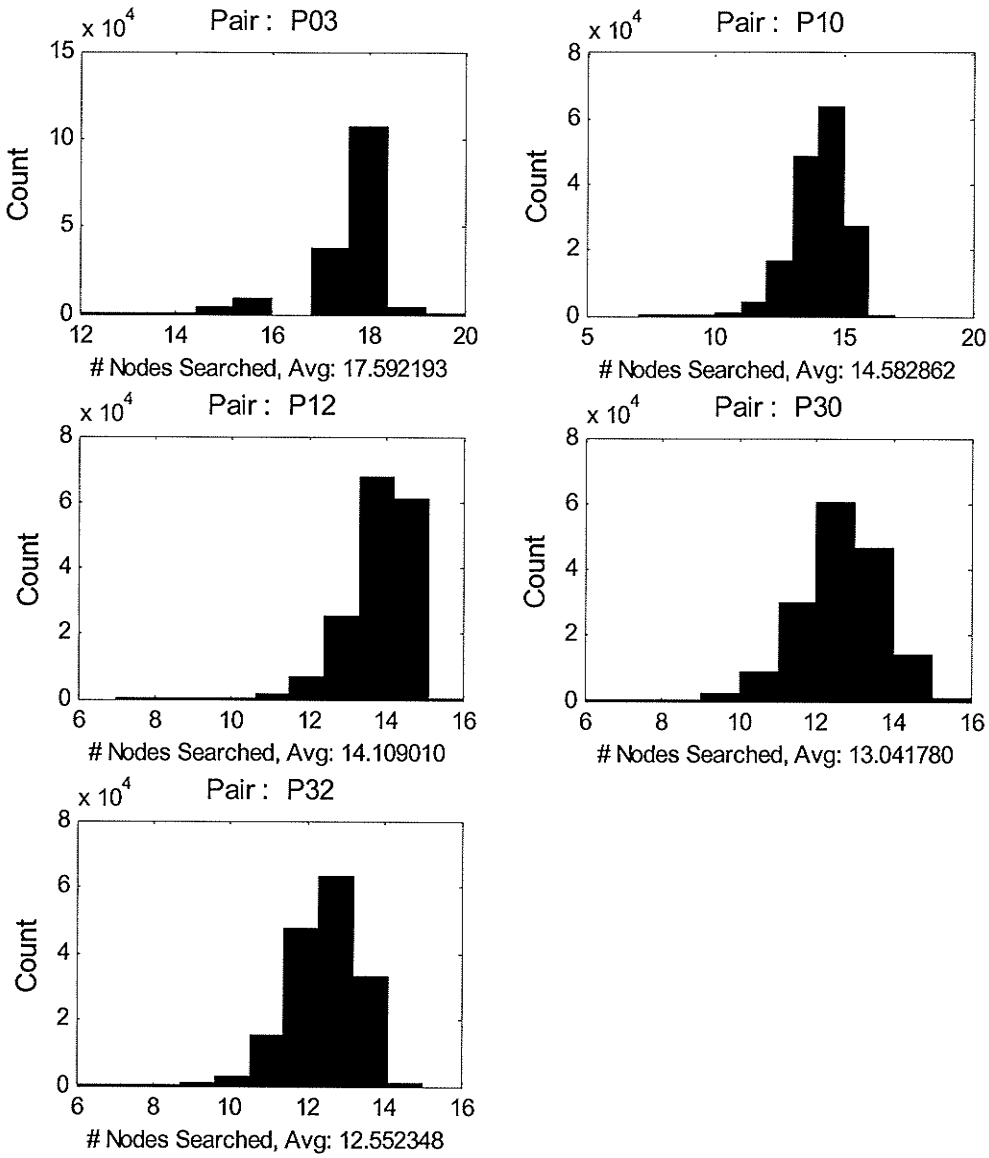


Figure B-1: Distribution of Inbound Nodes Searched for 16K Rules

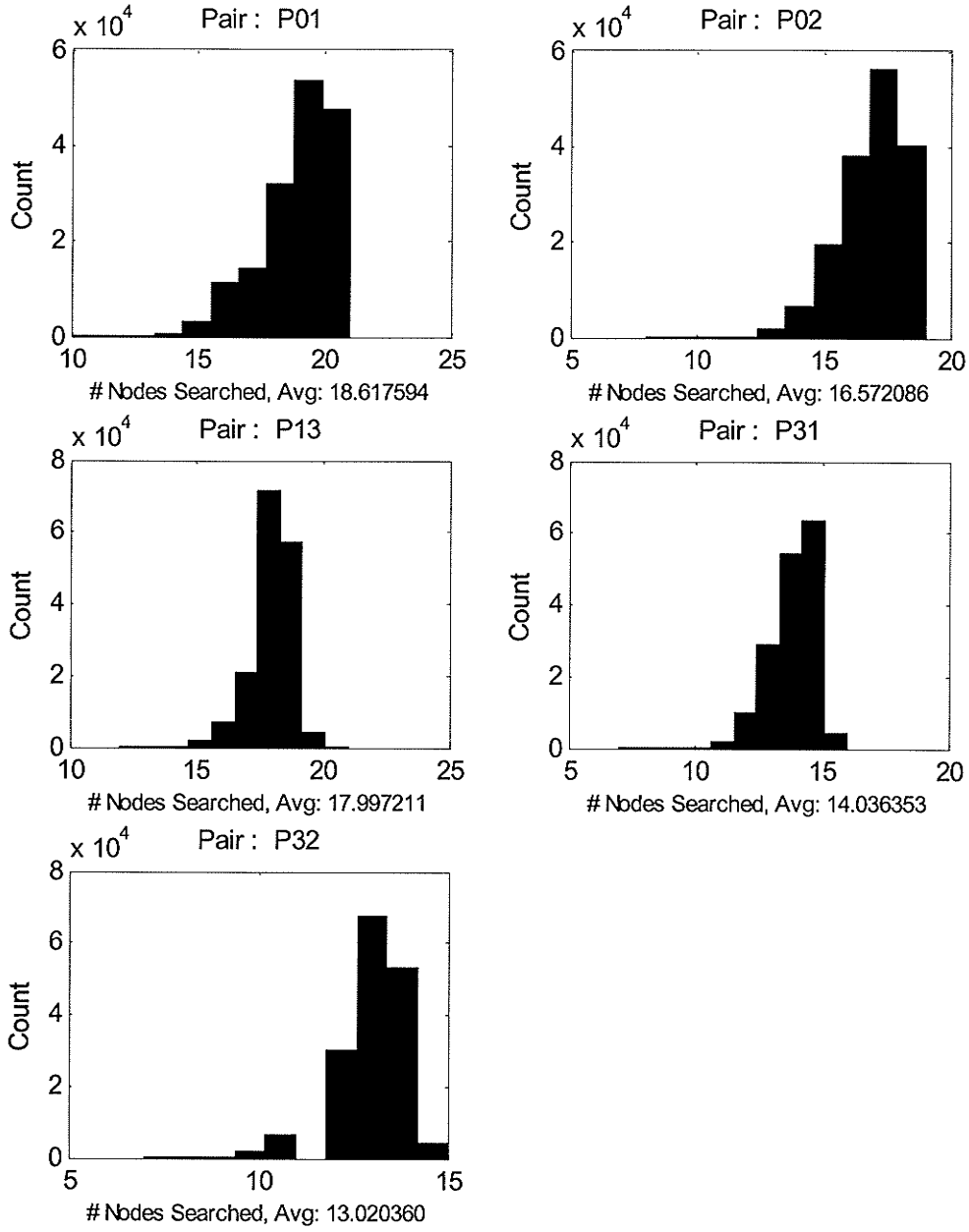


Figure B-2: Distribution of Outbound Nodes Searched for 16K Rules