Design and Verification of a System-on-Chip Packet Classification Implementation

by

Doug Cornelsen

A Thesis

Submitted to the Faculty of Graduate Studies

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, Ph.D.

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES

*****

COPYRIGHT PERMISSION

DESIGN AND VERIFICATION OF A SYSTEM-ON-CHIP PACKET CLASSIFICATION IMPLEMENTATION

BY

Doug Cornelsen

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 at 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. While many schemes have been proposed, to solve the multi-dimensional problem, none of them scale well beyond two dimensions in terms of speed and or rule size. As well, most schemes disregard updates, the ability to perform rule set modification, in order to increase throughput performance. This thesis presents a new scheme that decomposes the multi-dimensional problem into a set of 2-dimensional queries. Every 2-dimensional query can then be solved in parallel, each returning a set of possible solutions, which are intersected to find the best match. The 2-dimensional scheme is built upon a combination of prior research yet exhibits some unique features. Specifically, *Compressed Bit-Vectors (*CBV) a modification of the Lucent *Bit-Vector* (BV) [1] and *Aggregated Bit-Vector* (ABV) [2] schemes are introduced into a B-tree structure. The first dimension of the B-tree structure is split into four levels, referred to as buckets, into which rules are inserted based on rule characteristics. CBVs are stored in the second dimension of the B-tree structure and are returned as a result of a search. When a search is performed four CBVs are returned and then combined to produce one result. This thesis describes the verification and implementation of this new multi-dimensional approach using the *Canadian Microelectronics Corporation* (CMC) *Rapid Prototyping Platform* (RPP). An analysis of performance and scalability metrics obtained from extensive testing is provided along with a determination of *Application-Specific Integrated Circuit* (ASIC) implementation performance. The analysis shows the multi-dimensional scheme scales well with regard to memory usage and when implemented in an ASIC could sustain a *Gigabit Ethernet* (GE) line rate with packets of average size.

Acknowledgements

Many individuals and organizations were in involved in this thesis and deserve to be acknowledged. Firstly I would like to thank my advisor from the University of Manitoba Dr. R. D. McLeod for his patience and guidance throughout the course of the project. His involvement with the *Canadian Microelectronics Corporation* (CMC) proved invaluable in obtaining development hardware and resources for the project.

I would also like to thank the staff at TRLabs. In particular, Jeff Rohne and Jeff Diamond for the opportunity to work in such an excellent research environment. TRLabs also provided the project suggestion from lead industry sponsor PMC-Sierra. Additionally, I would like to thank both the *National Sciences and Engineering Research Council* (NSERC) and TRLabs for partnering and providing funding for this thesis. TRLabs also partnered with the University of Manitoba to provide the development tools which enabled the FPGA work performed in this thesis.

Without the support of CMC the implementation work performed in this thesis would certainly have not be at the level it is. CMC provided a *Rapid Prototyping Platform* (RPP), RLDRAM FPGA development board, ARM software development tools and RLDRAM controller intellectual property. In particular I would like to thank Hugh Pollitt-Smith for his assistance and involvement in this thesis.

Primarily, I would like to thank my research colleague Clint Stuart for his contributions to the design specifications and software-hardware integration. Clint provided many of the ideas and concepts for the thesis as well as designing the majority of the hardware and simulation environment.

I would also like to acknowledge the assistance provided by PMC-Sierra, Vansco Electronics and IDERS for this project. Finally, I would like thank my wife Jessica for her support and encouragement during my Masters research.

Table of Contents

List of Figures

List of Tables

List of Terms and Abbreviations

ABV	Aggregated Bit-Vector
AFS	ARM Firmware Suite
AHB	Advanced High-performance Bus
AMBA	Advanced Microcontroller Bus Architecture
APB	Advanced Peripheral Bus
API	Application Programming Interface
AQT	Area Based Quad Tree
ARM	Advanced RISC Machine
ASIC	Application Specific Integrated Circuit
BV	Bit-Vector
CBV	Compressed Bit-Vector
CMC	Canadian Microelectronics Corporation
cPCI	Compact PCI
DDR	Double Data Rate
DPR	Dual Port RAM
DMA	Direct Memory Access
EBI	External Bus Interface
EDIF	Electronic Design Interchange Format
FIFO	First In, First Out
FIS	Fat Inverted Segment
FPGA	Field Programmable Gate Array
Gbps	Gigabit per second
GE	Gigabit Ethernet
GEM	Geometric Efficient Matching
GOT	Grid of Tries
GUI	Graphical User Interface
HAL	Hardware Abstraction Layer
HW	Hardware
IANA	Internet Assigned Numbers Authority
ICE	In-Circuit Emulator
IP	Intellectual Property, Internet Protocol
IPv4	Internet Protocol Version 4

K	kilo binary (2^{10} or 1024)
LSB	Lease Significant Bit
LT	Logic Tile
LVDS	Low Voltage Differential Signaling
Mpps	Million Packets Per Second
MSB	Most Significant Bit
NAT	Network Address Translation
NSERC	National Sciences and Engineering Research Council
PC	Packet Classification
PCI	Peripheral Component Interconnect
PFAAE	Packet Filter Acceleration Assist Engine
PLD	Programmable Logic Device
QoS	Quality of Service
RAM	Random Access Memory
RLDRAM	Reduced Latency Dynamic Random-Access Memory
RPP	Rapid Prototyping Platform
RTOS	Real Timer Operating System
SerDes	Serializer/Deserializer
SSRAM	Synchronous Static Random Access Memory
SoC	System-on-Chip
SW	Software
Tcl	Tool Command Language
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
VHDL	VHSIC Hardware Description Language
VHSIC	Very-High-Speed Integrated Circuit
ZBT	Zero Bus Turnaround
µs	Microsecond

1 Introduction

The growth of the Internet has created a huge commercial market, which feeds competition between service providers. As user demands and expectations continue to rise so does the motivation for multi-dimensional packet classification. This type of classification is necessary to provide differentiated services and more QoS. A firewall provides a good example of a network component providing a differentiated service. It consists of a set of rules that apply to information usually extracted from packet header fields. The directive associated with each rule will result in a packet being accepted or denied. In addition, the state of a flow can be monitored from the packet header fields. This, combined with state information already maintained by the firewall allows flow control. For example, a firewall could accept TCP packets with *synchronization* (SYN) set only as part of TCP connection initiation or allow UDP packets through only if they are responses to outgoing UDP packets.

Quality of Service (QoS) routes provides another application for multi-dimensional packet classification. Traditional routing only uses the destination address to determine a destination, while QoS also routes and switches using *Internet Protocol Version 4* (IPv4) layer four fields. Based on IPv4 layer four information preferential treatment can be given to certain traffic while others may be denied.

Research Motivation

Current packet classification schemes are optimal for one or more aspects of packet classification and require tradeoffs between search speed, update performance and storage requirements. Typically, search speed is the first priority followed by storage requirements and update performance. While this methodology has worked in the past, new multi-dimensional packet classification capable of flow identification and state maintenance requires more emphasis to be placed on updating. An adaptable packet classifier is capable of performing intelligent operations that its static predecessor is not. As a result, one of the goals of this thesis is to find a more balanced solution to the multi-dimensional packet classification problem that is scalable and capable of updates while making only a modest sacrifice in search speed.

It is desired to develop a packet classifier scheme with the following characteristics:

- Scalable to large rule-sets (64K) and multiple dimensions (6 and beyond)
- Achieves good update time, storage and query time characteristics
- Works in worst-case conditions (small packet sizes)

Objective

The objective of this thesis is to verify a novel multi-dimensional packet classification implementation using the *Canadian Microelectronics Corporation* (CMC) *Rapid Prototyping Platform* (RPP). The implementation presented is based on knowledge gained from the review of state of the art algorithms and from direction made by TRLabs lead industry project sponsor PMC-Sierra. The architecture developed is to be fast, scalable, preserve the flexibility provided by software programmability and be capable of updates. As such, the project aim is to determine a balance between hardware optimization and programmable flexibility with the focus on classification services useful in a network edge device. Prototyping is used as a means to provide a baseline for determining expected performance when implemented in an ASIC. Conclusions are made with regards to the architecture designed to meet the project objectives and on the verification flow.

1.1 Report Structure

This thesis is divided into nine chapters covering the design and verification of a *System-On-Chip* (SoC) packet classification implementation. In Chapter 2, background is provided on a number of topics. First the basic packet classification problem is formulated. Next, a set of optimal design criteria is developed providing design considerations for the evaluation of algorithms. After the considerations are presented an outline of past literature and packet classification approaches is given with regards to the most suitable research for this thesis. Chapter 3 builds upon this information to outline a design specification, including an example illustrating typical operation, and justification for design decisions made. This is followed by a description of the development platform and design flow in Chapter 4. Chapter 5 provides a description of the hardware and software components designed and implemented in this thesis. In Chapter 6 the developed verification flow is shown, illustrating the steps taken to ensure proper operation and implementation of the design specification. The result of performance testing is outlined in Chapter 7, followed by recommendations for future work in Chapter 8 and conclusions in Chapter 9.

2 Background Information

2.1 Classification Problem

The basic packet classification problem is well documented and described in [3] and [4] as follows:

- A network element maintains a database of n rules for processing incoming packets. Each rule consists of a filter and has an associated action.
- Each filter has k fields corresponding to the fields in packet headers, which it should match. Each header field is assigned one of the three match types: exact match, prefix match, and range match.
- If more than one filter matches an incoming packet, the tie is broken by using a priority value assigned to each filter.
- 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.

2.2 Matching Styles

Exact Matching: For exact matching the header field of the packet must exactly match the corresponding rule or filter field. This type of matching may be associated with fields like the protocol field or TCP flag field for IPv4.

Prefix Matching: In prefix matching the rule field should be a prefix of the corresponding header field. This type of matching is amenable to IP source and destination addresses.

Range Matching: In a range match the header field must fall in a specific range outlined by the corresponding rule field. This type of matching can be used for matching up port numbers in ranges.

2.3 Design Considerations

The design considerations outlined in this section have been presented in the extensive collection of papers and address the criteria for efficient packet classification [1] [5] [6] [7].

1. ***Throughput:*** Internet Service providers are building networks with link capacities of 1 and 10 Gigabits and are envisaged to exceed 40 Gigabits/s [2]. Ideally an algorithm should be fast enough for use with these networks. This requires packet classification

throughput on the order of 1.49 million packets per second and up. Equation 2-1 shows how this number is derived.

Equation 2-1 : Gigabit-Ethernet Throughput Calculation

Wire speed : 1 Gbps
Smallest Packet Size : 64 bytes
Interframe gap : 12 bytes
Preamble : 8 bytes

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

2. **Worst-case vs. Average-case:** There is a widely held view that for access time performance of packet classification, one must focus on worst-case rather than average-case. An algorithm should have small worst-case execution times which are independent of traffic patterns.

3. **Fields:** It is uncertain which headers or fields should be used to provide next generation services. For the purpose of developing an algorithm however, it is convenient to exploit layer three and four fields from IPv4 packets. Table 2-1 shows typical IPv4 headers used for packet classification.

Table 2-1 : Typical Fields of Interest for IPv4 Packets

Layer Three Header Fields		Layer four Header Fields	
Source IP Address	(32 bits)	TCP and UDP Source port Numbers	(16 bits)
Destination IP Address	(32 bits)	TCP and UDP Destination port Numbers	(16 bits)
Protocol Field	(8 bits)	TCP flags	(8 bits)
Type of Service	(8 bits)		

4. ***Number of rules to be supported:*** Rule databases are growing and are predicted to increase to several million rules. In the past packet classification was used for security and firewalling 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 may 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 [7].

5. ***Nature of rules:*** Table 2-2 illustrates a few simple filter rules. Current routers use rules with prefix masks on destination IP addresses however, more general masks such as arbitrary ranges can also be used.

Table 2-2 : Example Rules (* indicates a wildcard)

Rule	Source Address	Destination Address	Protocol	Source Port	Destination Port
A	128.121.*.*	ANY	TCP	< 321	28 – 90
B	196.134.2.45	192.96.*.*	TCP	34	< 300
C	128.*.*.*	192.165.2.*	UDP	*	*

6. ***Updating the set of rules (adaptive):*** The number of changes to the rules depends on the application of the packet filter. Changes can occur as a result of a policy change or in stateful packet filtering when a new flow is inserted or deleted. To achieve this, an algorithm needs to perform inserts and deletes in times of 10-100 μs [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 or reduce the 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 pre-computation scheme explicitly attempts to optimize the update rates [5].

8. ***Priority:*** It is possible that some packets may match more than one rule. The rule must allow for priorities to be imposed on these rules, so that only one of these will finally be applicable to the packet (i.e. allows one to distinguish the lowest cost filter).

9. ***Hardware Implementation:*** For operation at very high-speed an algorithm must be amenable to hardware implementation. The algorithm structure should seek to take advantage of hardware parallelism and pipelining.

10. ***Memory Accesses:*** Memory accesses should be minimized since they are the main bottleneck to performance [1]. Memory accesses are the bottleneck because the time required to retrieve data from memory is much greater than the operating frequency of most processors.

11. ***Storage Requirements:*** The algorithm should achieve the required target access speed while minimizing the amount of memory used. In order for the algorithm to scale there must not be a memory explosion. An algorithm with memory usage which scales with a rate of $O(n^k)$, where k is the number of dimensions, is be considered to have explosive growth. Ideally the memory requirements should be linear, $O(n)$.

2.3.1 Performance Targets

Given the eleven design considerations previously outlined, Table 2-3 outlines performance goals for packet filters. It should be noted the targets outlined provide ideal goals for the major areas of packet filtering and do not consider the design trade-offs which usually occur. Typically, memory usage and the time required for updates are traded off with packet throughput. This thesis seeks to develop an algorithm which balances memory usage, update performance, expandability to multiple fields and throughput.

Table 2-3 : Performance Targets

Device target location	network edge
Number of rules	128K rules
Packets/sec performance	2 million
Update time	10-100 microsecond
Number of filter fields	6

2.4 Previous Work

The focus of the background material presented in this thesis is on software and software/hardware hybrid approaches for packet classification. As one of the main goals of this thesis is to evaluate and develop a solution that maintains the flexibility of a software solution this seemed appropriate. The first section of this chapter covers software based tree structures and the second section covers hardware based bit-vectoring approaches.

2.4.1 Software Tree Structures

The development of various types of search tree structures has long been a very important part of packet classification research. This section provides an overview of five search tree structures found in research literature: Fat Inverted Segment Tree [4], Area Based Quad Tree [3], Geometric Efficient Matching Algorithm [8], B-tree [9] and Grid of Tries [3].

Fat Inverted Segment Tree [4]

The Fat Inverted Segment (FIS) tree was developed to solve the 2-dimensional packet classification problem. Like other approaches FIS views the PC problem in geometric terms. Each rule is represented by a rectangle on a 2-dimensional grid with a specific cost. Preprocessing is done so that when a search is performed for a point the rule, or rectangle, with the lowest cost is returned. The FIS tree is described as follows in [4]:

> "The FIS tree is a balanced, inverted t-ary tree with l levels. Each node v has a pointer to its parent *parent(v)* and at most t incoming arcs. The leaves of the FIS tree correspond to the elementary intervals in order. An internal node v corresponds to the larger interval that is the union of the elementary intervals stored at its leaves."

Elementary intervals are simply the set of non overlapping intervals created when the rule ranges are projected in a given dimension. Given n rules it is possible to have up to $2n+1$ elementary intervals. To build a 2-dimensional FIS tree the projections of the rectangles must be considered in both the x and y axis. Figure 2-1 illustrates an example of the x projections for a FIS tree. In the figure the colored rectangles represent rules and the dashed lines represent the elementary intervals for the x axis. A search is performed by first finding the elementary interval at the top level of nodes which corresponds to the search point. At this time a second dimension FIS tree, pointed to by the node found in the first dimension, is searched. This second dimension FIS tree is built using the y projections of the rules found at the first dimension node. While the nodes at the top-level represent all of the elementary intervals not all of the rules are contained at these

nodes. The first dimension parent nodes, shown below the top-level, correspond to larger intervals which are the union of its children nodes. In effect the top-level nodes provide a guide for the appropriate parent nodes to search. Once all levels of the FIS structure have been searched the rule found with the lowest cost is selected and the appropriate action is taken. The FIS structure is reported to scale with complexity $O\left(n^{1+1/l}\right)$ and require approximately $(1+l)\log w$ memory accesses for a search (w indicates width of search field) [4]. While FIS does exhibit excellent characteristics with regard to structure size and search performance its primary limitation is update performance. While [4] does suggest some methods to allow improve update performance FIS is not ideally suited for a dynamic environment requiring updates.

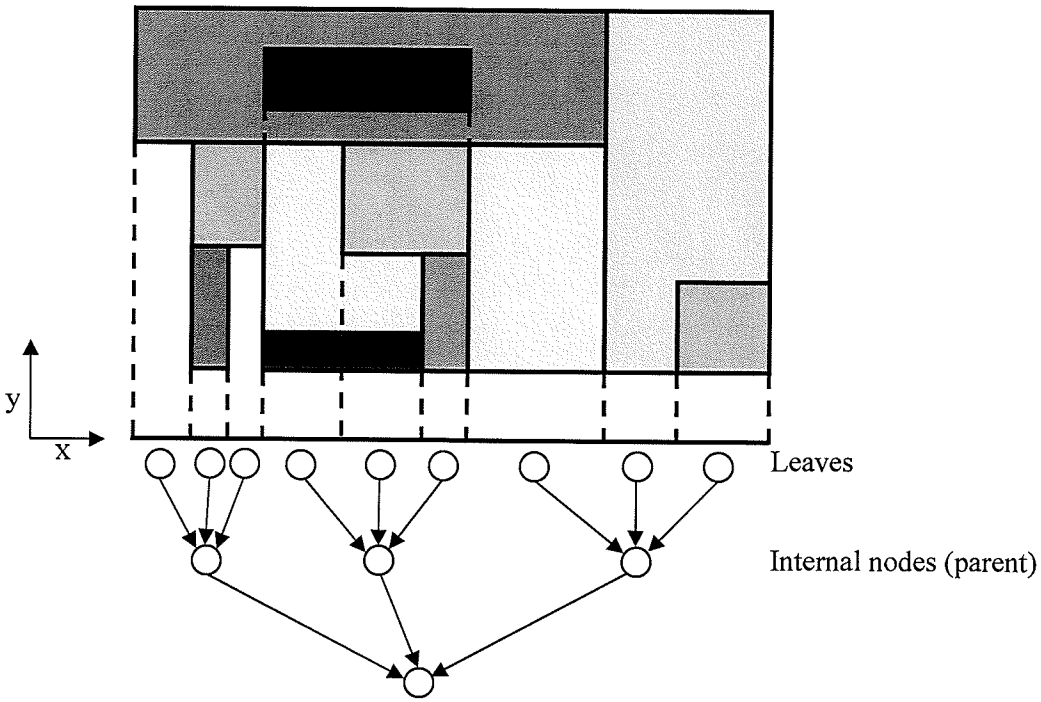

Figure 2-1 : x- FIS Tree

Area Based Quad Tree [3]

Like FIS, *Area Based Quad Tree* (AQT) views the PC problem in geometric terms. In this case a tree is implemented in which each node has four children to represent a hierarchically decomposed search space. As each node has four children the tree is referred to as a quad tree. Each child node represents one of the four squares obtained by dividing the parent square into four equal sub-squares. When used in a two dimensional scheme the quad-tree is referred to as an

Area-based Quad Tree. Every node in an AQT at depth h has a square of size $2^{32-h} \times 2^{32-h}$ associated with it. Worst-case run time for AQT search is $O(w \log n)$ were w is the width of the search field and n is the number of rules. The structure scales with a worst-case complexity of $O(n^2)$.

Geometric Efficient Matching Algorithm [8]

As its name implies the *Geometric Efficient Matching* (GEM) takes a geometric approach to solving the PC problem. However, unlike FIS and AQT, GEM uses more than two dimensions for implementation and testing. When using d dimensions each field of a rule defines a particular dimension of a d-dimensional hyper rectangle. These hyper rectangles may overlap and must be organized into non-overlapping hyper-rectangles so a search can be performed. Searching the developed geometric structure is done in logarithmic time and exhibits a worst-case space complexity of $O(n^d)$ for a rule-set with n rules. As the search is performed in logarithmic time per dimension the search complexity of the algorithm is $O(d \log n)$. Testing indicates when using more than two dimensions field order has a large effect on structure size. In particular, it is illustrated that with large rule sets and the proper field order a performance of over 1 million packets per second can be maintained. Also introduced in the paper is the concept of a space time trade off in which a rule-set is split into l groups of size n/l. By splitting the rule-set into groups l search structures are built each of which need to be searched to produce a final result. As a result, the search complexity becomes $O(ld \log n/l)$ and the worst-case space complexity becomes $O(n^d/l^{d-1})$.

B-tree

The B-tree is common tree structure used typically for I/O operations. Background of the B-tree is provided because it is used in many systems and is implemented in this thesis. The basic element of a B-tree is a node containing a set of keys arranged in ascending order and a set of pointers to link the nodes of the tree together. A node contains x keys has x+1 pointers to connect to other nodes. Additionally, a node contains a count of the number of keys present and a flag indicating if it is a leaf. Figure 2-2 provides an example of a node. A standard way of referring to a B-tree is t, the minimum number of keys a node can have. This is also known as the minimum degree of the tree and is always greater or equal to two. With regard to t, a B-tree

follows two basic parameters:

- Every node other than the root has a minimum of t-1 keys.
- Every node can have a maximum of $2t$-1 keys.

The maximum height h of an n key B-tree is found using the following formula [9]:

Equation 2-2: Maximum B-tree Height

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

Like the GEM structure a B-tree has worst-case space complexity of $O(n^d)$ and search complexity of $O(d \log n)$ for d dimensions. More details on the B-tree can be found in [9] including build and search operations.

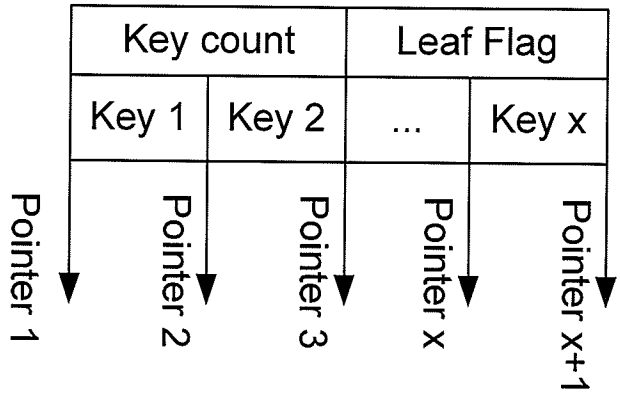

Figure 2-2 : Example B-tree Node

Grid-of-Tries [3]

The *Grid-of-Tries* (GOT) algorithm performs a basic extension of the standard trie structure to extend it for 2-dimensional rule matching. In a typical single dimension trie, used for prefix matching, the position of the node shows the corresponding key rather than the key being stored at the node. When used in an IP router destination addresses are used to traverse the trie structure to find the longest matching prefix.

> "Grid-of-Tries extends a basic trie to two dimensions, by maintaining two tries – a trie for destination address and a trie for source address in the packet. Each node in the destination trie, instead of storing a rule, now points to a relevant source trie, and each node of the source trie contains a rule that matches the appropriate destination and source prefix pair. [3]"

GOT has worst-case search complexity of $O(2w)$ and worse case space complexity of $O(n)$, where w is the width of the field and n is the number of rules. While these complexities are low it should be noted this structure only works for prefix matching and not range matching.

2.4.2 Hardware Based Bit-Vector Schemes

Lucent Bit-Vector Scheme [1]

This algorithm uses a divide and conquer approach in which a d dimensional problem is separated in d one dimensional problems. A binary search is used to find a result for each one dimensional problem with $O(\log n)$ search complexity. The results are then combined by performing AND operations on resultant bit-vectors from all dimensions as seen in Figure 2-3. This scheme is hardware oriented and requires the use of large buses (1000 bits wide). It is shown in [1] that the bit-vector intersection step requires examining each of the rules at least once thus requires $O(n)$ execution time, where n is the number of bits in the bit-vector. The use of bit level parallelism does accelerate the execution time but is only reasonable for small rule-sets [1]. The space requirements for this algorithm are $O(n^2)$ where n is the number of rules.

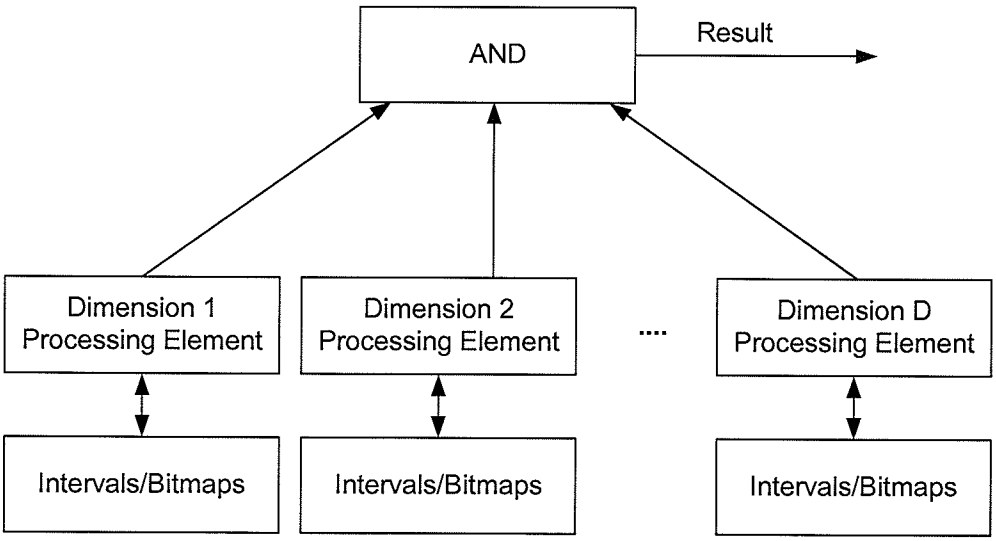

Figure 2-3 : Parallel Implementation

Bit-Vector intersection Problem

Given a set of l binary vectors $v_1,.....v_l$ each of length n , 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 previous research has been performed in the area of two dimensional schemes, particularly for routing and IP forwarding, where the IP source and destination address are the two fields of interest. Figure 2-4 provides a simple example of matching up 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 2-4 : Bit-Vector Example

With all applicable rules found, the one with the highest priority will determine which one is applied. The real problem is how to find the common pairs or elements in a timely and efficient manner. Lucent Bit-Vector [1] uses bit-level parallelism to solve this problem while the Aggregated Bit-Vector [2] uses aggregation.

Aggregated Bit-Vector [2]

The *Aggregated Bit-Vector* (ABV) scheme is based on the bit-vector scheme (BV) described previously. It makes two distinct contributions, the recursive aggregation of bit maps and filter rearrangement. The paper [2] suggests it takes logarithmic time for many databases. Aggregation is used to reduce the memory accesses, based on the assumption that the number of set rules in the bit-vector will be very sparse.

In this scheme, each bit-vector is represented by an aggregate BV with word size A. Each bit in this vector then represents n/A elements in the bit-vector where n is the number of bits in the bit-vector. If nothing is set in a range it contains a zero. The ABV is then the OR of the corresponding bits in the BV. This process can be repeated at multiple levels. The goal of this system is to effectively construct the bit map intersection without looking at all of the leaf bit map values for each field. This allows one to quickly filter out bit positions where there is no match.

Rearrangement is used to localize matches creating sparser matches in the ABV. However, vectors must be stored to retain the mapping of priorities. The time required for an insertion or deletion of a rule is similar to the BV scheme. This is because the ABV is updated each time the associated bit-vector is updated. The updates can be expensive because adding a filter can potentially change the bit maps of several nodes.

2.4.3 Summary

Overall the background material illustrates no particular tree structure is ideally suited for multi-dimensional packet classification. In particular, most schemes are suited for 2-dimensional classification and do not scale well beyond this limit. The one distinction is GEM but it has worst-case size complexity of $O(n^d)$. While the hardware bit-vectoring schemes scale well to a large number of dimensions they do not scale well in terms of the number of rules. As the number of rules grow the memory required for bit-vector storage becomes prohibitive.

3 Design Specification

3.1 Overview

When producing a system design specification it is important to first identify the primary goal. As outlined in the introduction the primary goal of this thesis is to implement and verify a fast, flexible, scalable and novel approach to multi-dimensional packet classification problem with the CMC RPP. The specification outlined in this section strives to meet this goal while taking into account the design considerations specified in Section 2.3. Like most thesis projects compromises are made with respect to implementation complexity and time availability. In particular, a representative implementation based on the RPP is outlined with the end goal of determining performance scalability based on an ASIC.

After a review of previous work in the field it is clear a fast, flexible and scalable method for a completely arbitrary multi-dimensional classifier is difficult to find. It is also clear that no one search scheme can meet all the considerations to produce an ideal packet classification design. Compromises need to be made based of the target environment and the design goals. The result is a design which combines several techniques to achieve balance with regard to the design considerations and the major goal of implementation and verification using the available hardware.

It is proposed to break the typical multi-dimensional packet classification problem into multiple 2-dimensional problems and then perform an intersection operation on the solution sets. This decision is made for two main reasons. Firstly, 2-dimensional schemes have been known to scale well, with respect to speed and memory usage, to a large number of rules. However, beyond two dimensions packet classification does not scale well. Potentially one, two or three different 2-dimensional schemes can be used in a typically system. For example, something like a *Grid-of-Tries* (GOT) in paper [3] can be used for the source and destination IP addresses where both fields contain prefixes. A different scheme can then be used where ranges are important such as in the port addresses field. It is somewhat intuitive to use different forms of classifiers for the three forms of matching Exact, Range and Prefix. Secondly, when coupled with a group of software based 2-dimensional schemes, hardware based intersection has the potential to provide for a high performance, flexible and scalable solution.

3.1.1 2-Dimensional Software Search

No speculation is made as to which services should be provided or as to which headers or fields should be used to provide the services of the future. Effort is focused on a more general problem, how to find the appropriate rules, having fields either solely or a mixture of exact, prefix or range matches, from a multi-dimension rule-set. As a goal is to provide a generic solution with a balance of speed and update performance a B-tree structure is used for the software search. The B-tree structure is an attractive solution because unlike many other 2-dimensional search structures it supports updates. Many algorithms, like FIS [4], sacrifice updates to benefit speed and require a large amount of pre-computation to build the search structure. As inserts and deletes are required for a system which could potentially be used for stateful packet classification the B-tree is seen as good solution for the 2-dimensional software search structure. As well, a B-tree structure has tunable performance, by adjusting the size of the B-tree node the height of the tree can be tuned to change the worst-case number of nodes accessed during a search. It should be noted a B-tree with a t value of 3 has been selected for use in this thesis. This means each node has room for five keys and six pointers.

Another important feature of the B-tree is its performance in a system with a large cache line. As mentioned earlier in section 2.4.1 a B-tree is made up of nodes containing a number of keys examined to find the desired search point. If the search point is not found at a node then the keys are used as guides to determine which pointer should be followed to the subsequent search node. A B-tree node will typically be built as a data structure found in a continuous space in memory. This is as opposed to finding all of the keys spread out throughout the system memory. A significant performance gain is made when data structure access can be made with one or more burst operations into a high-speed memory like a cache. The processor is then able access the data structure, with little latency, from the cache.

3.1.2 Hardware Based Intersection

The solutions from each of the 2-dimensional searches then produce a pointer to a bit-vector stored in a high-speed memory. As expected the bit-vectors contain a representation of the rules found to match a particular search. At first glance a scheme based on bit-vectors may not seem to be the optimal choice because of large storage requirements and lengthy intersection operations. The prohibitive storage requirements illustrated in [1] and [2] show both Lucent BV and ABV require large amounts of memory for bit-vector storage. For bit-vectoring to be used successfully some method of compression needs to be used to mitigate these large storage requirements. Of course, for compression to be attractive the bit-vectors must have the appropriate properties. The

primary reason a compressed bit-vectoring scheme is chosen is because research [2] has shown large rules sets typically only contain a handful of rules which match each incoming packet.

The compression scheme selected to achieve a reduction in memory usage is known as hierarchical compression. Hierarchical compression employs a tree like structure to compress the full bit-vector into a smaller representation. Figure 3-1 illustrates the concept by providing a small example. The top of Figure 3-1 shows the complete bit-vector in its entirety with the colored items indicating the bits found in the compressed version. Each bit location one level below the complete bit-vector represents a number of bits above. If nothing is set in the range of bits represented the bit location contains a '0'. However, if a bit is set the bit location contains a '1'. When the bit-vectors are sparse a memory savings is obtained because portions containing all '0's need not be stored because they contain no information. The process of bit representation is continued to the bottom level of the structure in which each bit represents three bits at a lower level. It is easy to see this type of compression works well for sparse bit-vectors and provides rapid identification of only the necessary portions of the bit-vector to check for intersection. The number of levels chosen for implementation in this thesis is three with each bit at a lower level representing 32 bits. This allows for bit-vectors of length 32768 to be represented using this scheme.

The next issue to address is the large intersection operations possible with large bit-vectors. The ABV scheme seeks to exploit the sparse characteristic of bit-vectors by using aggregation to reduce the intersection operation as compared to Lucent BV. While the aggregation does allow ABV to perform an intersection operation on only a fraction of the bit-vector it still requires storage of all complete bit-vectors. A compressed bit-vector scheme allows both the benefits of reduced storage and intersection time at the cost of having to perform compression operations. It is projected the memory and intersection time savings will out weigh the cost of the compression operations. If the compressed bit-vectors are sparse enough then the intersection and compression operations can be done at extremely fast rates. Additionally, compressed bit-vector solutions can be combined using simple hardware and pipelined with the 2-dimensional software searches. This allows as much time to perform the final intersection operation on the compressed bit-vector results as is used for the 2-dimensional search done in software. As such a goal of an implementation taking advantage of pipelining is achieved.

Furthermore, using a solution based on 2-dimensional search structure takes the next logical step as bit-vectors should be even more sparse then produced by a 1-dimensional search structure. As the bit-vectors will likely be sparser, the system should have greater performance. It should

be noted the performance of any packet filter system is mostly determined by the characteristics of the rule-set. In particular, the system specified is believed to work best with rule-sets which produced resultant 2-dimensional searches having sparse bit-vectors.

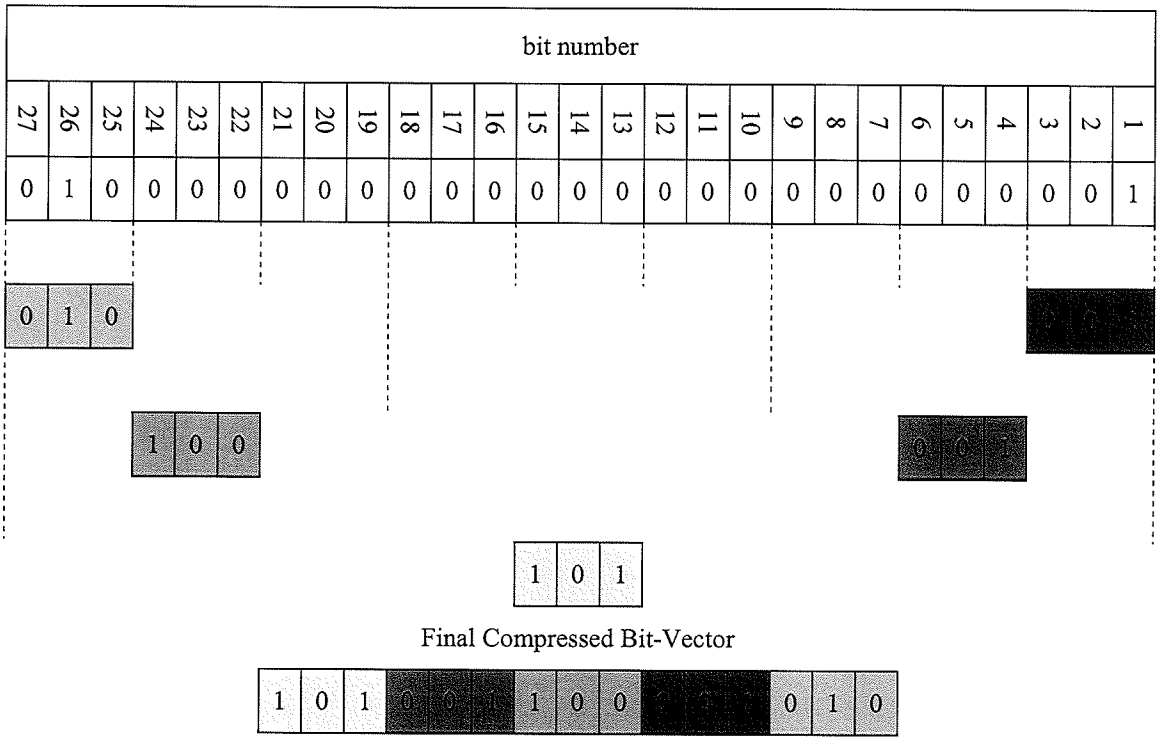

Figure 3-1 : Hierarchical Compression Example

3.1.3 Bucketing Enhancement

To further enhance the insert and delete potential of the system an additional approach for breaking the rule-sets into levels is introduced. Levels can be thought of as buckets that rules are put into as the 2-dimensional search structure is built. Each level corresponds to one of four range sizes of a field. For example, level one is defined as a field having a range width of less 2^8 in the first dimension. Likewise, level two is defined as a field having a range greater or equal to 2^8 and less than 2^{16}. In this way each of the four levels corresponded to a different size of ranges. Effectively this creates four different B-trees each filled with ranges corresponding to the size specified for their level. It is clear to see in a rule-set with an equal distribution of range sizes four B-trees will be built in the first dimension instead of just one. This type of implementation reduces the number of overlapping ranges as large ranges are never combined in the same tree with small ranges. The reduction of overlapping ranges also makes insertion and

deletion of a rule quicker because the possible ranges to combine or split is also reduced. As well, because the rules are split between four levels the CBVs will be sparser and easier to update. The approach of splitting into levels does improve the insertion and deletion speed it also reduces the software search speed. The search speed is lower because four trees have to be searched instead of one. Although each of the new trees is smaller, the end effect is a longer search as shown by using Equation 2-2. The worst-case number of nodes, before implementing bucketing, is the height of the tree, $h \leq \log_t \frac{n+1}{2}$. After using four buckets the sum of the heights of the new trees is $h \leq 4\log_t \frac{n+1}{8}$. Using a value of 3 for t and 100 for n the height without bucketing is 3.56 and the combined height of the four trees after bucketing is 9.23. As well, each level of B-tree searched produces a pointer to a compressed bit-vector which needs to be combined with the others to create a complete result. Unlike the final results from the 2-dimensional searches, which are ANDed together to find matches across dimensions, the results from the levels are ORed together. This is because each level produces one set of the possible matching rules. In summary, this method basically trades some of the search speed in order to improve the update capabilities.

3.2 Build Sequence

To provide further insight into the design implementation the sequence of operations used to build the software and hardware based search structures is described. The sequence is described at a high level to provide a basic level of understanding with lower level details provided in subsequent sections.

It is assumed n rules are available; each containing four fields corresponding to source IP address, destination IP address, source port and destination port. The IP address portions of the rules are of prefix match type and the ports are of range match type. Of the four dimensions specified in the rules two are selected for one of the 2-dimensional B-trees. The other two fields are selected for the second 2-dimensional B-trees. As mentioned previously each 2-dimensional B-tree is broken up into four levels based on range size to improve update performance.

The structure for the multi-level 2-dimensional B-tree is first created in software so rules can be inserted. When a rule is processed for insertion the field corresponding to the first dimension of the B-tree is evaluated to determine which level the field should be inserted into. Once the level has been determined the field is inserted into the first dimension of the B-tree. Unlike a

typical B-tree insert which would insert a number this B-tree insert actually inserts a range. Each key in the B-tree node corresponds to a non overlapping range known as an elementary interval. An elementary interval is simply a subspace of the complete rule space of which one or more rules are part. For example, the IP rule space has values from 0 to 2^{32}-1, an example elementary interval is from 1 to 3. This elementary interval can be the result of a single rule or the intersection of one or more rules. As new rules are added to the B-tree they may overlap, intersect, or define new elementary intervals in the B-tree structure. The process of inserting a new rule is recursive in nature as intersections and overlaps must be handled until the entire range of the rule has been specified. When an elementary interval, of a B-tree from the first dimension, is added or matched to a rule the second dimension must be processed as well. Each B-tree elementary interval in the first dimension has a corresponding second dimension B-tree. This second dimension B-tree has exactly the same structure as the first dimension but its elementary intervals are based on the field specified for the second dimension.

Unlike the first dimension B-tree the elementary intervals of the second dimension have a corresponding B-tree of rule identifications. This B-tree holds all of the identifications for rules which match the first and second dimension elementary intervals. As a rule may contain a range which covers more than one elementary interval in the first or second dimensions its rule identification may be found in multiple rule identification B-trees.

As each rule is inserted into the developing B-tree structure more elementary intervals are created and the structures grows in size. Once all of the rules have been processed and inserted into the structure the rule identification B-trees are converted into CBVs by a *Packet Filter Acceleration Assist Engine* (PFAAE) contained in an FPGA. An ordered list of rules is provided, from the rule identification B-tree, to the PFAAE to create a CBV and store it in a high-speed memory. The PFAAE provides a pointer back to the software to insert into the location of the rule identification B-tree. At this point the memory for the rule identification B-tree is freed because it is no longer required. The choice to build the complete B-tree structure and then build the compressed bit-vectors is done to simplify implementation. A system build for actual use would build the bit-vectors dynamically as rules are inserted and deleted. This approach, while being more flexible, was considered too complex of an implementation to fit within the time and hardware constraints of the project.

In a typical 4-dimensional search both 2-dimensional multi-level B-tree structures would be built on separate processors each having their own PFAAE for compressed bit-vector creation.

Unfortunately due to hardware availability issues only one processor was available providing for only one half of the implementation. The resulting build operations for the implementation presented in the rest of this thesis create only a 2-dimensional multi-level B-tree and the corresponding compressed bit-vectors. To achieve the results that would be obtained from having a full hardware system the available hardware is used to build all possible 2-dimensional field pairs. In this way all the build information and results are obtained even through the ideal hardware setup is not available.

3.3 Search Example

The following is an example of the search operation performed after the B-tree search structures and compressed bit-vectors have been created. The example provided illustrates a basic set of rules, an example incoming packet, its search through the B-tree structure, the data flow between hardware and software and the finally the solution. Table 3-1 shows the simple rule-set used for the example. This example employs only two levels of B-trees, the first is used for first dimensional range widths between 1 and 2 and the second level for those range widths greater than 2. Range width is defined as the number obtained when the start value of the range is subtracted from the end value.

To illustrate a search operation it is assumed a packet has arrived with a value of 4 for the first dimension and value of 32 for the second dimension. Figure 3-2 shows a partially constructed first level, first and second dimension B-tree. The B-tree nodes illustrated are capable of holding three keys and have four pointers to connect to subsequent nodes. The search is performed as follows:

1. The root node of the B-tree is searched for the point 4. The three elementary intervals found, contained in each node key, are searched from left to right until one is found that contains 4 or is greater than 4. The elementary interval eleven to twelve is found first and since it is greater than 4 it is known to access the node left of the root as it contains elementary intervals less than eleven. If the pointer would have been a NULL it would be known no elementary intervals were created for the value of three and the search could stop. In this case a message would be passed to the hardware OR operation to indicate the first level returned a NULL pointer.
2. The node below and to the left of the root node is searched next to determine if 4

can be found. As before the elementary intervals of this node are searched from left to right to find an interval which contains 4. The search reveals a match from the elementary interval from 3 to 4.

3. The second dimension B-tree is now searched to find the value of 32. Like in the first dimension search the B-tree is searched from root to leaf to find the value. The second dimension B-tree contains only one node which is searched to find the elementary interval from 31 to 32. This elementary interval contains a pointer to a compressed bit-vector which contains the rules R2 and R27.

4. The pointer contained at the second dimension elementary interval is passed to the PFAAE which retrieves the compressed bit-vector shown in Figure 3-3.

Table 3-1 : Example Rules

Rule ID	Rule Range Dimension 1	Rule Range Dimension 2
R1	1-2	33-34
R2	3-4	31-32
R3	5-6	29-30
R4	7-8	27-28
R5	9-10	25-26
R6	11-12	23-24
R7	13-14	21-22
R8	1-8	1-16
R9	1-8	18-34
R10	1-8	33-34
R11	9-16	1-2
R12	17-25	5-12
R13	26-34	3-9
:	:	:
R27	1-4	1-34

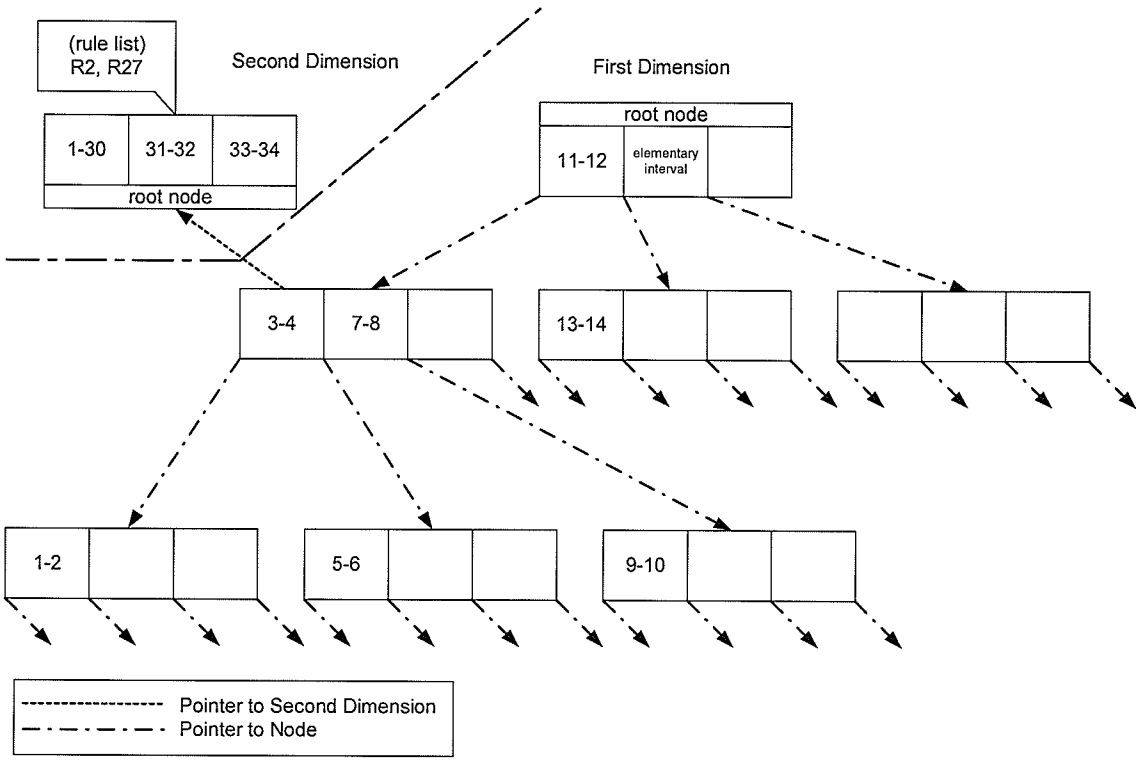

Figure 3-2 : First Level B-tree Search Example

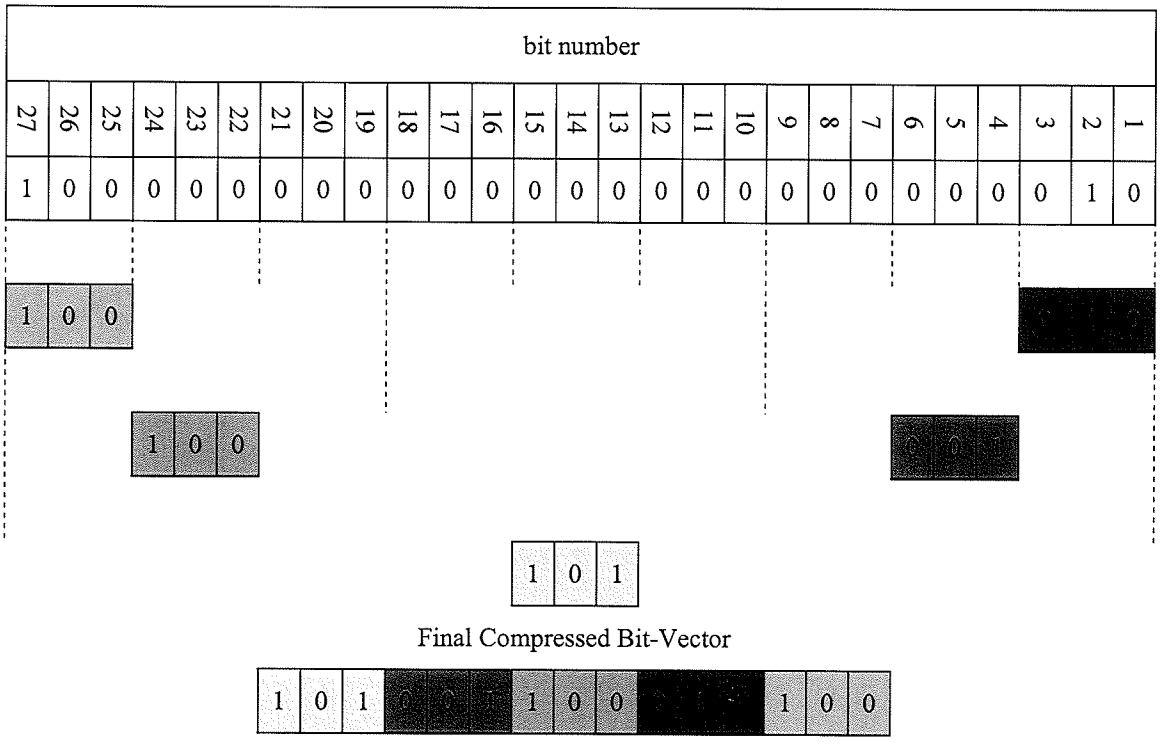

Figure 3-3 : First Level B-tree CBV Search Result

5. The exact same sequence is performed on the second level B-tree shown in Figure 3-4. This search also passes a pointer to the PFAAE for which it retrieves the compressed bit-vector shown in Figure 3-5.

6. Once the PFAAE retrieves both compressed bit-vectors an OR operation is performed to obtain the complete solution. The complete solution is shown in Figure 3-6. Should the PFAAE have been passed an indication that no elementary interval was found the OR operation would not have needed to be performed as the solution would have been the single retrieved bit-vector. Likewise, if an indication was provided that both searches could not find a matching elementary interval no bit-vectors would be retrieved and there would be no resultant match. Depending on the policy of the packet filter this would likely result in an action to deny the packet.

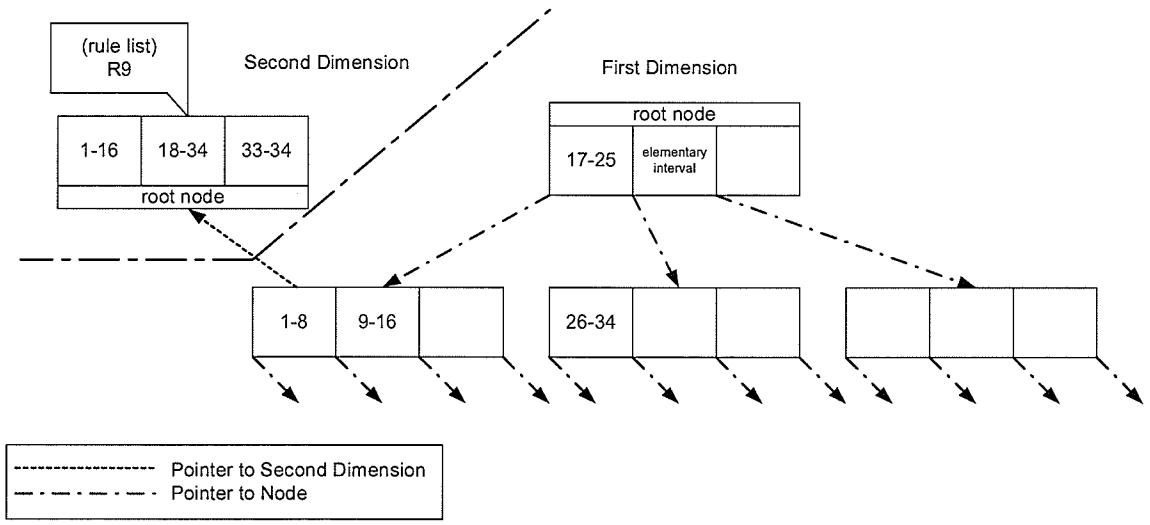

Figure 3-4 : Second Level B-tree Search Example

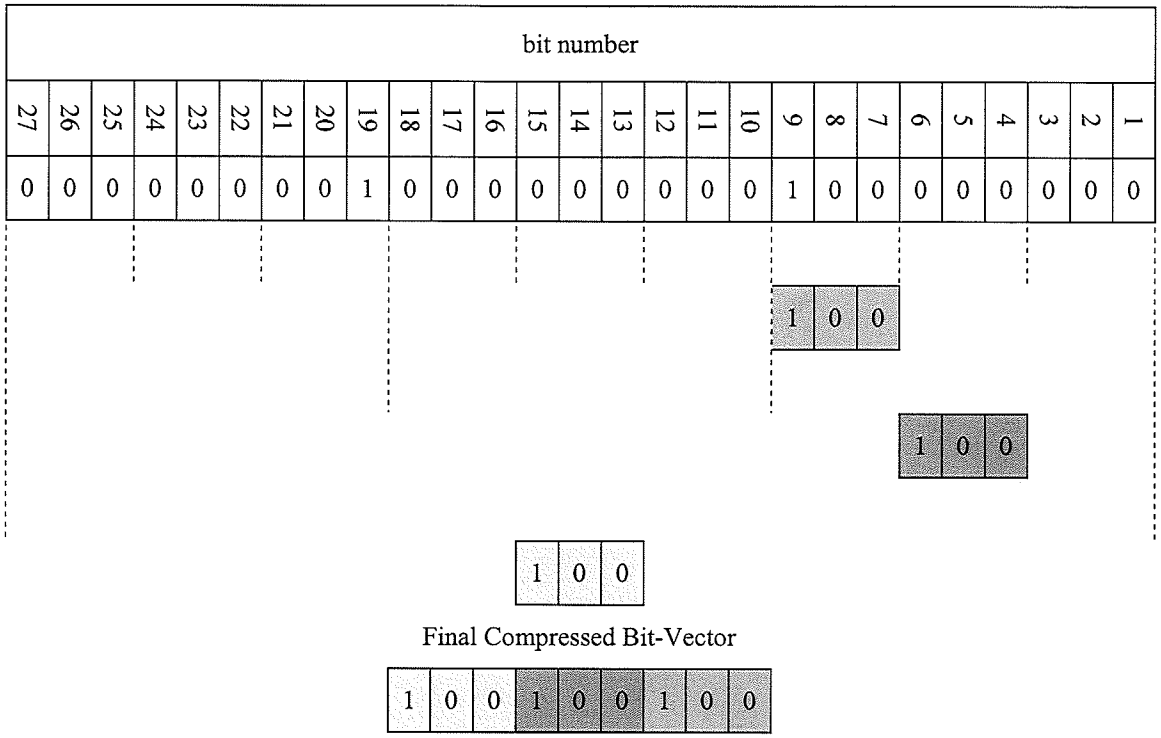

Figure 3-5 : Second Level B-tree CBV Search Result

7. The resultant solution is then returned to software for logging. With regard to the actual implementation, and not the example, it would have been preferable to provide this solution to a hardware block to perform the final AND operation but due to hardware limitations this was not done. As mentioned earlier only one processor was available to produce the results from one 2-dimensional search so the implementation of the AND hardware was determined to be unnecessary. Rather this was proposed to be done in a post processing step. The OR operation is more complex than the AND operation and as such was seen as an adequate measure of performance considering the two operations would be pipelined anyways.

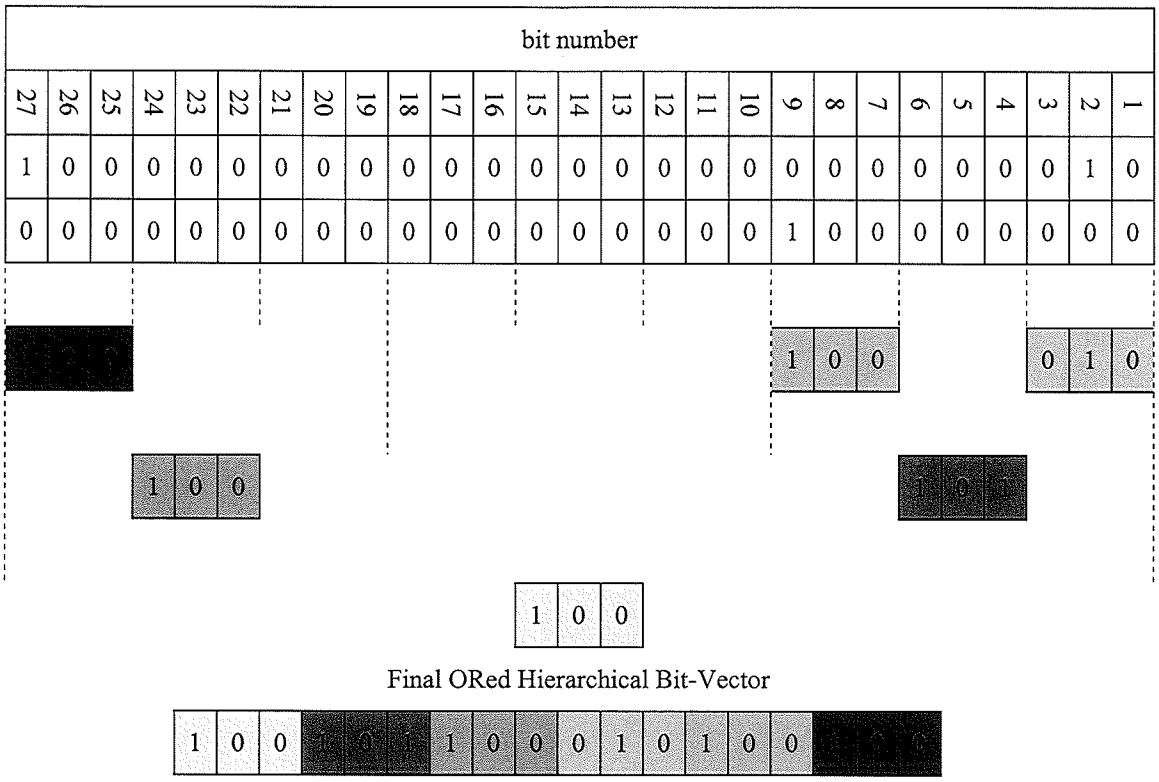

Figure 3-6 : Final ORed Result of Level 1 & 2 CBVs

3.4 Operation Complexity

Given the operations previously described the complexity for the search, build time, and memory usage is presented.

3.4.1 Search Time Complexity

As described previously the search time is broken up into two pipelined operations: a software search and a hardware search.

The software search complexity is calculated by determining the amount of time spent at each node of in the B-tree, multiplying by the worst-case height of the tree, and then by factoring in the dimensions and bucketing factor. As the number of keys to search at each node is a function of the B-tree parameter *t*, the amount of time spent at each node is of complexity $O(t)$. Multiplying by the height of the tree, h, produces a complexity of $O(th)$. Based on Equation 2-2 the height of the B-tree is substituted to produce $O\left(t \log_t \left(k + 1/2\right)\right)$, where k is the number of

intervals. Given n rules the maximum number of elementary intervals is $2n+1$. After substitution for k the complexity simplifies to $O(t\log_t n)$. Taking into consideration the dimensions, d, the complexity becomes $O(d \times t\log_t n)$. Including the bucket factor results in a reduction in the number of rules by a factor of b, but increases the number of structures searched by b. The result is a multiplication by b and a division of n by b to produce $O\left(b \times d \times t\log_t \frac{n}{b}\right)$. For constant values of b, d, and t this is reduced to $O(\log_t n)$.

The hardware portion of the search operation is quite a bit simpler to derive. The most complicated operation found in hardware is the OR operation. The OR operation is performed on bit-vectors with a maximum size of n, where n is the number of rules. The worst-case operation is to perform an OR operation such that all n bits are examined. As such, the worst-case operation is a function of n, and is therefore $O(n)$.

Given that the complexity of the search operation is simply the largest of the complexities of the software and hardware operations the complexity of the search operation is $O(n)$.

3.4.2 Build Time Complexity

Like the search complexity the build complexity can be broken down into a software and a hardware component.

The software portion of the build complexity is calculated by considering the complexity to insert a single item into the B-tree structure. The complexity to insert a range into a single dimension of the B-tree is $O(\log_t n)$, where n is the number of rules and t is the B-tree parameter. Given the maximum number of ranges to insert is a function of n the complexity becomes $O(n\log_t n)$. Taking into consideration both dimensions of the B-tree the complexity becomes $O(n^2(\log_t n)^2)$.

The hardware complexity is simply the complexity of the operation to produce the CBVs. The CBVs are produced from bit-vectors with a maximum size of n, where n is the number of rules. The worst-case operation is found when all n bits are examined. As such, the worst-case complexity is a function of n, and is therefore $O(n)$.

Unlike the search operations, the software and hardware portions of the build operations are not pipelined. As such, the worst-case hardware and software combined build time

complexity is $O\left(n^3(\log_t n)^2\right)$.

3.4.3 Memory Use Complexity

The memory use complexity is calculated by considering the size of the B-tree structure and compressed bit-vectors. Given n rules and b buckets the maximum number of elementary intervals in the first dimension of one level of the B-tree is $O\left(2n+1/b\right)$. Taking into consideration the second dimension the complexity is squared to produce $O\left(\left(2n+1/b\right)^2\right)$. In the worst-case each elementary interval in the second dimension has n bits set so the complexity becomes $O\left(\left(2n+1/b\right)^2 n\right)$. The equation is then simplified to remove constants, including b, to produce a worst-case memory use complexity of $O\left(n^3\right)$.

4 Design Environment and Platform

For the purposes of implementation of the design specification the RPP from CMC is employed. The platform, described at a high level in the following sections, contains the basic elements and development environment needed for such a complex design. These elements include a processor, software development environment, support code for low level firmware drivers, large FPGA for logic development and basic bus logic blocks for interfacing with the processor. While the performance of the processor is limited, the system provides an exceptional prototyping tool. Figure 4-1 shows a picture of the development system and Figure 4-2 and shows a close up of the hardware.

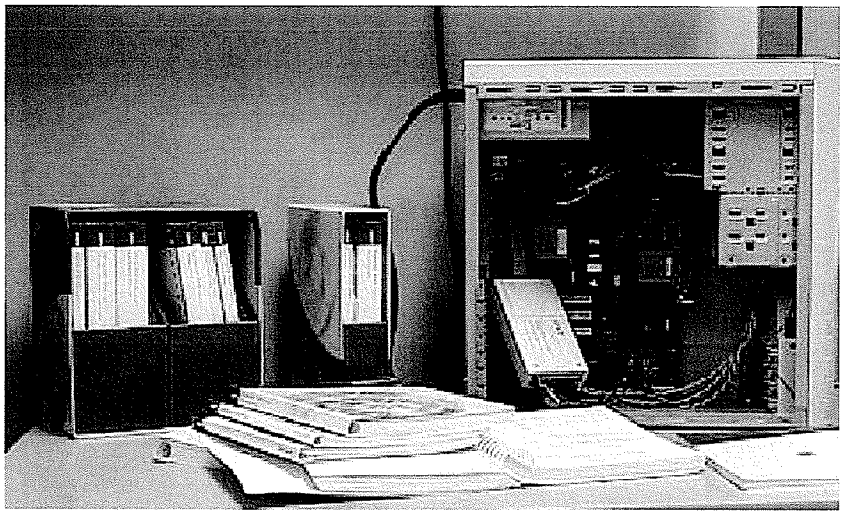

Figure 4-1: ARM Integrator Rapid-Prototyping Platform [10]

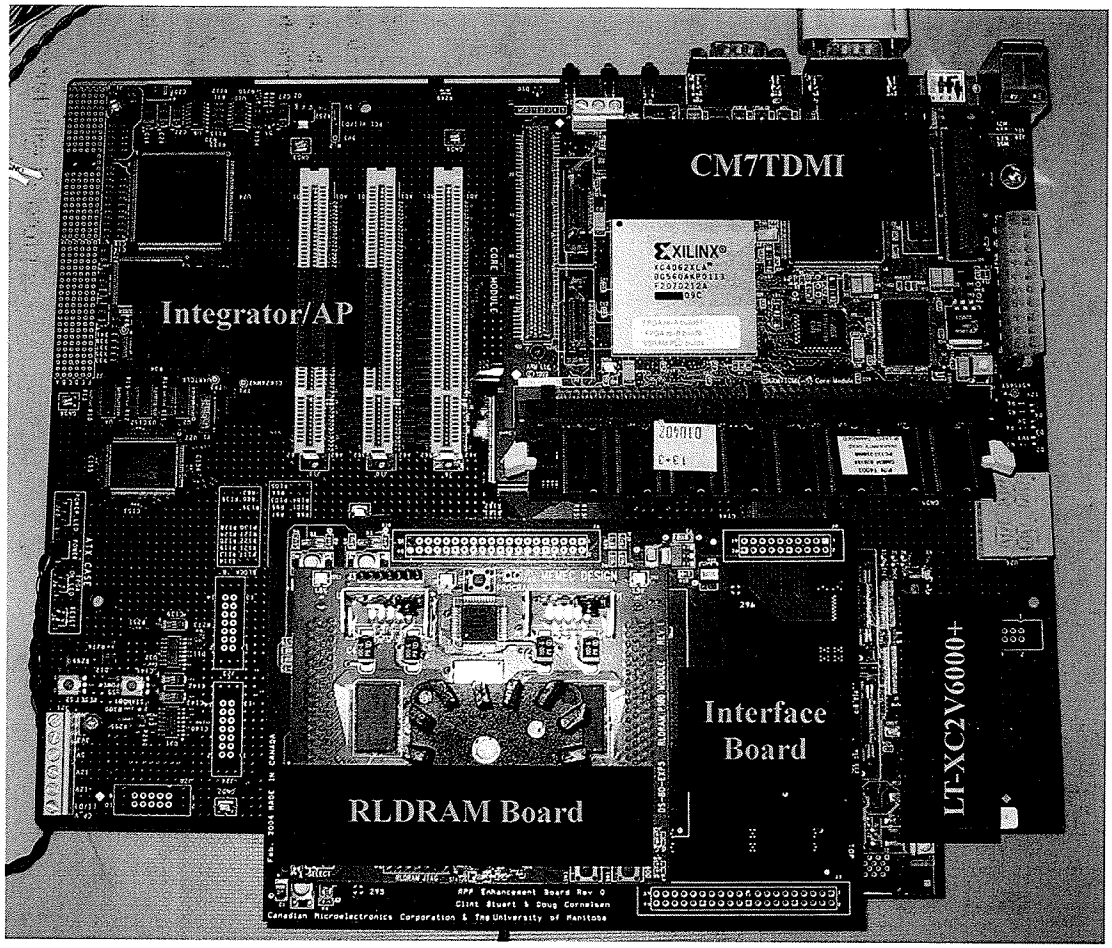

Figure 4-2: Development Hardware Close Up

4.1 ARM Integrator/AP ASIC Development Motherboard

At the base of the RPP system is the ARM Integrator ASIC development mother board. Its primary function is to provide a connection point for a large selection of ARM processor and logic development boards. It provides these development boards with clocks, bus arbitration, interrupt handling, flash memory, boot ROM and basic input and output functionality. Expansion of the system is provided by three PCI slots, connectors for stack ups of both processors and logic modules and a cPCI interface for rack mounting.

Figure 4-3 and Figure 4-4 show a functional block diagram and layout of the mother board.

The basic features of the mother board, as described in [11] are as follows:

- system controller FPGA that implements:
    - system bus interface to processor core and logic modules
    - system bus arbiter
    - interrupt controller
    - peripheral input and output controller
    - three counter/timers
    - reset controller
    - system status and control registers
- clock generator
- 32 MB flash memory
- 256 KB boot ROM
- 512 KB SSRAM
- two serial ports (RS232 DTE)
- system expansion supporting processor core and logic modules
- PCI interface bus
- External Bus Interface supporting memory expansion

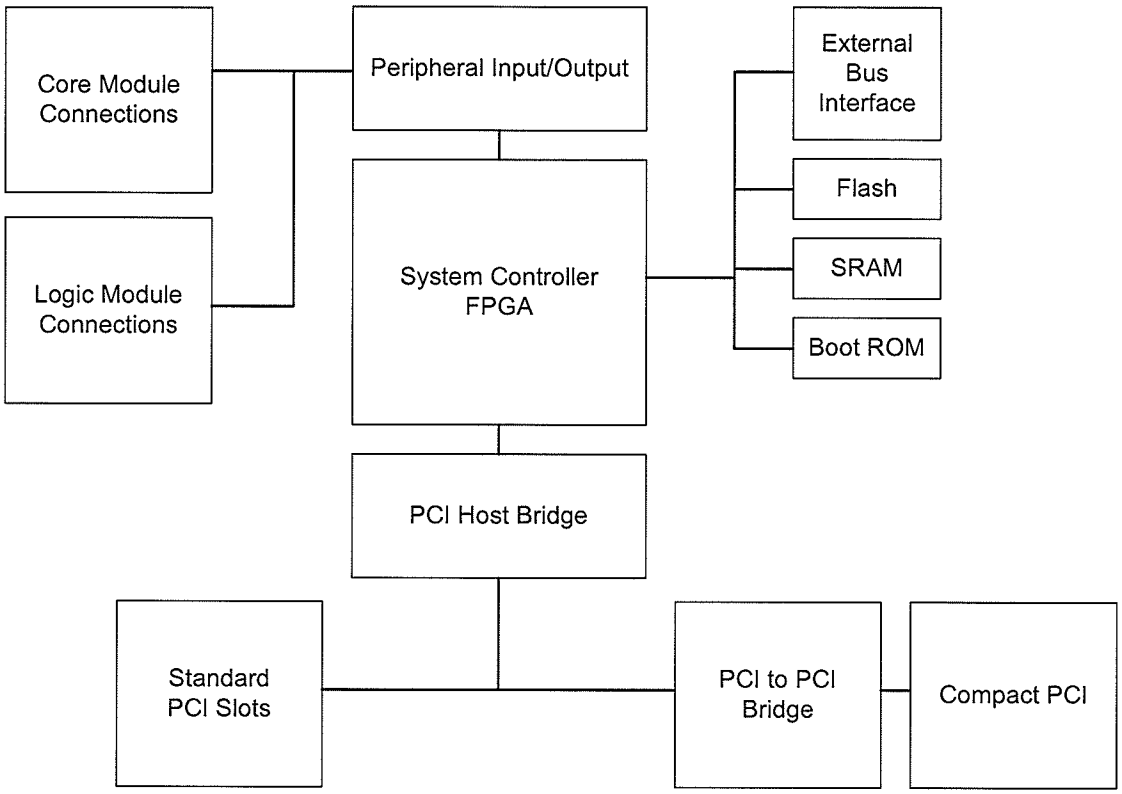

Figure 4-3 : ARM Integrator/AP Functional Block Diagram [11]

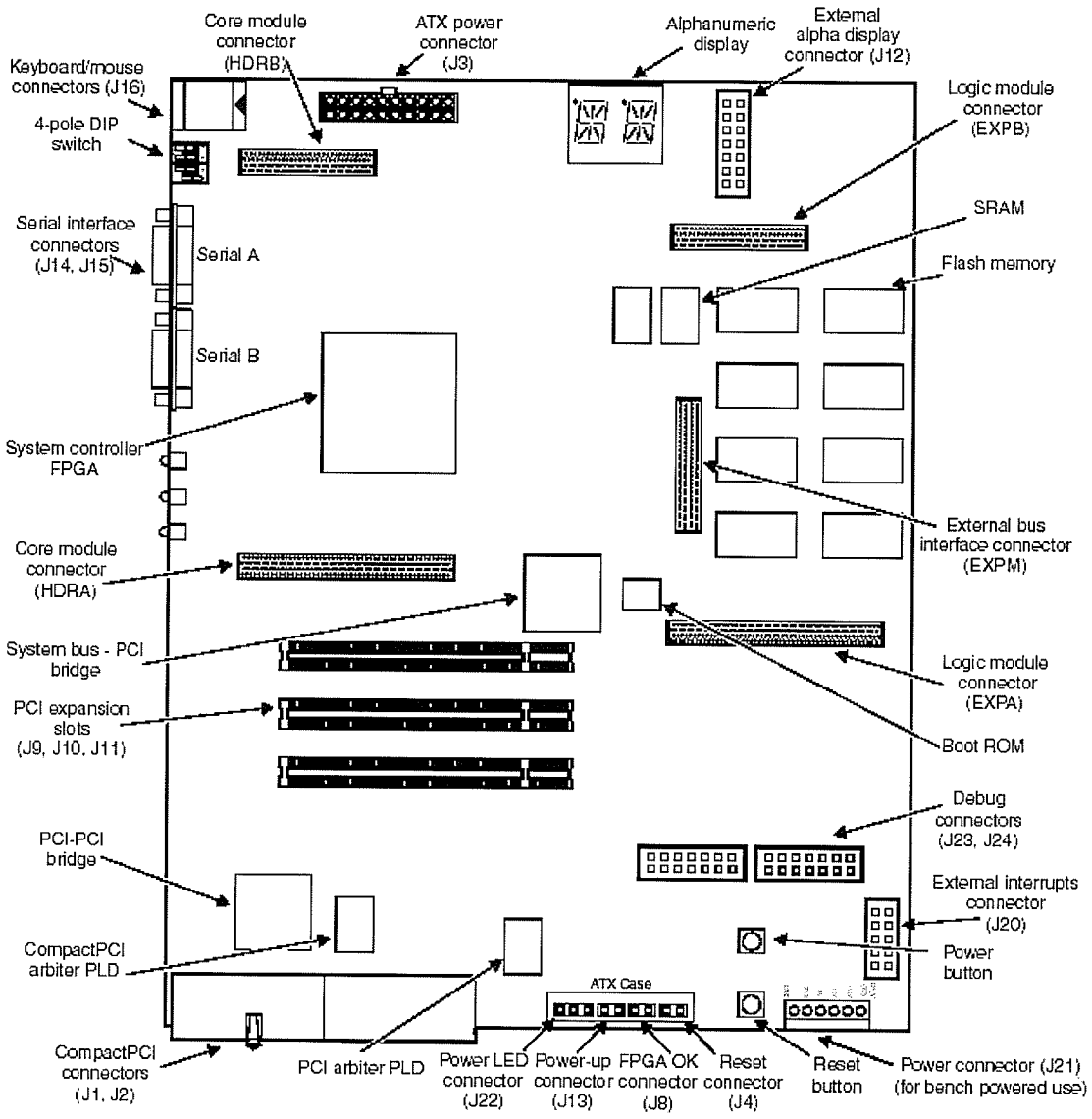

Figure 4-4 : ARM Integrator/AP Layout [11]

4.2 ARM Integrator/CM7TDMI

The processor core board available for use with the AP ASIC development mother board is the CM7DTMI. The CM7DTMI development board contains the popular ARM7TDMI processor, an FPGA for memory control and bus interfacing, memory and a Multi-ICE interface for debugging. Figure 4-5 and Figure 4-6 show a functional and layout block diagram of the board.

The basic features of this board, as described in [12] are as follows:

- ARM7TDMI microprocessor core
- core module FPGA that implements:
    - SDRAM controller
    - system bus bridge
    - reset controller
    - interrupt controller
    - status, configuration, and interrupt registers
- volatile memory comprising:
- 128 MB of SDRAM
- 256 KB SSRAM
- SSRAM controller
- clock generator
- system bus connectors
- Multi-ICE, logic-analyzer, and optional Trace connectors

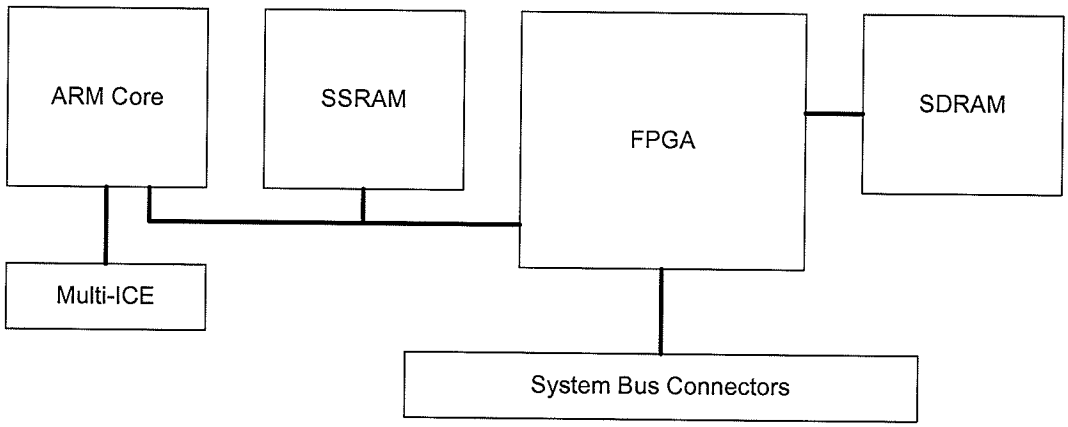

Figure 4-5 : ARM Integrator/CM7TDMI Functional Block Diagram [12]

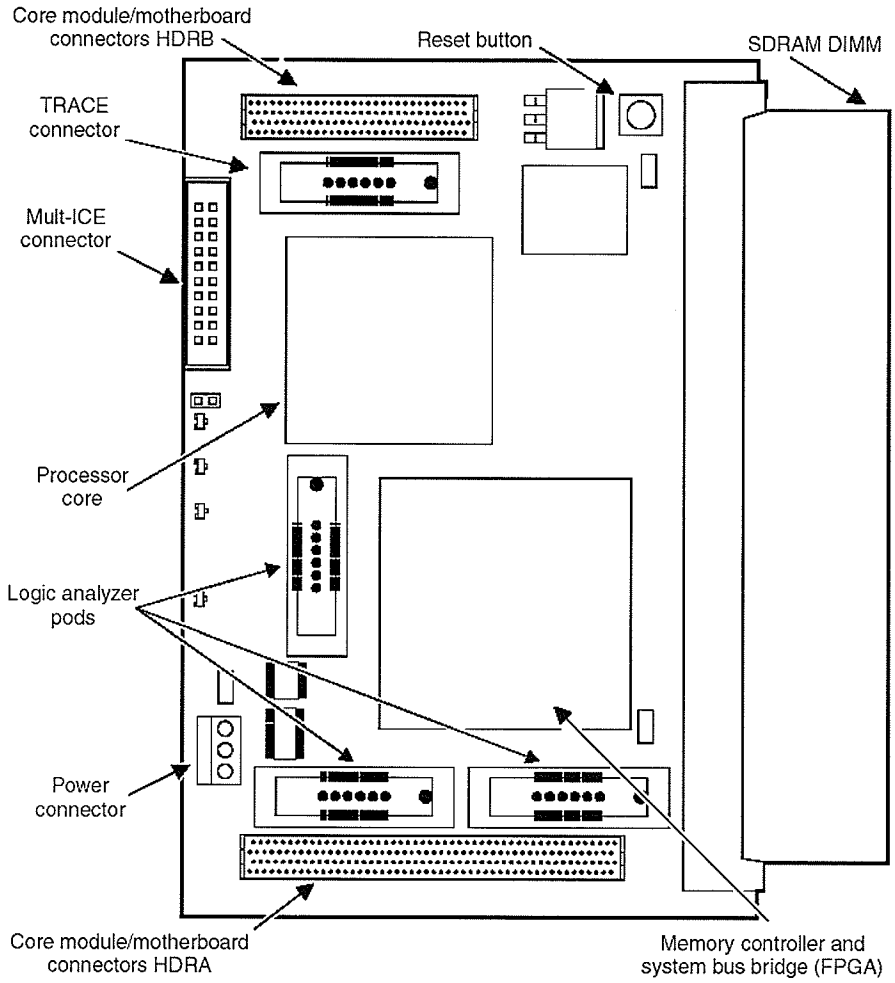

Figure 4-6 : ARM Integrator/LT-XC2V6000+ Logic Tile [12]

4.3 ARM Integrator/ LT-XC2V6000+

The FPGA board available for use with the AP ASIC development mother board is known as the LT-XC2V6000+ Logic Tile. The LT-XC2V6000+ development board contains a Xilinx XC2V6000 FPGA and is specifically designed for the development of ARM AHB and ASB bus peripherals. Figure 4-7 and Figure 4-8 show a functional and layout block diagram of the board.

The basic features of this board, as described in [13] are as follows:

- Xilinx Virtex II FPGA
- configuration *Programmable Logic Device* (PLD) and flash memory for storing FPGA configurations
- two 2MB ZBT SSRAM chips

- clock generators and reset sources
- switches
- LEDs
- connectors to other tiles

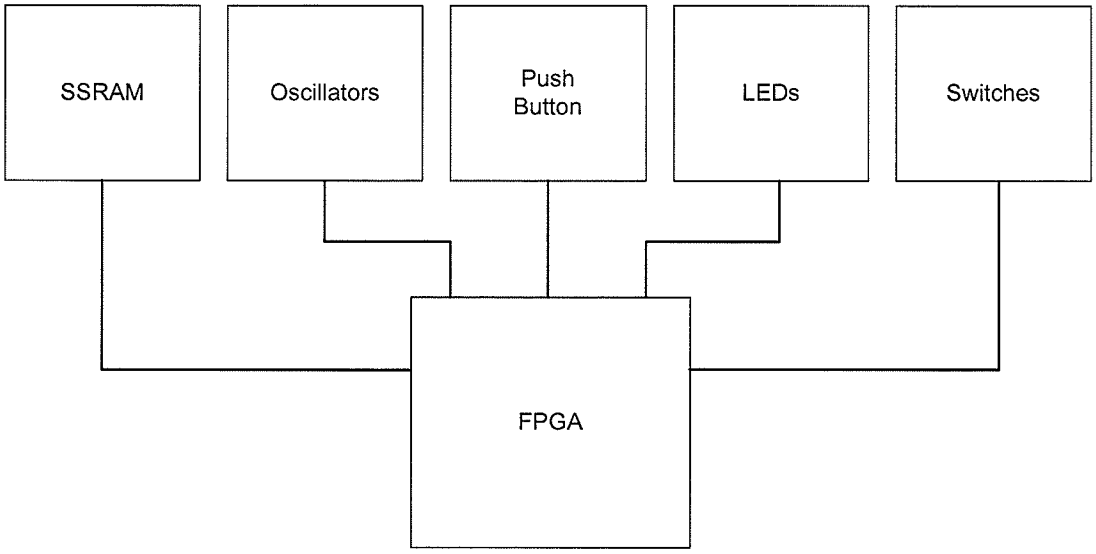

Figure 4-7 : ARM Integrator/LT-XC2V6000+ Functional Block Diagram [13]

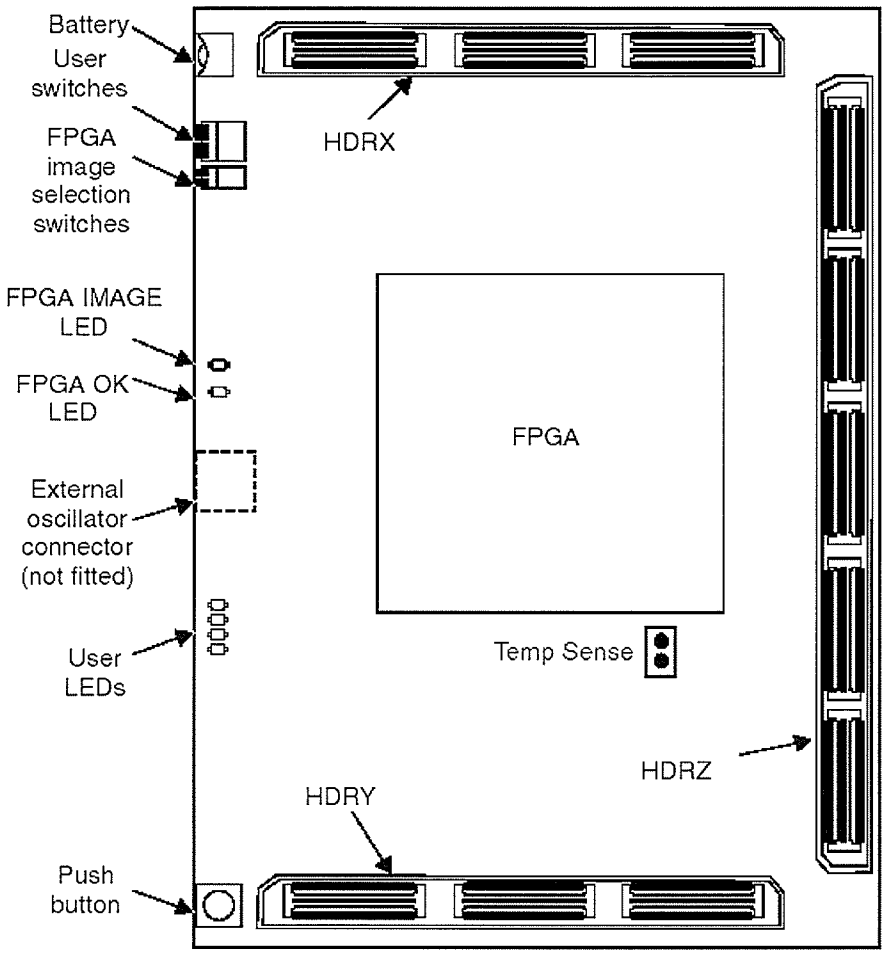

Figure 4-8 : ARM Integrator/LT-XC2V6000+ Layout [13]

4.4 Memec MC-XIL-RLDRAM Controller and Board

Early in the project it was determined the four Megabytes of memory available on the LT-XC2V6000 development board would not be enough for compressed bit-vector storage. A suitable option found to provide a large amount of very high-speed memory and additional logic space was the Memec *Reduced Latency Dynamic Random Access Memory* (RLDRAM) evaluation board. The RLDRAM evaluation board was chosen because of its exceptional memory throughput, additional logic space provided by a Xilinx Virtex-II FPGA and standard Memec P160 interface. The board contains four Infineon Technologies HYB18RL25632 32-bit wide RLDRAM chips operating at 200 MHz DDR (400Mbit/s/pin). In total the four chips provide 128 Megabytes of storage space with an aggregate access speed of 51.2 Gbit/s. As well

as providing high bandwidth solution RLDRAM also provides a random access time much faster than typical SDRAM. The RLDRAM controller, resident in the FPGA, was purchased as an intellectual property core in EDIF form. Due to a non disclosure agreement signed at the time of purchase no discussion is provided on it operation or design. Figure 4-9 shows a block diagram of the board.

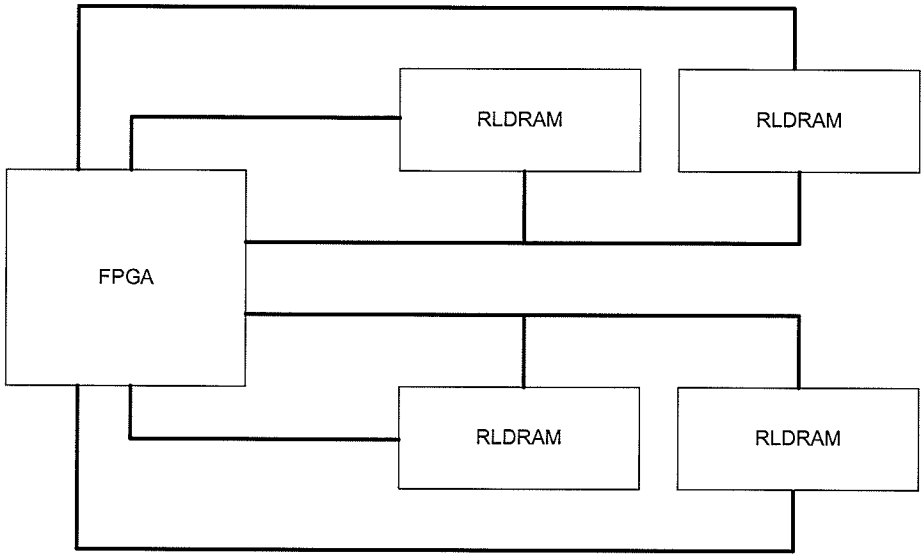

Figure 4-9 : RLDRAM Board Block Diagram

4.5 Custom Interface Board

To provide a high-speed connection between the logic tile and the RLDRAM board a custom interface board was designed. The high-speed interface between the two boards comprises high-speed LVDS pairs utilizing Xilinx Virtex II capabilities. Enough pairs are routed between the P160 interface on the RLDRAM board and the Samtec connector interface on the logic tile to provide for eight LVDS pairs in each direction. Each pair operates at 200 MHz DDR (400 Mbit/sec/pair) providing 1.6 Gbits/sec in each direction. Given the limited performance of the ARM processor the design effort was made to gain useful experience and to provide a path for future growth. In particular, design experience was gained performing the impendence calculations and routing for the high-speed LVDS pairs. Figure 4-10 shows a picture of the custom interface board.

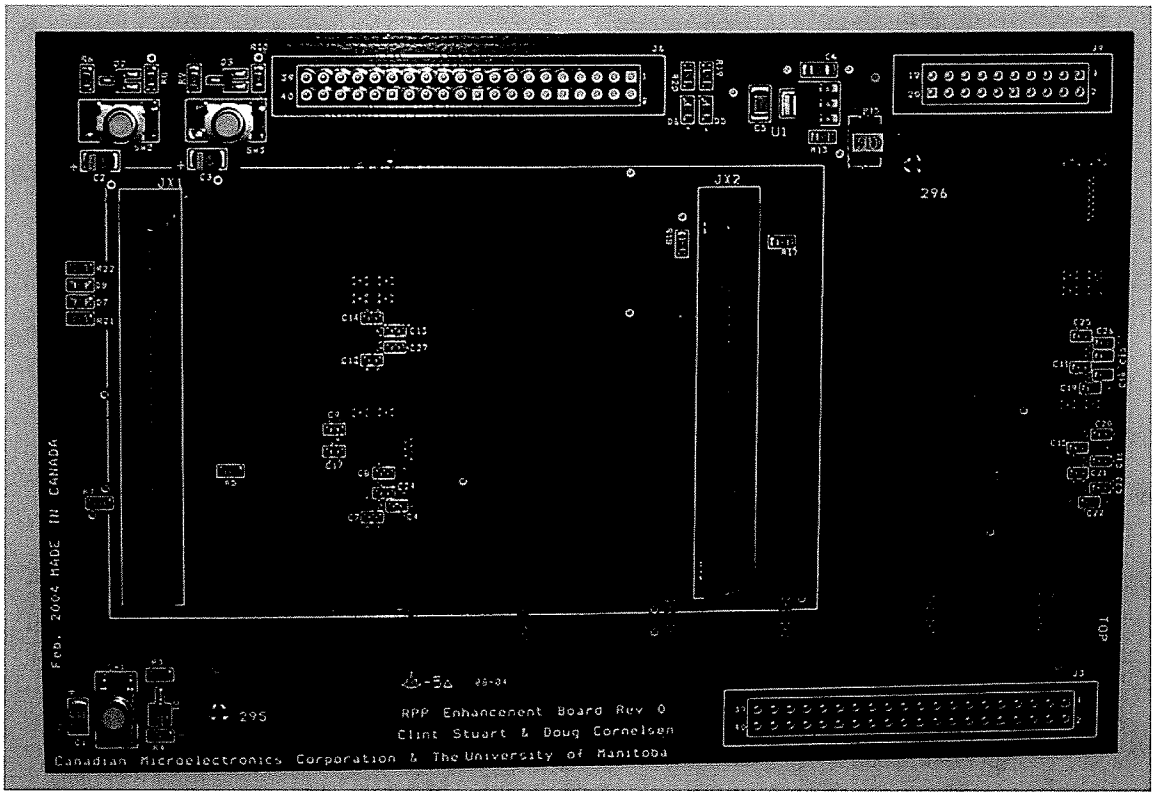

Figure 4-10 : Custom Interface Board

5 System Design

5.1 Design Responsibilities

As the specification, design and verification effort for the entire project was quite large it was determined two group members would be required. At the outset of the project clear divisions in responsibilities were setup with regard to the hardware and software design tasks. As with most complex embedded systems the hardware and software design tasks are numerous and often interdependent. The goal of the project was to create a specification for the system as a group then divide the tasks between hardware and software with integration being performed on a continual basis. The major focus of my efforts during this thesis was spent on software development for the ARM processor and developing FPGA hardware to interface to a purchased RLDRAM controller. In particular my design tasks included:

1. Software design:
    a. Implement B-tree structure and functions for insertion of rules.
    b. Implement menu structure for software tasks to ensure ease of use.
    c. Implement low level tests for bus interfaces and memories.
    d. Implement software tests for custom and purchased hardware blocks.
    e. Implement low level drivers and functions for performing B-tree build operations and search operations with custom hardware.
    f. Implement software logging features to match hardware simulation inputs so failures discovered in tests with actual hardware could be used as inputs in simulations. Tests ran on actual hardware and software perform orders of magnitude faster than simulation but do not provide the same visibility.
    g. Implement rule generation software to produce random rule-sets based on input files containing probability distribution functions.
2. Hardware design:
    a. Assist in the implementation and testing of Xilinx hard macro LVDS SerDes blocks.
    b. Develop a wrapper for purchased RLDAM controller core to handle

refreshing operations and commands from a FIFO to transfer memory between block FPGA block RAMs and RLDRAM memory.

c. Assist in the design of the interface board between logic tile board and RLDRAM board.

The remaining responsibilities, handled by my research partner Clint Stuart, were as follows:

1. Software design:
    a. Implement Tcl test bench routines for interfacing with simulation environment.
    b. Implement Tcl routines for conversions of software logged files for running in simulation environment.
    c. Design a simulation environment for the FPGA hardware.
2. Hardware design:
    a. Implement ARM AHB bus wrappers for hardware command and response FIFOs.
    b. Implement and test Xilinx hard macro LVDS SerDes blocks.
    c. Implement logic blocks for compressed bit-vector build operations.
    d. Implement logic blocks for compressed bit-vector OR operation.
    e. Implement logic blocks for hardware test modes.
    f. Produce schematic and layout designs for interface board between logic tile and RLDRAM board.

With regard to responsibilities, this report focuses largely on my responsibilities related to the overall project. As the system required both hardware and software for full functionality the testing, verification and results portions of the work were shared. While the design and implementation work were separated as much as possible the work done by the both of us produced one final system producing one set of final results.

5.1.1 Hardware/Software Design Flow

As was identified in the previous section the project tasks were largely divided between hardware and software development. When designing an embedded system it is important the hardware and software are co-designed to make testing and integration easier. During the

development phase it became important for the hardware development to have access to test software so it could be tested at hardware speeds instead of in simulation. Likewise the software development benefited from testing using actual hardware instead of relying solely on simulation. The design flow used, illustrated in Figure 5-1, allows for rapid development of both the hardware and software and leverages the benefits of both simulation and hardware testing as appropriate. The top of the design flow begins with the most important phase of any project, the design specification. The design specification breaks down a complex design into fundamental operations which can be assigned to either a software or hardware. Design partitioning is the phase in the design flow to determine which operations will be done in software or hardware. Typically after design partitioning marks the point at which software and hardware diverge until each item is ready for integration. Using a design platform specifically designed for rapid prototyping allows for software and hardware integration to be done on a continual basis. In effect many loops can be done from the top to the bottom integrating software and hardware as development occurs. This type of design philosophy proved to be effective as the different buses and memories were integrated into the design. A major benefit of this flow is the enforcement of a bottom up verification process in which each element is completely tested before the element above it is designed.

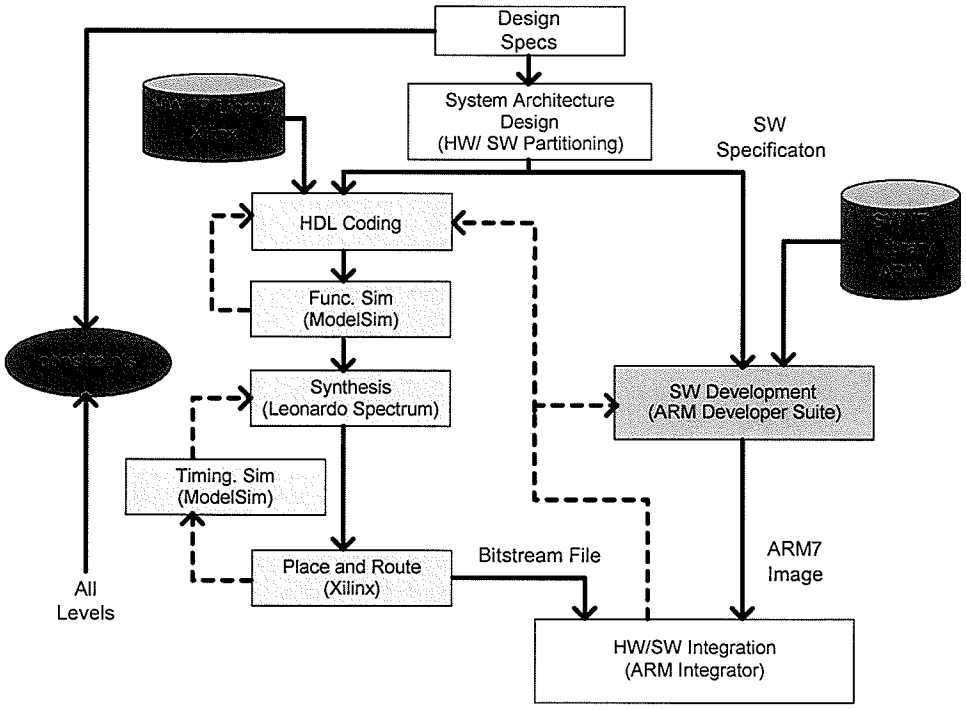

Figure 5-1 : SOC Hardware/Software Design Flow [14]

5.2 Hardware Design and Implementation

5.2.1 Design Methodology

The hardware portions of this thesis were written in VHDL targeted for the two separate Xilinx Virtex II FPGAs found on the RLDRAM and LT-XC2V6000+ boards. The design flow follows Figure 5-1 which shows that ModelSim was used for RTL and timing simulations, Leonardo Spectrum was used for synthesis and the Xilinx tool ISE was used for place and route. Over the course of the project this suite of tools proved to be very effective providing excellent simulation capabilities and detailed representations of the designed hardware.

5.3 Hardware Overview

When all of the circuit boards, described in previous sections, are connected together a very powerful and capable development system is created. Figure 5-2 illustrates the bus and component hierarchy of the final design. The items shown in Figure 5-2 are color coded based on the location of the component in the particular development board.

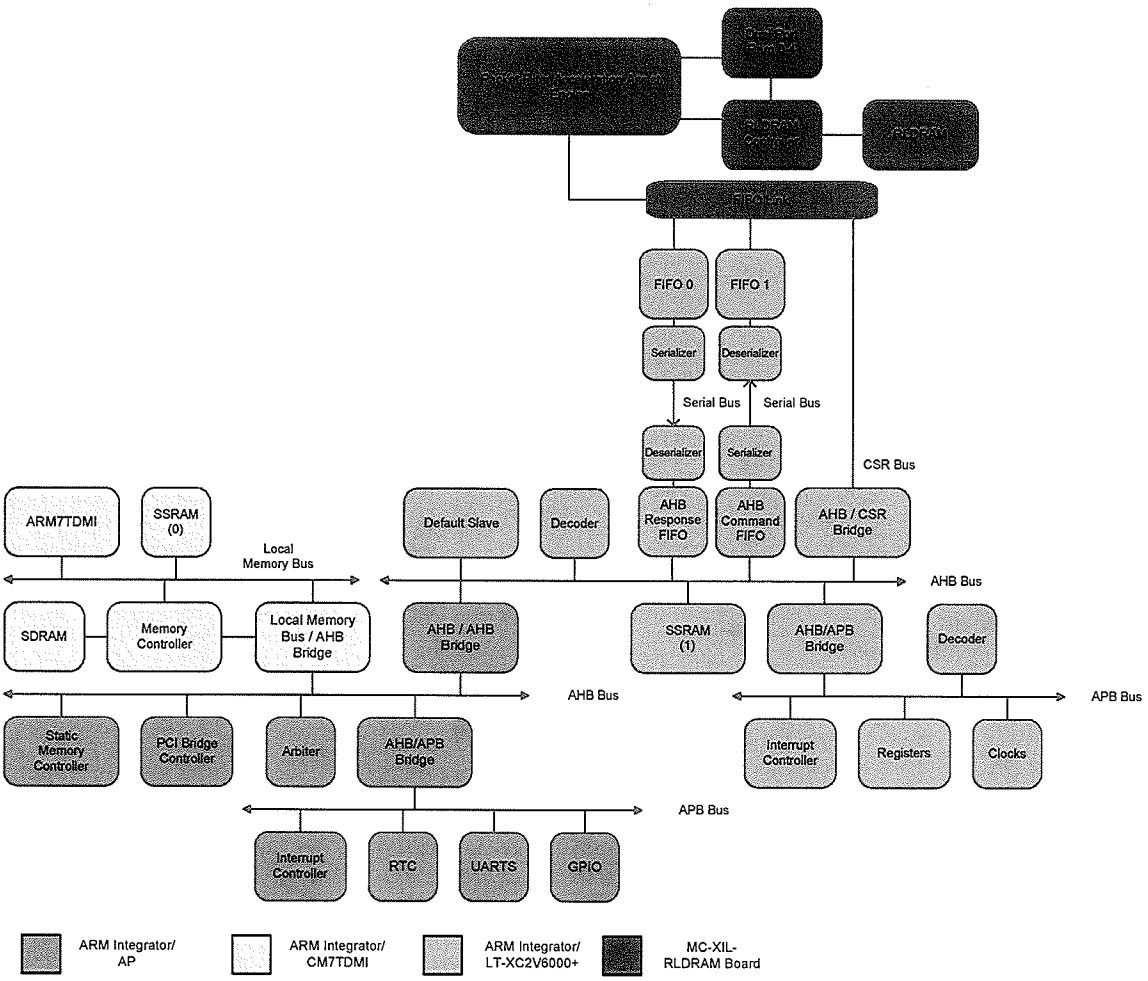

Figure 5-2 : Hardware Development Components and Bus Hierarchy

5.3.1 Hardware Blocks

As the focus of this thesis is largely on the software development only a high level overview of the hardware blocks is provided.

ARM7TDMI [15]

The processor used in this thesis is the ARM7TDMI. Figure 5-3 provides a high level block diagram of this 32-bit embedded RISC processor. Of note of the blocks shown in Figure 5-3 is the embedded ICE logic. The embedded ICE logic is an extremely useful feature of the ARM7TDMI providing access for a JTAG-based debugging system. The ARM Multi-ICE debugging tool is used for this purpose to provide typical debugging features and access to files on a Multi-ICE host computer. For the purposes of this thesis the processor was run at a speed of 40 MHz.

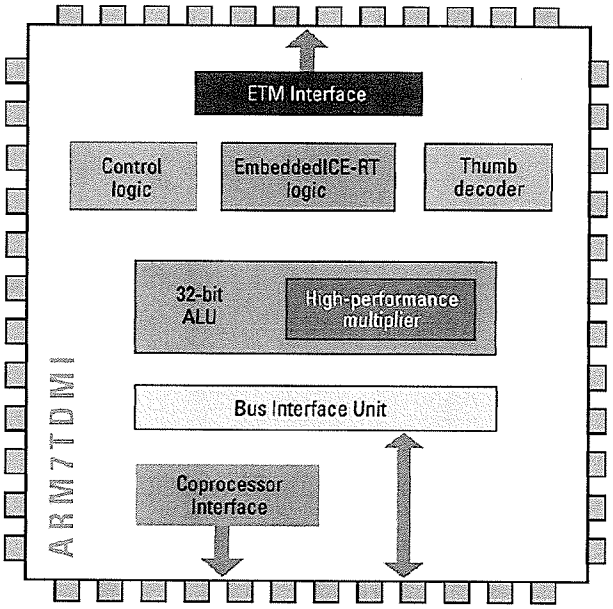

Figure 5-3 : ARM7TDMI Processor Block Diagram [16]

ARM APB Bus

The *Advanced Peripheral Bus* (APB) is part of the ARM family of *Advanced Microcontroller Bus Architecture* (AMBA) buses designed for on chip communication. It is primarily used to provide connectively to low speed devices and is designed with a simple interface and for minimal power usage. The APB bus is typically used in conjunction with a system bus like AHB. APB is implemented in the system as a 32-bit bus running at 20 MHz. For more information see reference [17].

ARM AHB Bus

The *Advanced High-Performance Bus* (AHB) provides a high level of capabilities and is typically used as a system back bone. AHB is implemented in the system as 32-bit bus running at 20 MHz. For more information see reference [17].

Command FIFO

The command FIFO logic block provides an AHB interface to send commands to the PFAAE. Commands inputted into the FIFO are serialized by a special hardware block and then sent via LVDS pairs to the RLDRAM FPGA. Special logic was made so commands could be gated in the FIFO so that accurate performance estimates could be made of the PFAAE. This was

done because the PFAAE could process commands faster than the processor was able to write them to the FIFO.

Response FIFO

The response FIFO logic block provides an AHB interface to a buffer containing responses from the PFAAE.

Control Status Register Bus

The *control status register bus* (CSR) is used to read and write registers in the PFAAE. It provides a low speed method of accessing registers as opposed to using the high-speed serial buses. The serial buses use complex commanding and response methods for transmitting and receiving large packets of data. This type of complexity is not needed for simple reads and writes so the CSR bus was implemented.

LVDS SerDes

The LVDS *serializer/deserilizer* (SerDes) blocks are ported from Xilinx application note XAPP265 "High-Speed Data Serialization and Deserialization (840 Mb/s LVDS)" [18]. The application note provides a code base and details on how to implement high-speed SerDes logic for the Xilinx Virtex II family of FGPAs.

SSRAM 1

The SSRAM found on the LT-XC2V6000+ board is extremely important for testing. This memory is used for storing rules and CBV pointers. The rules are used for tests in which the complete B-tree structures and CBV are built. CBV pointer storage is used for hardware performance testing of the PFAAE.

Dual Port RAM 0 - 4

Dual Port RAM 0 to 4, located in the RLDRAM FPGA are memories used by PFAAE for operating on and creating CBVs. As the name implies, these memories are dual port such that two different logic blocks have interfaces to the memory. Data transferred to and from the RLDRAM can only be done through the dual port memories. When the PFAAE finishes constructing a CBV, it is transferred from one of the dual port memories to the RLDRAM in a DMA like operation. Likewise, in search mode when a CBV pointer arrives from the software the PFAAE requests the CBV located in RLDRAM be transferred to one of the dual port memories.

RLDRAM Wrapper

The main purpose of the RLDRAM wrapper is to provide a command interface to the RLDRAM controller and to ensure refresh commands are issued as required. The most important portion of the wrapper is a FIFO command interface so the PFAAE can queue up several different transfers between the dual port memories and the RLDRAM. A state machine in the wrapper reads out and executes commands from the FIFO one at a time inserting refreshes between FIFO commands when necessary. To execute a command the state machine reads the type of command, read or write, issues the proper command to the RLDRAM controller and then controls the data flow between the RLDRAM controller. Whenever a command from the FIFO finishes being executed, a done signal is pulsed so tracking of command completion can be performed.

RLDRAM Controller

The RLDRAM controller was a purchased IP block from Memec provided in EDIF form. No details of the controller will be provided as a non-discloser agreement was signed at the time of purchase.

Packet Filter Acceleration Assist Engine

The main component found in the RLDRAM FPGA is the PFAAE. It consists of four main subcomponents each handling one of the four major operations required. Each of these four operations corresponds to a particular operating mode outlined below:

Build Mode: The major function of this mode is to convert lists of rules into hierarchical CBVs. Once built the CBVs are stored in external RLDRAM and a pointer is returned to the software.

Filter Mode: The function of this mode is to provide hardware acceleration for the packet filtering operations. In particular it retrieves CBVs from RLDRAM and then performs the OR operation. Two sub modes of operation are available known as quick and normal mode. In normal mode the full resultant CBV of the OR operation is provided in the response FIFO. By contrast in quick mode only a flag indicating the operation is complete is written into the response FIFO. Quick mode is used to as a feature to accurately benchmark the throughput of the filter mode operations without the overhead of the sending the full CBV responses.

Loop Mode: The function of this mode is to provide a feature to loop back data received

by the FIFO link block on the RLDRAM FPGA back to the response FIFO. This feature is used for testing of the buses and FIFOs.

User Command Mode: The function of this mode is to provide access for test commands to ensure correct hardware operation. Primary functions include commands for performing memory transfers between RLDRAM board dual port RAM memories, the command/response FIFOs and the RLDRAM.

5.4 Software Development and Implementation

5.4.1 Design Methodology

The software portion of this thesis is written in C using Code Warrior and the *ARM Firmware Suite* (AFS) targeted for the ARM7TDMI processor. The ARM Firmware Suite proved to be vitally important in development as it provides development board independent functions including: system initialization, serial port drivers, timers, interrupt control and memory management. Software written using the API provided by the ARM firmware suite is capable of running on various development platforms provided the proper *Hardware Abstraction Layer* (HAL) is used. The Hardware Abstraction Layer provided by the ARM Firmware Suite is called *Micro Hardware Abstraction Layer* (μHAL). μHAL provides the middle layer, shown in Figure 5-4, which aids in the development of new code and porting of operating systems.

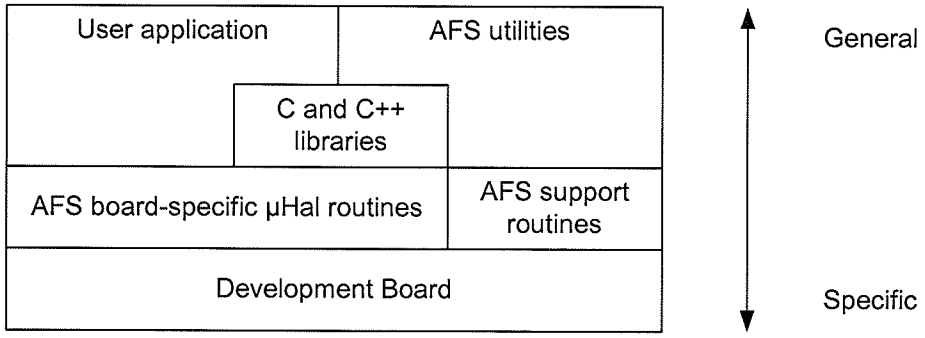

Figure 5-4 : ARM Firmware Suite [19]

In an embedded system designed to run multiple tasks, like the one employed for this project, a *Real Time Operating System* (RTOS) is typically used to provide the framework for task switching. At the start of the project the MicroC/OS-II RTOS was reviewed as one potential candidate. Given more time the MicroC/OS-II operating system would have been selected because of its small size, free academic license, excellent reputation in industry and an available

port for the development platform. The reason it was not used is because of the time constraints of the project and the limited processing capabilities of the processor. As the processor does not have the capability to run one task at the speed of the hardware, there is little benefit in having the ability to switch tasks. As well, the operations performed during testing are sequential in nature and can be performed adequately without task switching. While concurrent operations are necessary in an actual design it is not required for testing purposes. The following are the three major operating modes of the software, each of which is run at a different time:

1. **User Command Mode**, also known as Test Mode: Includes memory tests and bus tests.
2. **Build Mode**: Includes all of the operations to build the B-tree structures and CBV's from an input rule file.
3. **Filter Mode**: Includes all of the operations necessary to perform a search of test points.

To capitalize further on the simplification of having separate non concurrent modes the hardware is also written to operate in one of these modes at a given time.

A high level flow of the software showing its basic operating states is shown in Figure 5-5. The software begins with a boot sequence which initializes clocks, timers, memory and interrupts. Once the system is initialized a menu is displayed outlining various groupings of software which can be run.

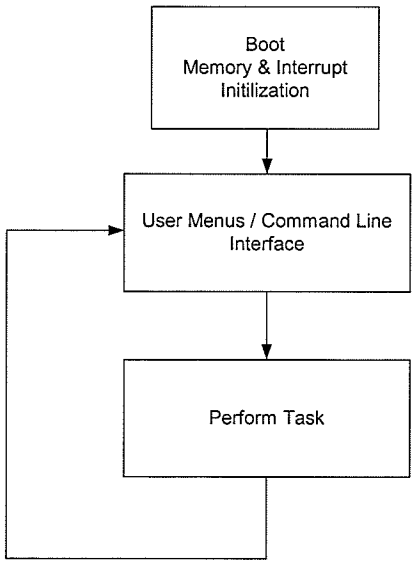

Figure 5-5 : Software Operating States

5.4.2 Menu Description

The hierarchy of the menu system described in this section is shown through indentation in a series of figures. Access to items in the menu or sub menus is controlled through a serial port, on a test computer, using the number keys of the keyboard. For example, if a user desires to view the *Tests* Menu under *Main* Menu they simply push three on the keyboard. If the user then wants to run a test they push the number of the desired test. The test runs and the menu becomes active again once it is finished running. A press of the number zero results in a move back one step in the menu hierarchy. Described in the rest of this section are the submenus for running various tasks.

```
****************************************
Main Menu
1: User Cmd Mode...
2: Packet Filter Mode...
3: Tests...
Please enter a selection
****************************************
```

Figure 5-6: Main Menu

Main Menu: Main Menu is the top level menu seen when the software is started. It contains three sub menus: User Cmd Mode, Packet Filter Mode and Tests.

```
****************************************
User Cmd Mode
1: RLDRAM Tasks...
2: FIFO Tasks...
Please enter a selection
****************************************
```

Figure 5-7: User Command Mode Menu

User Command Mode Menu: User Cmd Mode contains two items RLDRAM Tasks and FIFO Tasks. These items are a grouping of tasks related to the commands available to be sent to the hardware through the command FIFO.

```
****************************************
RLDRAM Tasks
1: Rd DPR 0 to RLDRAM
2: Rd DPR 1 to RLDRAM
3: Rd RLDRAM to DPR 0
4: Rd RLDRAM to DPR 1
5: RLDRAM Task Parameters...
Please enter a selection
****************************************
```

Figure 5-8: RLDRAM Task Menu

RLDRAM Task Menu: RLDRAM Tasks Menu contains a list of tasks for testing the transferring of data between RLDRAM and Dual Port RAM 0 or 1.

```
****************************************
FIFO Tasks
1: Rd FIFO 1 to DPR 0
2: Rd FIFO 1 to DPR 1
3: Rd DPR 0 to FIFO 0
4: Rd DPR 1 to FIFO 0
5: FIFO Task Parameters...
Please enter a selection
****************************************
```

Figure 5-9 FIFO Tasks Menu

FIFO Tasks Menu: FIFO Tasks Menu contains a list of tasks for testing the FIFOs and Dual Port SRAMs found on the Memec RLDRAM board. These tasks consist of transferring data between one of two dual port SRAMs and FIFOs, internal to the FPGA.

```
****************************************
Packet Filter Mode
1: Build Search Trees
2: Build Search Tree
3: Run Oring
4: Ld SRAM Search Pts
5: Run Search File
6: Ld SRAM Ptrs
7: Run Ptr Search
8: Run Search Files
Please enter a selection
****************************************
```

Figure 5-10: Packet Filter Mode Menu

Packet Filter Mode Menu: The Packet Filter Mode Menu contains a list of tasks for testing the PFAAE. In particular it contains tasks to:

1) Build all of the search trees

   In this task all 12 different search trees, described in section 7.3, are built each with 10 different random rule sets.

2) Build Search Tree

   In this task one search tree is built with 10 different random sets of rules.

3) Run ORing

   In this task pre-calculated CBV pointers are read from a file and then passed to the ORing hardware to perform the OR operation. This task is primarily used for testing.

4) Load SRAM Search Points

   In this task search points are loaded from a file on a test computer into SSRAM 1.

5) Run Search File

   In this task search points are read out of SSRAM 1 and then searched. The

search operation consists of first passing the search points to the software. Software then passes the pointers to the CBVs found during the search to the PFAAE. The ORed resultant bit-vector from all of the retrieved bit-vectors is sent back to software to be logged. As well, performance information and B-tree node pointers obtained while searching are all logged for off-line analysis.

6) Load SRAM Pointers

In this task CBV pointers are loaded into SSRAM 1 external to the FPGA on the LT-XC2V6000+ Logic Tile Board. The CBV pointers are obtained from a previous search which logged the pointers during the search operations.

7) Run Pointer Search

In this task software passes the pointers of the CBVs stored in SSRAM 1 to the PFAAE. This task is the same as the Run Search File task except the software portion of the test is not run. Rather the CBV pointers have been pre-calculated and stored into the SSRAM 1. This mode is used purely to obtain accurate performance results for the PFAAE.

8) Run Search Files

This task runs a script of previous tasks to perform all necessary operations and calculate all results required for a particular rule test set. When setup with a rule size, a rule test set and direction it will perform the following operations for each of the ten different rule files.

a. Build all the appropriate tree's and CBVs using the function run_single_build_tree [Table 5-7].

b. Load search points into SSRAM using the function load_search_file_sram [Table 5-7].

c. Run the search using the function run_search_file [Table 5-7].

d. Reload the search pointers using the function load_ptr_file_sram [Table 5-7].

e. Run the pointer search using the function run_ptr_search [Table 5-7].

f. Free up all memory

Upon completion all data will have been logged for later analysis.

```
****************************************
Tests
1: RLDRAM Test
2: DUAL PORT 0 RAM Test
3: DUAL PORT 1 RAM Test
4: Run Batch Command File
5: Run Loop Back Test
6: Run RLDRAM Refresh Test
7: Read All HASPF Regs
8: Run SRAM Tests
9: Run SRAM Write Test
Please enter a selection
****************************************
```

Figure 5-11: Tests Menu

The tests menu contains a list of tests originally run during hardware development. They allow for regression tests to be preformed on the hardware as modifications are made during development. Tests are written for exercising RLDRAM, exercising internal FPGA SRAMs, loop back for board to board communication links, RLDRAM refresh tests, register tests, and external FPGA SSRAM tests.

5.4.3 Software Design Issues

Dynamic Memory Allocation

Dynamic memory allocation turned out to be one of the larger issues when initially designing the software. The C functions Malloc and Free are typically used most often to allocate and de-allocate memory. For the purposes of this software design the blocks of memory required are for the nodes of B-trees and various linked lists. These nodes are quite small, typically less than 50 bytes, and very numerous. The default settings for the built-in Malloc function requires that the smallest block of memory allocated to be significantly larger than the typical size actually required. Although this number can be adjusted to a lower value the associated overhead grows proportionally. Each time Malloc is called a large amount of memory is wasted in overhead and

only a small portion of memory is actually used for storage. To allocate the memory more efficiently and faster special Malloc functions were written to allocate memory for both B-tree nodes and linked list nodes. The speed is primarily improved because the new Malloc functions do not reassemble the freed memory into larger blocks and simply return the block to the available pool of memory. The blocks of memory for nodes and linked lists have the following characteristics used to create new functions:

1) Most of the memory blocks are not freed until the entire structure is destroyed. For example a linked list node is used and not freed until the entire linked list is no longer needed. As well, the number of blocks freed is quite small and can be reused.
2) The entire structure needs to be freed quickly to speed up processing between tests using different parameters.

These two main requirements led to the creation of special functions for allocating and de-allocating memory. Each of the functions had very similar characterizes:

1) **An array of large memory blocks.**

   Each large memory block contains an array of the required memory elements. Figure 5-12 shows the created memory structure. Variables called index and block_index are used to store the current pointer to the next available element and to the current block pointer respectively.

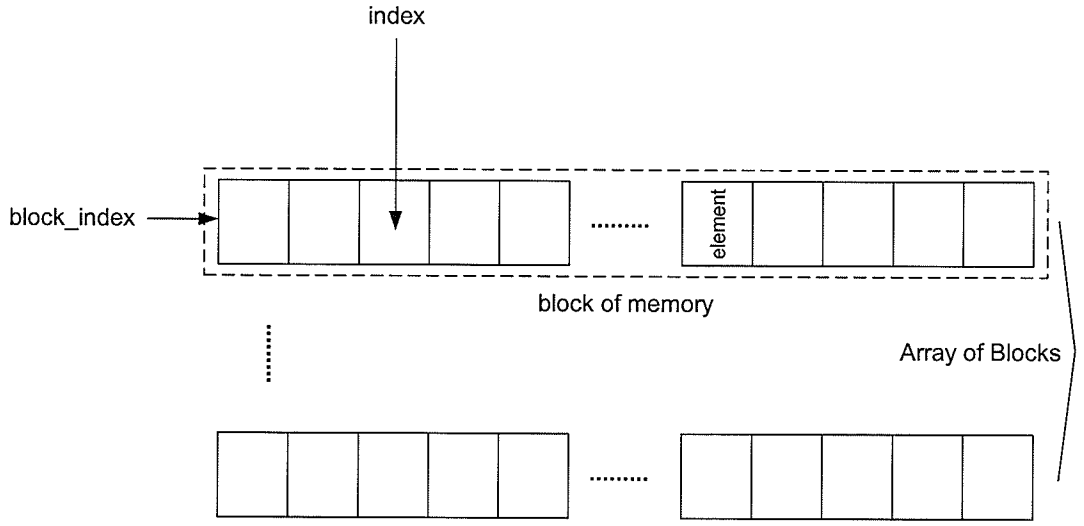

Figure 5-12 : Memory Allocation from an Array of Blocks of Memory

2) **A circular buffer for holding pointers to freed memory elements captured during processing.**

When a new memory element is required this buffer is checked to see if it contains any pointers to available memory before obtaining an element from one of the large blocks of memory. If any memory elements are de-allocated during the course of processing the pointer to this element is inserted into the circular buffer so the memory can be reused. The buffer is circular with a read and write pointer so that pointers to memory elements can be added and removed continuously. The buffer only needs to be as large as the largest amount of memory de-allocated at one time and allows for an efficient way of recycling memory. The circular buffer is shown in Figure 5-13.

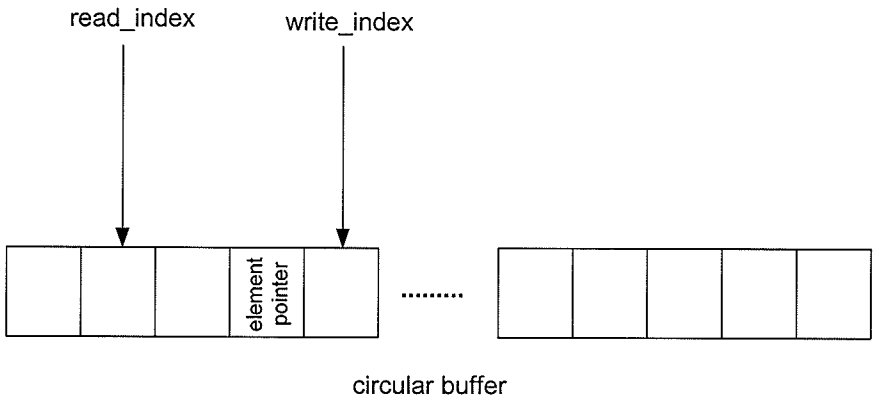

Figure 5-13 : Circular Buffer for Storing De-Allocated Memory Elements

Hardware Software Interface

Communication between software and hardware takes place through the use of command and response FIFOs. When software requires hardware to perform a particular task it writes a command into the command FIFO. In the case when a response is required software waits for an interrupt indicating data is available from the response FIFO. Initially the software was to be designed using an RTOS allowing for preemptive multitasking. The primary reason for this was to make the software more efficient, allowing for task switching while waiting for hardware. Unfortunately the combination of the developed software and available processor turned out to be much slower than the hardware. In particular, it takes the software longer to read the response

buffer than it takes for the hardware to create the response. The software can not even write the commands to the command FIFO as fast as the hardware can process them. Even with little or no processing, just memory reads and writes, the processor can not keep up with the hardware.

To fetch response data an interrupt indicating data is available results in the execution of an interrupt service routine which clears the interrupt and reads out the data. Once again because the hardware is so much quicker than the software, the whole response packet is guaranteed to be in the FIFO by the time the interrupt service routine is executed.

Command and Response Packets

Commands and responses available at a particular time is dependant on the mode of operation the hardware is in. The current mode is controlled by a write to a register over the CSR bus. Available modes include build mode, filter mode, and user command mode. The following is a list of commands and responses available in each mode.

1. **Build Mode** :

In build mode one type of command packet is available for use. This packet contains a packet header, shown in Figure 5-14, and payload data of rule identification numbers. In build mode the final step is for software to send the PFAAE a list of rules contained at each node. The PFAAE then converts this list of rules into a compressed bit-vector and stores it in the RLDRAM. Once completed, the PFAAE returns a pointer to the software identifying the location of the compressed bit-vector. The response packet is shown in Figure 5-14. The parameters used in defining the command and response packets are described below:

a) s : [packet size] indicates the size of the current packet being sent. It has a maximum size of 255 32-bit words including the header.

b) # : [number of rules] indicates the total number of rules contained at the node to be received by the hardware. This is used by hardware so it can keep track of how many rules should be received.

c) p : [RLDRAM pointer]

d) r : [rule identification list]

e) x : [don't care]

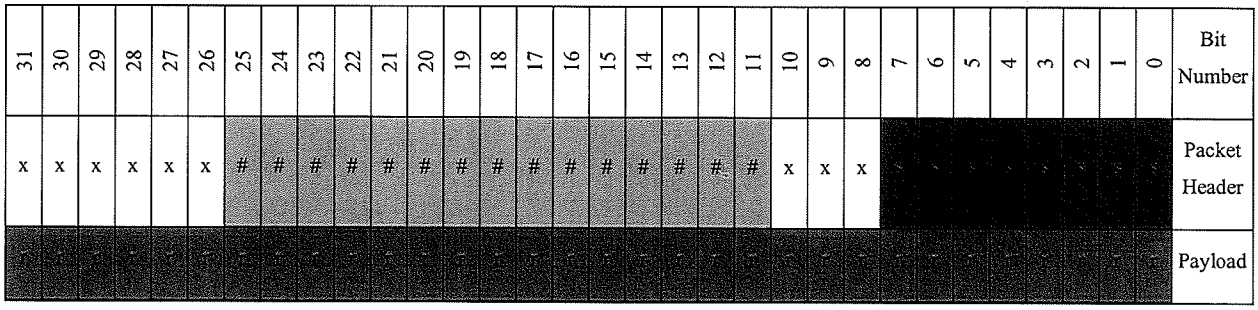

Figure 5-14 : Build Mode Command Packet

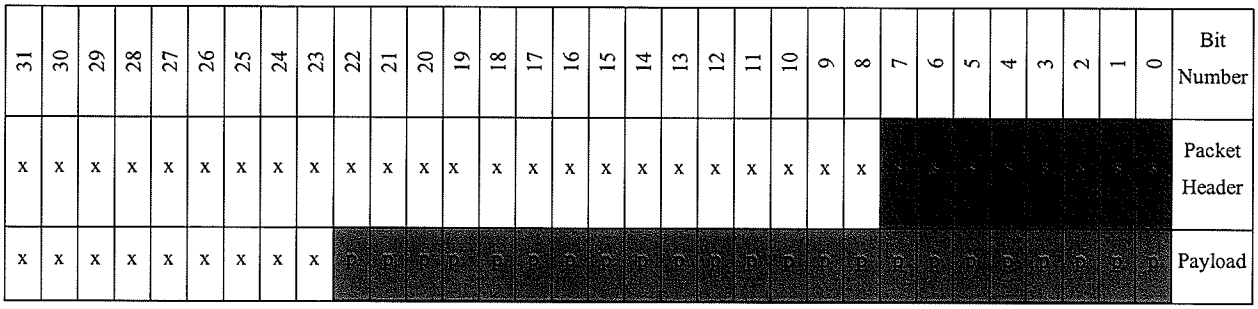

Figure 5-15 : Build Mode Response Packet

2. **Filter Mode** :

In filter mode one type of command packet, shown in Figure 5-16, is available. The purpose of filter mode is to OR together compressed bit-vectors obtained from pointers sent by software. As expected, the packet available in filter mode is used by software to send information about the pointer obtained during a search. It is possible a particular search can find no match at a particular B-tree level so a null pointer flag is inserted into the packet. This indicates to the hardware that no compressed bit-vector has to be retrieved. A flag is used instead of a particular memory value because it was unclear at the start of the design which addresses would be unused. The result of sending four pointers to the PFAAE is a response containing the resultant ORed compressed bit-vector. The response packet is shown in Figure 5-16. The parameters used in defining the command and response packets are described below:

a. n : [null pointer flag] indicates no pointer is found during search of a

particular B-tree level because no range is found containing the search point.

b. p : [pointer value]

c. s : [packet size] indicates the size of the current packet being sent. It has a maximum size of 255 32-bit words including the header.

d. b : [bit-vector] series of bits containing the compressed bit-vector value

e. x : [don't care]

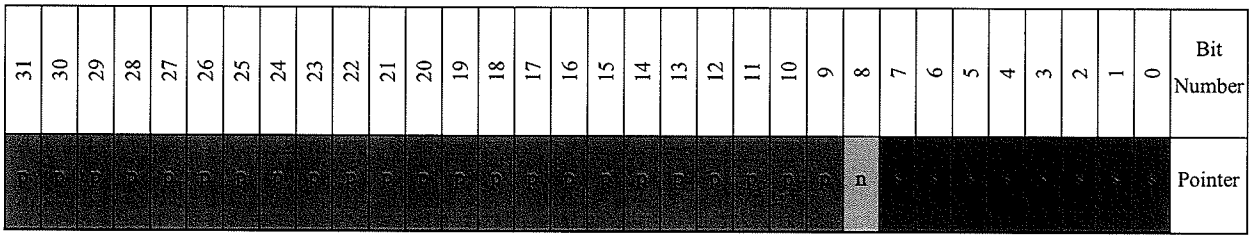

Figure 5-16 : OR Mode Command Packet

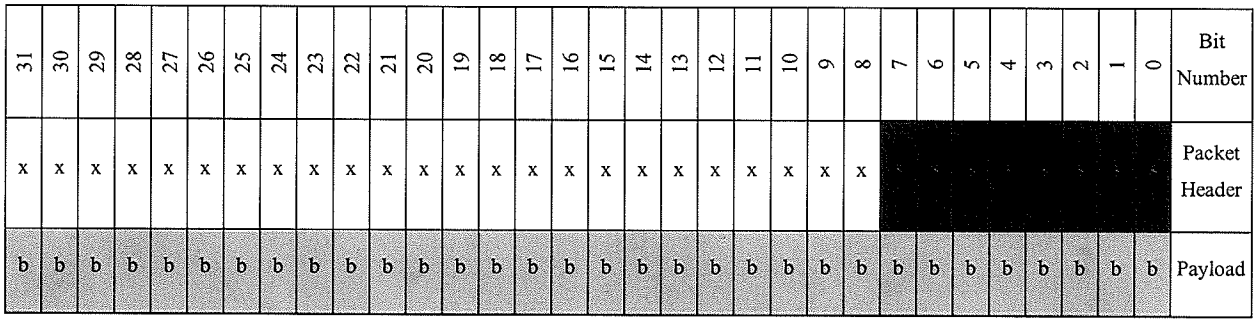

Figure 5-17 : OR Mode Response Packet

3. **User Command Mode** :

User command mode commands are used primarily for testing purposes. These commands allow transfers between the command and response FIFOs, dual port memories and the RLDRAM. For development purposes each command is given a different task number which makes up part of the packet header. These tasks are outlined below.

Transfers between RLDRAM and Dual Port Memory

The first four commands used in user command mode share the same packet

structure and create no response packets. These first four commands are used solely for transferring data between the dual port memories on the RLDRAM board FPGA and the RLDRAM memory. These commands implement a function very similar to a DMA operation. The packet structure of each of these commands is shown in Figure 5-18.

a. Task 1 : read data from dual port RAM 0, write data to RLDRAM
b. Task 2 : read data from dual port RAM 1, write data to RLDRAM
c. Task 3 : read data from RLDRAM, write data to dual port RAM 0
d. Task 4 : read data from RLDRAM, write data to dual port RAM 1

The parameters used in defining the command and response packets are described below:

a. s : [packet size] indicates the size of the current packet being sent. It has a maximum size of 255 32-bit words including the header.
b. a : [dual port RAM address] starting address of the dual port RAM
c. c : [count] count of the number of 64 bit words to transfer
d. t : [task identifier] determines which task to be performed
e. r : [RLDRAM address] starting address of the RLDRAM memory
f. x : [don't care]

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Bit Number
				c	c	c	c	c	c	c	c	c	a	a	a	a	a	a	a	a	x	x	x									Packet Header
x	x	x	x	x	x	x	x	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	Payload

Figure 5-18 : User Command Mode Task 1 - 4 Command Packet

Transfers from FIFO 1 to Dual Port Memory

The next two commands used in user command mode share the same packet structure and create no response packets. These two commands are used solely for transferring

data between the command FIFO and the dual port RAM on the RLDRAM board FPGA. These commands implement a function very similar to a DMA operation. The packet structure of each of these commands is shown in Figure 5-19.

a. Task 5 : read data from FIFO 1, write data to dual port RAM 0

b. Task 6 : read data from FIFO 1, write data to dual port RAM 1

The parameters used in defining the command and response packets are described below:

a. s : [packet size] indicates the size of the current packet being sent. It has a maximum size of 255 32 words including the header.

b. a : [dual port RAM address] starting address of the dual port RAM

c. c : [count] count of the number of 64 bit words to transfer

d. t : [task identifier] determines which task to be performed

e. d : [data to transfer]

f. x : [don't care]

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Bit Number
t	t	t	t	c	c	c	c	c	c	c	c	c	a	a	a	a	a	a	a	a	x	x	x	t	t	t	t	t	t	t	t	Packet Header
d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	Payload

Figure 5-19 : User Command Mode Task 5 - 6 Command Packet

Transfers from Dual Port Memory to FIFO 0

The next two commands used in user command mode share the same packet structure and create response packets. These two commands are used solely for transferring data between the dual port RAM on the RLDRAM board FPGA and the response FIFO. These commands implement a function very similar to a DMA operation. The packet structure of the command and response packets is shown in Figure 5-20 and

Figure 5-21 respectively.

a. Task 7 : read data from dual port RAM 0, write data to FIFO 0

b. Task 8 : read data from dual port RAM 1, write data to FIFO 0

The parameters used in defining the command and response packets are described below:

a. s : [packet size] indicates the size of the current packet being sent. It has a maximum size of 255 32-bit words including the header.

b. a : [dual port RAM address] starting address of the dual port RAM

c. c : [count] count of the number of 64 bit words to transfer

d. t : [task identifier] determines which task to be performed

e. d : [response data]

f. x : [don't care]

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Bit Number
s	s	s	s	c	c	c	c	c	c	c	c	c	a	a	a	a	a	a	a	a	x	x	x	s	s	s	s	s	s	s	s	Packet Header

Figure 5-20 : User Command Mode Task 7 - 8 Command Packet

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Bit Number
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	s	s	s	s	s	s	s	s	Packet Header
d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	Payload

Figure 5-21 : User Command Mode Task 7 - 8 Response Packet

5.4.4 File I/O

Through the Multi-ICE debugging interface code running on the processor has the capability of accessing files on the development computer. This feature of the system proved to be invaluable with regard to inputting test vectors and logging results. The files produced by the software are described in section 6.3.

5.4.5 Software Function Descriptions

The following section provides high level details about the most important software functions built for use in this thesis. The functions are grouped by task function.

Response FIFO Functions

The functions related to the response FIFO are used to manage an array of buffers for holding response packets. The array is managed by two pointers; a read pointer and a write pointer. The read pointer indicates the next buffer to read from and the write pointer indicates the next buffer to insert a response packet into. The difference in values between the two pointers is used to determine the number of response buffers containing pending response data. The functions are outlined in Table 5-1.

Table 5-1 : Response FIFO Functions

Function Name: *Buffer_Free*				
Function Description	**Arguments**	**Description**	**Returns**	**Description**
Determines if a buffer is available to store a packet received from the response FIFO.	*void*		*int*	Returns FAIL if no buffers are available and PASS if buffers are available.

Function Name: *Initilize_Buff_Ptrs*				
Function Description	**Arguments**	**Description**	**Returns**	**Description**
Clears buffer write and read pointer indexes.	*void*		*void*	

Function Name: *Buffers_Pending*				
Function Description	**Arguments**	**Description**	**Returns**	**Description**
Indicates if there are any buffers with response packets waiting to be read.	*void*		*int*	Returns FAIL if no buffers are available with data and PASS if a buffer is available with data.

Function Name: *Read_RD_Rec_Ptr*				
Function Description	**Arguments**	**Description**	**Returns**	**Description**
Reads the current buffer read pointer index.	*void*		*int*	Returns an integer of the index for the current read pointer.

Function Name: *Read_WR_Rec_Ptr*				
Function Description	**Arguments**	**Description**	**Returns**	**Description**
Reads the current buffer write pointer index.	*void*		*int*	Returns an integer of the index for the current write pointer.

(Table continued on next page)

Function Name: *Increment_Rec_Rd_Ptr*				
Function Description	**Arguments**	**Description**	**Returns**	**Description**
Increments the current read pointer index. Used when data is read from a buffer and the read pointer index needs to be incremented.	*void*		*void*	

Function Name: *Increment_Rec_Wr_Ptr*				
Function Description	**Arguments**	**Description**	**Returns**	**Description**
Increments the current write pointer index. Used when data is written into a buffer and the write pointer index needs to be incremented.	*void*		*void*	

Command FIFO Functions

The functions related to the command FIFO are used to create the proper packet headers and payload data for sending command packets to the PFAAE. The commands are formatted according to mode of operation and to ensure large data transfers are broken up into maximum size packets. The functions are outlined in Table 5-2.

Table 5-2 : Command FIFO Functions

Function Name: *Fifo_Write_Line*				
Function Description	**Arguments**	**Description**	**Returns**	**Description**
When passed a buffer of data and a record containing information about the header this function breaks up the data into packets and writes them to the command FIFO. It calls Build_Pkt_Header to write out the appropriate header to the command FIFO.	*data_buff*	Buffer of 32 bit data words.	*void*	
	pkt_header_info	Information used to create the headers for the packets.		
	num_words	Number of words in the data buffer.		

Function Name: *Build_Pkt_Header*				
Function Description	**Arguments**	**Description**	**Returns**	**Description**
When passed packet header information and the packet size this function writes out the proper packet header to the command FIFO. The operating mode is checked to create the proper header for the appropriate mode.	*info*	Pointer to information about the packet header to create.	*void*	
	size	Size of the packet in 32 bit words.		
	rule_size	Number or rules being written out, only applicable to build mode commands. All other commands have pre-determined packet lengths		

Linked List Functions

Linked list functions are created for storing test rules in memory. The rules are stored in a linked list structure if a post build verification of the B-tree structure was to be performed. After the B-tree structures are built the rules contained in the linked list are checked against the B-trees to ensure a proper build operation had been performed. The functions are outlined in Table 5-3.

Table 5-3 : Linked List Functions

Function Name: *Allocate_L_Node_Memory*				
Function Description	**Arguments**	**Description**	**Returns**	**Description**
Allocates memory for a specified number of blocks of memory. Each block contains an array of linked list nodes. The nodes are pre-allocated because the C Malloc function is too inefficient. The minimum size block is much too large to make efficient use of available memory. As such arrays of nodes are pre-allocated then used when the linked list is dynamically created.	*void*		*void*	

Function Name: *Initilize_L_Node_Memory*				
Function Description	**Arguments**	**Description**	**Returns**	**Description**
Initialize the indexes into the arrays of block memory used to allocate and free node memory.	*void*		*void*	

Function Name: *Free_L_Node*				
Function Description	**Arguments**	**Description**	**Returns**	**Description**
Used to return a linked list node to the pool of available node memory.	*ptr*	This function is passed a pointer to the link list node to be freed. It puts this pointer into an array so that it can be reused when a new linked list node is required.	*void*	

Function Name: *Malloc_L_Node*				
Function Description	**Arguments**	**Description**	**Returns**	**Description**
Used to allocate a linked list node of memory from a pool of available linked list nodes. This function will first use memory returned by Free_L_Node and then retrieve from the available blocks of memory.	*void*		*l_node*	l_node is a pointer to an available linked list node of memory.

Function Name: *ll_create*				
Function Description	**Arguments**	**Description**	**Returns**	**Description**
Used to create a new root of a linked list. The root contains a pointer to the head and tail of the linked list.	*void*		*l_root*	l_root is a pointer to the root of a new linked list.

(Table continued on next page)

Function Name: *ll_append*				
Function Description	**Arguments**	**Description**	**Returns**	**Description**
Used to append a new node to a linked list. New node is appended to the end of the list.	*root*	Pointer to the linked list to add node to.	*void*	
	rule_ptr	Pointer to a node to add to the linked list. In the context of this software the node is always a rule.		

Function Name: *ll_print*				
Function Description	**Arguments**	**Description**	**Returns**	**Description**
Used to print out a linked list.	*root*	Pointer to the root of a linked list to print out.	*void*	

Function Name: *ll_delete*				
Function Description	**Arguments**	**Description**	**Returns**	**Description**
Used to free the memory used to create the root of the linked list.	*root*	Pointer to the root of a linked list to be deleted.	*void*	

Function Name: *ll_rule_test*				
Function Description	**Arguments**	**Description**	**Returns**	**Description**
This function is passed a linked list of rules and a 2-dimensional B-tree structure which is supposed to contain a representation of the rules. The tree is checked to ensure it contains each rule properly. This function calls *BTreeRngLvlRuleTest* to ensure the rule has been properly inserted into the tree.	*lvl_tree*	Multilevel B-tree containing all of the rules.	*int*	Returns FAIL if rules have been inserted improperly and PASS if rules have been inserted properly.
	root	Pointer to the root of a linked list containing rules.		

B-tree Point Functions

For the purposes of this thesis two types of B-trees are created. The first is a point B-tree and the second is a range B-tree. The point B-tree is used to store single or point values which do not correspond to a range. In other words these values can be represented by a single point and do not require a start and end value like a range. A point B-tree is used as temporary storage for the rule identifier list found at every second dimension B-tree node elementary internal. This is the list which eventually becomes converted into a compressed bit-vector and replaced with a pointer. The functions used to manage point B-trees are outlined in Table 5-4.

Table 5-4 : Point B-tree Functions

Function Name: *Allocate_PT_Memory*				
Function Description	**Arguments**	**Description**	**Returns**	**Description**
Allocates memory for a specified number of blocks of memory. Each block contains an array of B-tree nodes or B-trees. The nodes are pre-allocated because of the C Malloc function is to inefficient. The minimum size block is much too large to make efficient use of available memory. As such, arrays of nodes are pre-allocated then used when the B-tree is dynamically created.	*void*		*void*	

Function Name: *Initilize_Pt_Memory*				
Function Description	**Arguments**	**Description**	**Returns**	**Description**
Initialize the indexes into the arrays of block memory used to allocate and free node and B-tree memory.	*void*		*void*	

Function Name: *Free_Pt_Tree*				
Function Description	**Arguments**	**Description**	**Returns**	**Description**
Used to return a B-tree to the pool of available node memory.	*ptr*	This function is passed a pointer to the B-tree to be freed. This function then goes and puts this pointer into an array so that it can be reused when a new B-tree is required.	*void*	

Function Name: *Malloc_Pt_Tree*				
Function Description	**Arguments**	**Description**	**Returns**	**Description**
Used to allocate a node of memory from a pool of available B-trees. This function will first use memory returned by Free_Pt_Tree and then retrieve from the available blocks of memory.	*void*		*b_tree_pt*	b_tree_pt is a pointer to an available B-tree.

Function Name: *Free_Pt_Node*				
Function Description	**Arguments**	**Description**	**Returns**	**Description**
Used to return a B-tree node to the pool of available node memory.	*ptr*	This function is passed a pointer to the B-tree node to be freed. This function then goes and puts this pointer into an array so that it can be reused when a new node is required.	*void*	

Function Name: *Malloc_Pt_Node*				
Function Description	**Arguments**	**Description**	**Returns**	**Description**
Used to allocate a node of memory from a pool of available B-tree nodes. This function will first use memory returned by Free_Pt_Node and then retrieve from the available blocks of memory.	*void*		*b_tree_pt_node*	b_tree_pt_node is a pointer to an available B-tree node of memory.

(Table continued on next page)

Function Name: *BTreePtCreate*				
Function Description	**Arguments**	**Description**	**Returns**	**Description**
Used to create a B-tree and its root node. The root node is the first node inserted into the B-tree. Upon insertion the root node is set to be a leaf node with no keys.	*void*		*b_tree_pt*	b_tree_pt is a pointer to a newly created B-tree.

Function Name: *BTreePtInsert*				
Function Description	**Arguments**	**Description**	**Returns**	**Description**
Used to insert a new point rule value into a B-tree. This function checks to ensure the root node is not full before inserting a value. If the root node is full a new root node is created and a BTreePtSplitChild operation is performed on the old root node.	*tree*	Pointer to the B-tree to insert the new point into.	*void*	
	rule_val	Integer value of the rule identification number, referred to as a point, to be inserted into the tree		

Function Name: *BTreePtSplitChild*				
Function Description	**Arguments**	**Description**	**Returns**	**Description**
This function splits a full B-tree node y in the middle inserting y's middle key into the i^{th} key position in x. Half of y's keys are inserted into a new node and half remain in the old node. See [9] for more details.	*x*	x is a pointer to a B-tree node which has a child node which is full.	*void*	
	i	i is the index into an array of child pointers in x which points to a full B-tree node.		
	y	y is a pointer to the full B-tree node to be split.		

Function Name: *BTreePtInsertNonfull*				
Function Description	**Arguments**	**Description**	**Returns**	**Description**
This function inserts a point or value into a non full B-tree. Non full indicates the B-tree root node has room for at least one key. Room for one key is required because of the split operation. The split operation pushes one of the keys from a full node up the B-tree into the parent node when a split is performed. At least one free key is required in the root node to ensure when a split operation occurs space is available in the root node. See [9] for more information.	*tree*	tree is a pointer to a B-tree to insert the point into.	*void*	
	x	x is a pointer to a B-tree node which is the current location in the B-tree which the insert algorithm is looking to insert the new value.		
	rule_val	rule_val is the value attempting to be inserted into the B-tree.		

Function Name: *BTreePtCopy*				
Function Description	**Arguments**	**Description**	**Returns**	**Description**
This function copies a B-tree and returns a pointer to the copy.	*tree*	tree is a pointer to a B-tree to copy.	*b_tree_pt*	b_tree_pt is a pointer to the copy created.

(Table continued on next page)

Function Name: *BTreePtPtrCopy*				
Function Description	**Arguments**	**Description**	**Returns**	**Description**
This function copies a B-tree node and then recursively copies its children.	*node*	node is a pointer to a node to copy.	*b_tree_pt_node*	b_tree_pt_node is a pointer to copy created.

Function Name: *MergeBTreePt*				
Function Description	**Arguments**	**Description**	**Returns**	**Description**
This function merges the nodes from tree2 into tree1.	*tree1*	tree1 is a pointer to a B-tree to insert nodes from tree2 into.	*void*	
	tree2	tree2 is a pointer to a B-tree to copy nodes from to insert into tree1.		
	remove	Indicates if tree2 should be deleted after the merge operation.		

Function Name: *MergeNodeBTreePt*				
Function Description	**Arguments**	**Description**	**Returns**	**Description**
This function inserts a node into a tree.	*tree*	tree is a pointer to a B-tree to insert the new node into.	*void*	
	node	node is pointer to a B-tree node.		
	remove	Indicates if node should be deleted after the merge operation.		

Function Name: *BTreePtPrint*				
Function Description	**Arguments**	**Description**	**Returns**	**Description**
This function copies all of the values in a B-tree into an array in ascending order. This function is used to copy all of the rules at a B-tree range node and then send them to the ORing hardware. It is recursively called to obtain all of the values in the point B-tree.	*node*	node is the current pointer in the B-tree being printed out.	*int*	Count of all of the values in the B-tree.
	level	Unused variable for future use.		
	buff	Array passed recursively to the function to store all of the values in the tree.		

Function Name: *BTreePtNodeFree*				
Function Description	**Arguments**	**Description**	**Returns**	**Description**
This function frees the memory from a B-tree node and its children.	*node*	node is pointer to a B-tree node	*void*	

Function Name: *BTreePtFree*				
Function Description	**Arguments**	**Description**	**Returns**	**Description**
This function frees the memory used by a B-tree	*tree*	tree is a pointer to a B-tree	*void*	

(Table continued on next page)

Function Name: *BTreePtSearchPoint*				
Function Description	**Arguments**	**Description**	**Returns**	**Description**
This function searches a B-tree to find a particular value. It is recursively called to search from one node to the next.	*x*	Pointer to the current B-tree node being searched.	*int*	PASS or FAIL indication of whether or not the search value was found.
	point	Value to search for in the B-tree.		
	count	Pointer to an integer to keep track of how many keys have been searched against.		

B-tree Range Functions

The second type of B-tree created is used for storing ranges. The functions are used to manage range B-trees are outlined in Table 5-5.

Table 5-5 : B-tree Range Functions

Function Name: *Allocate_Rng_Node_Memory*				
Function Description	**Arguments**	**Description**	**Returns**	**Description**
Allocates memory for a specified number of blocks of memory. Each block contains an array of B-tree nodes or B-trees. The nodes are pre-allocated because of the C Malloc function is to inefficient. The minimum size block is much too large to make efficient use of available memory. As such, arrays of nodes and trees are pre-allocated then used when the B-tree is dynamically created.	*void*		*void*	

Function Name: *Initilize_Rng_Node_Memory*				
Function Description	**Arguments**	**Description**	**Returns**	**Description**
Initialize the indexes into the arrays of block memory used to allocate and free node and B-tree memory.	*void*		*void*	

Function Name: *Free_Rng_Node*				
Function Description	**Arguments**	**Description**	**Returns**	**Description**
Used to return a node to the pool of available node memory.	*ptr*	This function is passed a pointer to the B-tree node to be freed. It puts this pointer into an array so that it can be reused when a new node is required.	*void*	

(Table continued on next page)

Function Name: *Malloc_Rng_Node*				
Function Description	**Arguments**	**Description**	**Returns**	**Description**
Used to allocate a node of memory from a pool of available B-tree nodes. This function will first use memory returned by Free_Rng_Node and then retrieve from the available blocks of memory.	*void*		*b_tree_rng_node*	b_tree_rng_node is a pointer to an available B-tree node of memory.

Function Name: *BTreeRngCreate*				
Function Description	**Arguments**	**Description**	**Returns**	**Description**
Used to create a B-tree and its root node. The root node is the first node inserted into the B-tree. Upon insertion the root node is set to be a leaf node with no keys.	*void*		*b_tree_rng*	b_tree_rng is a pointer to a newly created B-tree.

Function Name: *BTreeRngTreeDestroy*				
Function Description	**Arguments**	**Description**	**Returns**	**Description**
Free the memory used to create the range B-tree.	*tree*	Pointer to the range B-tree to be freed.	*void*	

Function Name: *BTreeRngInsert*				
Function Description	**Arguments**	**Description**	**Returns**	**Description**
Used to insert a new 2-dimensional range rule value into a 2-dimensional B-tree. This function checks to ensure the root node is not full before inserting a value. If the root node is full a new root node is created and a BTreeRngSplitChild operation is performed on the old root node.	*tree*	Pointer to the B-tree to insert the new range into.	*void*	
	level	Used to keep track of which level is being inserted into, not currently used.		
	dimen	Used to keep track of which dimension is being inserted into.		
	rule_ptr	Pointer to a structure containing a range		

Function Name: *BTreeRngSplitChild*				
Function Description	**Arguments**	**Description**	**Returns**	**Description**
This function splits a full B-tree node y in the middle inserting y's middle key into the i^{th} key position in x. Half of y's keys are inserted into a new node and half remain in the old node. See [9] for more details.	*x*	x is a pointer to a B-tree node which has a child node which is full.	*void*	
	i	i is the index into an array of child pointers in x which points to a full B-tree node.		
	y	y is a pointer to the full B-tree node to be split.		

(Table continued on next page)

Function Name: *BTreeRngInsertNonfull*				
Function Description	**Arguments**	**Description**	**Returns**	**Description**
This function inserts a 2-dimensional range value into a non full 2-dimensional B-tree. Non full indicates the B-tree root node has room for at least one key. Room for one key is required because of the split operation. The split operation pushes one of the keys from a full node up the B-tree into the parent node when a split is performed. At least one free key is required in the root node to ensure when a split operation occurs space is available in the root node. See figure [9] for more information.	*tree*	tree is a pointer to a B-tree to insert the point into.	*void*	
	level	Used to keep track of which level is being inserted into, not currently used.		
	dimen	Used to keep track of which dimension is being inserted into.		
	x	x is a pointer to a B-tree node which is the current location in the B-tree which the insert algorithm is looking to insert the new value.		
	rule_ptr	pointer to a structure containing a range		

Function Name: *BTreeRngSearch*				
Function Description	**Arguments**	**Description**	**Returns**	**Description**
This function searches a range tree for a point.	*x*	x is the current pointer in the B-tree being searched.	*void**	Pointer to the compressed bit-vector in RLDRAM memory.
	point	Value to be searched for.		
	count	Used to keep track of the number of key comparisons done during the search operation.		
	node_count	Used to keep track of the number of B-tree nodes accessed during the search operation.		

Function Name: *BTreeRngCreateCBV*				
Function Description	**Arguments**	**Description**	**Returns**	**Description**
This function calls the functions needed to create compressed bit-vectors for every rule identification list.	*node*	node is the current pointer in the B-tree being printed out.	*void*	
	dimen	Used to keep track of which dimension is being operated on.		
	level	Unused variable for future use.		
	count	used to keep track of the number of compressed bit-vectors created		

(Table continued on next page)

Function Name: *BTreeRngPrint*				
Function Description	**Arguments**	**Description**	**Returns**	**Description**
This function prints out the contents of a range tree.	*node*	node is the current pointer in the B-tree being printed out.	*void*	
	dimen	Used to keep track of which dimension is being operated on.		
	level	Unused variable for future use.		
	node_count	Used to keep track of the number of B-tree nodes in the B-tree structure		
	num_key	Used to keep track of the number total number of keys used for all of the nodes in the B-tree structure.		
	point_count	Used to keep track of rule identifier counts while printing.		

Function Name: *BTreeRngCopy*				
Function Description	**Arguments**	**Description**	**Returns**	**Description**
This function copies a B-tree and returns a pointer to this copy.	*tree*	tree is a pointer to a B-tree to copy.	*b_tree_rng*	b_tree_rng is a pointer to the copy created.

Function Name: *BTreeRngPtrCopy*				
Function Description	**Arguments**	**Description**	**Returns**	**Description**
This function copies a B-tree node and then recursively copies its children.	*parent*	Pointer to the parent node of the node being copied.	*b_tree_rng_node*	b_tree_rng_node is a pointer to copy created.
	dimen	Used to keep track of which dimension is being operated on.		
	node	Pointer to the node to be copied.		

Function Name: *MergeBTreeRng*				
Function Description	**Arguments**	**Description**	**Returns**	**Description**
This function merges the nodes from tree2 into tree1	*level*	Used to keep track of which level is being operated on.	*void*	
	dimen	Used to keep track of which dimension is being operated on.		
	tree1	tree1 is a pointer to a B-tree to insert nodes from tree2 into.		
	tree2	tree2 is a pointer to a B-tree to copy nodes from to insert into tree1.		
	remove	Indicates if tree2 should be deleted after the merge operation.		

(Table continued on next page)

Function Name: *MergeNodeBTreeRng*				
Function Description	**Arguments**	**Description**	**Returns**	**Description**
This function inserts a node into tree.	*level*	Used to keep track of which level is being operated on.	*void*	
	dimen	Used to keep track of which dimension is being operated on.		
	tree	tree1 is a pointer to a B-tree to insert nodes from tree2 into.		
	node	node is pointer to a B-tree node.		
	remove	Indicates if node should be deleted after the merge operation.		

B-tree Range Level Functions

B-tree range level functions are created to manage the multi-level B-tree structure. The functions outlined in Table 5-6 are used to create and perform operations on all four levels of the B-tree structure.

Table 5-6 : B-tree Range Level Functions

Function Name: *BTreeRngLvlCreate*				
Function Description	**Arguments**	**Description**	**Returns**	**Description**
Used to create the four B-trees which make up the multi-level B-tree structure.	*void*		*b_tree_rng_lvl*	b_tree_rng_lvl is a pointer to a newly created multi-level B-tree.

Function Name: *BTreeRndLvlDestroy*				
Function Description	**Arguments**	**Description**	**Returns**	**Description**
Free the memory used to create the multi-level B-tree.	*tree_rng_lvl_ptr*	Pointer to multi-level B-tree structure.	*void*	

Function Name: *BTreeRngLvlInsert*				
Function Description	**Arguments**	**Description**	**Returns**	**Description**
Used to determine which of the four levels a 2-dimensional rule should be inserted into. Determination is made based on the range width of the first dimension.	*lvl_tree*	Pointer to multi-level B-tree structure.	*void*	
	rule_ptr	Pointer to a structure containing a 2-dimensional rule.		

(Table continued on next page)

Function Name: *BTreeRngLvlPrint*				
Function Description	**Arguments**	**Description**	**Returns**	**Description**
This function prints out the contents of a multi-level B-tree.	*tree*	tree is a pointer to a multi-level B-tree structure to be printed out.	*void*	
	node_count	Used to keep track of the number of B-tree nodes in the multi-level B-tree structure		
	num_key	Used to keep track of the number total number of keys used for all of the nodes in the multi-level B-tree structure.		
	point_count	Used to keep track of rule identifier counts while printing.		

Function Name: *BTreeRngLvlCreateCBV*				
Function Description	**Argument**	**Description**	**Returns**	**Description**
This function calls the functions need to create compressed bit-vectors for the four levels of B-trees.	*tree*	tree is a pointer to a multi-level B-tree structure for which the CBV will be created for.	*void*	

Function Name: *BTreeRndLvlSearch*				
Function Description	**Arguments**	**Description**	**Returns**	**Description**
This function searches the multi-level B-tree structure for a 2-dimensional point. The compressed bit-vector pointers found during the search as sent the PFAAE to perform the ORing operation.	*lvl_tree*	tree is a pointer to a multi-level B-tree structure to be searched.	*void*	
	dim1	Value to be searched for in first dimension.		
	dim2	Value to be searched for in second dimension.		
	count	Used to keep track of the number of key comparisons done during the search operation.		
	node_count	Used to keep track of the number of B-tree nodes accessed during the search operation.		

Function Name: *BTreeRngLvlRuleTest*				
Function Description	**Arguments**	**Description**	**Returns**	**Description**
Used to check if a 2-dimensional rule has been properly inserted into the B-tree after the structure is finished being built.	*lvl_tree*	Pointer to multi-level B-tree structure.	*int*	Returns a PASS or a FAIL depending on if the rule has been inserted properly.
	test_rule	Pointer to a structure containing a 2-dimensional rule to be checked.		

Menu Item Functions

Menu item functions are created to run high level tasks for running tests to create results files. The functions are outlined in Table 5-7.

Table 5-7 : Menu Item Functions

Function Name	Function Description
run_searchfiles	This function runs a script of functions to perform all necessary operations and calculate all results required for a particular rule test set. When setup with a rule size, a rule test set and direction it will perform the following operations for each of the ten different rule sets: a. Build all the appropriate B-tree structure and CBVs using the function run_single_build_tree. b. Load search points into SSRAM using the function load_search_file_sram. c. Run the search using the function run_search_file. d. Reload the search pointers using the function load_ptr_file_sram. e. Run the pointer search using the function run_ptr_search. f. Free up all memory Upon completion all data will have been logged for later analysis.

Function Name	Function Description
run_single_build_tree	This function builds a single multi-level B-tree based on a rule-set provided. After the B-tree structure is complete compressed bit-vectors are created and the end result is ready to perform search operations.

Function Name	Function Description
load_search_file_sram	In this function search points are loaded from a file on a test computer into SSRAM 1.

Function Name	Function Description
load_ptr_file_sram	In this function CBV pointers are loaded into SSRAM 1 external to the FPGA on the LT-XC2V6000+ Logic Tile Board. The CBV pointers are obtained from a previous search which logged the pointers during the search operations.

Function Name	Function Description
run_ptr_search	In this function software passes the pointers of the CBVs stored in SSRAM 1 to the PFAAE. This function is the same as the run_search_file function except the software portion of the test is not run. Rather the CBV pointers have been pre-calculated and stored into the SSRAM 1. This mode of operation is used purely to obtain accurate performance results for the PFAAE.

Function Name	Function Description
run_search_file	In this function search points are read out of SSRAM 1 and then searched. The search operation consists of first passing the search points to a software search function. Software then passes the pointers to the CBVs found during the search to the PFAAE. The ORed resultant bit-vector from all of the retrieved bit-vectors are logged. As well, performance information and B-tree node pointers obtained while searching are all logged for later analysis.

Function Name: Create_CBV				
Function Description	**Arguments**	**Description**	**Returns**	**Description**
Sends a buffer of rules to the PFAAE to build a compressed bit-vector, waits for the response pointer and then logs the resultant bit-vector using the Read_CBV function.	buff	Buffer containing a list of rule identifiers used to create the compressed bit-vector.	int	Returns a PASS if successful and FAIL if not.
	cbv_pointer	Pointer in RLDRAM to the created compressed bit-vector.		

: (Table continued on next page)

Function Name: Read_CBV				
Function Description	**Arguments**	**Description**	**Returns**	**Description**
Reads a compressed bit-vector located at a pointer in RLDRAM and logs the result to a file.	ptr	Pointer in RLDRAM to read compressed bit-vector from.	int	Returns a PASS if successful and FAIL if not.

Function Name	**Function Description**
run_dpr1_test	This function is used to write data to the dual port RAM 1 and then read it back and perform a comparison.

Function Name	**Function Description**
run_dpr0_test	This function is used to write data to the dual port RAM 0 and then read it back and perform a comparison.

Function Name	**Function Description**
run_rldram_test	This function is used to test that the RLDRAM controller and wrapper is working properly. The test is performed in a number of stages. Firstly data is written to dual port RAM 0, next this data is written to RLDRAM, next the data is written from the RLDRAM to dual port RAM 1 and finally the data is returned from dual port RAM 1 for comparison with the sent data.

Function Name: TestSsram				
Function Description	**Arguments**	**Description**	**Returns**	**Description**
Test SSRAM functionality by writing and reading a test pattern to and from memory	start	Starting address of memory to test.	int	TRUE is returned if an error is found, FALSE is returned otherwise.
	end	Ending address of memory to test.		
	ltbase	Base address of memory to test.		

6 Verification

6.1 Hardware Verification

Hardware verification was done in an incremental fashion testing the basic building blocks of the system until all of the logic components and buses were fully verified. The time required for testing was somewhat underestimated in the project and took longer than expected. The following are the major elements tested during hardware verification:

1. **LVDS Serial Bus Communication Testing**: *Low-Voltage Differential Signaling* (LVDS) serial bus testing was the primary test item which took longer than expected. The two primary reasons for this was inadequate equipment and inexperience in the determination of appropriate Virtex II input/output block delays. Without having a logic analyzer there is no effective method of viewing the serial data stream during testing and determining delay adjustments. The Virtex II FPGA allows for precise adjustments to be made to the delays of outputs. These delays are adjusted until no bit errors are found in a loop back test of the serial bus. Unfortunately, the process used to determine appropriate delay values involves tweaking the values until the serial bus operated properly. As such, the tweaking process is quite time consuming.

2. **System Communication Testing** (Loop Back): Once reliable serial bus transmit and receive functionality was proven, loop back testing was performed with the rest of the communication components. The loop back test consists of software writing data to the command FIFO, which is then sent to the RLDRAM FPGA and then looped back into the response FIFO. The software then checks that the data received to ensure it is the same as data sent.

3. **Memory Testing**: Memory testing involves transfers of data between the dual port memories internal to the RLDRAM FPGA and the RLDRAM memory. These tests ensure proper operation of the RLDRAM wrapper, RLDRAM controller, memory refresh operations, and dual port RAM memories.

6.2 Software Verification

For the most part the software verification was done at the same time as the hardware verification. Co-verification of hardware and low level software drivers was almost always done

to ensure both items functioned properly. The one major exception to this was the verification of the B-tree structure. At the time of B-tree construction each input rule from the test set was added to a linked list for a final post processing step. In this step each field of every rule was double checked via a search in the B-tree structure to ensure its entire range is covered properly. As well, each second dimension elementary interval, a range was apart of, was double checked to ensure the rule number was apart of the node rule list. In this way the B-tree was ensured to be built correctly.

6.3 Packet Filter Algorithm Verification

In order for a smooth hardware and software integration a collaborative flow was developed. This flow was used to verify the functionality and evaluate the performance of the packet filter design. The flow is presented as a series of figures, each containing one portion of the integration phase. At the left side of each figure a box shows the input files needed for the particular operation, shown by the box with rounded edges in the center. On the right side of the diagram the output files produced are listed. Each diagram illustrates the flow of operations from top to bottom and includes labels to show the flow from one diagram to the next.

Asterisks are used to delimit the location of variables inserted into the file names. Most of the files are appended with the mode, seed and rule size of the test to organize the results. Mode refers to the 2-dimensional pair under test, seed refers to the seed used for the test and rule size indicates the number of rules used for the test. These variables were inserted to make the results files identifiable.

6.3.1 Step 1 : Rule Generation

To verify and benchmark the packet filter, realistic rules sets needed to be acquired or developed. It was found to be extremely difficult to acquire rules sets to test against. Unfortunately, unlike compression testing there are no corpus' available providing commercial rules sets. However, as part of the project a C++ tool was developed capable of generating test rule-sets of varying size and distribution statistics. Figure 6-1 shows a picture of the created tool and Figure 6-2 shows a basic flow of the rule generation software.

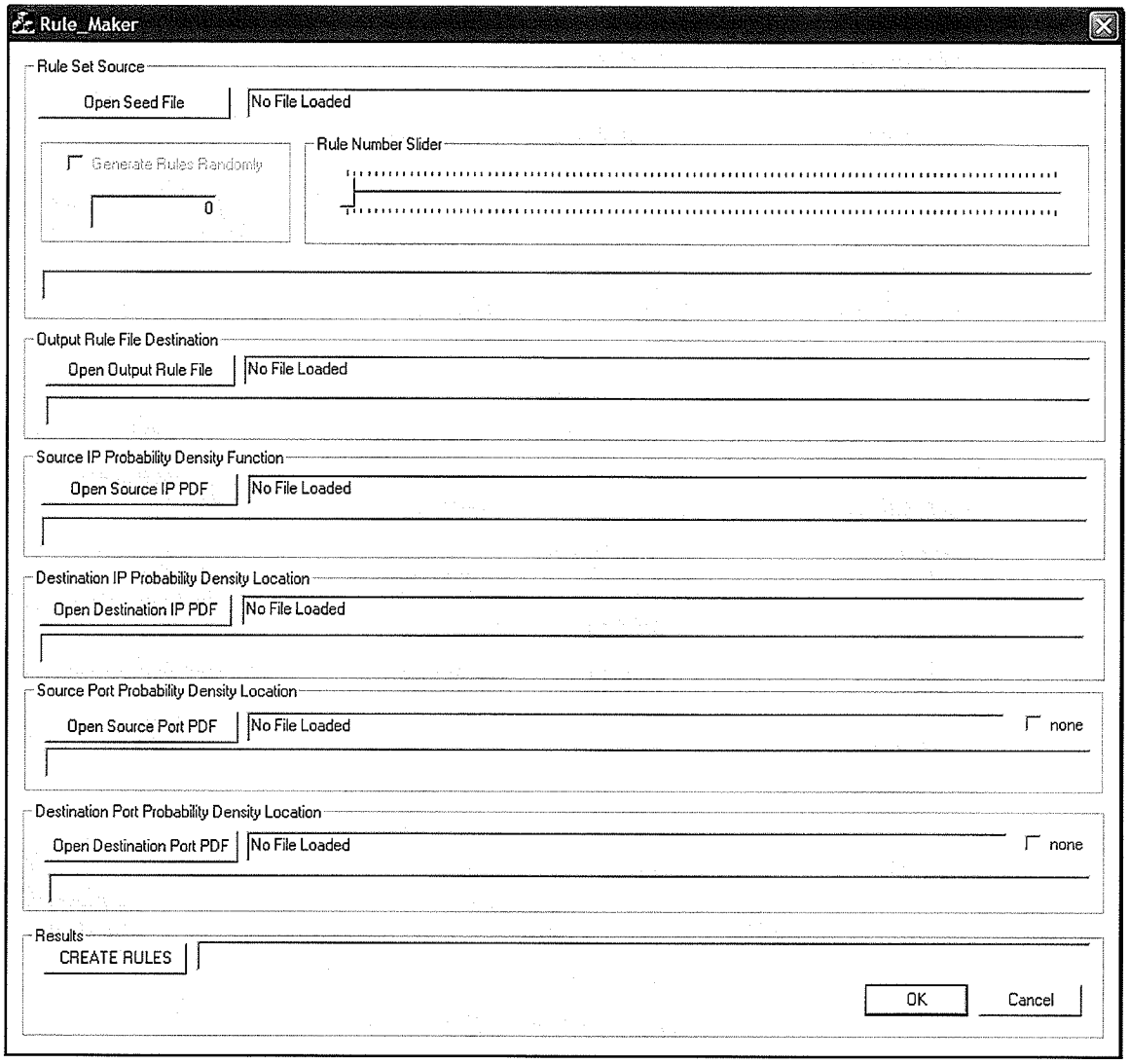

Figure 6-1 : Rule Generator Software

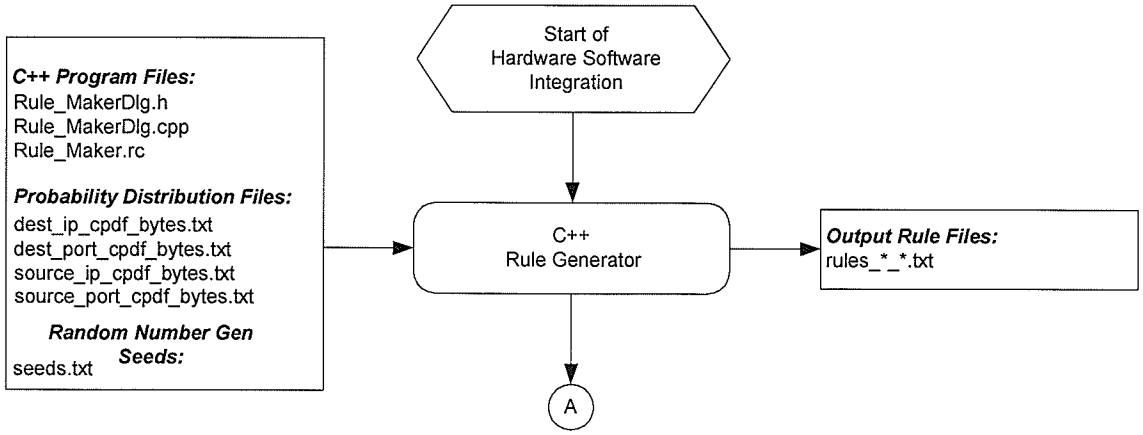

Figure 6-2 : Rule Generation Files and Process for Inbound Rules Example

Rule Generation Input Files Descriptions

1. C++ Program Files:
    a. **Rule_MakerDlg.h**: Contains constants used in creating a rule generation executable.
    b. **Rule_MakerDlg.cpp**: Contains classes and functions used in creating a rule generation executable.
    c. **Rule_Maker.rc**: Resource file containing project parameters for the *Graphical User Interface* (GUI).
2. Discrete Cumulative Probability Distribution Files:
    a. **dest_ip_cpdf_bytes.txt**: Defines the destination IP probability distribution.
    b. **source_ip_cpdf_bytes.txt**: Defines the source IP probability distribution.
    c. **dest_port_cpdf_bytes.txt**: Defines the destination port probability distribution.
    d. **source_port_cpdf_bytes.txt**: Defines the source port probability distribution.
3. Random Number Generation Seed File
    a. **seeds.txt**: Holds ten different seeds for the random number generator. The ten different seeds are used to generate ten different files required for creation of a confidence interval.

Rule Generation Output Files

1. **rules_*model*_*seed*.txt:** Stores a set of rules for a particular seed and rule model. A rule model consists of the set of four probability distribution files outlined above.

Rule Generation Process

The steps performed when generating a set of rules are as follows:

1. Select the appropriate seed file so that ten different rule files can be created, each with a different seed.
2. Select the number of output rules desired for testing.
3. Select a base file to output the rules into. The output of the rules then appears in ten different files with the prefix of each rule file being the name of the base file.
4. Select the four distribution files describing the distributions for the source and destination IP and port protocols.
5. Press Create Rules and the ten rule files are created.

An example of how a rule is stored in a file is shown below (numeric values shown in hexadecimal format):

```
dd07496b : Source IP (32 bit value)
b389185b : Destination IP (32 bit value)
81b00000 : Bits 31:26 Mask Length for source IP (32 in this case)
           Bits 25:20 Mask Length for destination IP (27 in this
           case)
           Bits 19:0 Rule ID number
0000ffff : Source Port (start of range is first 16 MSB, end is LSB
           16)
006a006a : Destination port (start of range is first 16 MSB, end is
           LSB 16)
```

The rule format allows ports to be represented using the range matching style and IPs to be represented using the prefix matching style. Having the start and the end specified allows a port rule to be specified as an exact value or a particular range. Likewise the five bits used to specify the mask for an IP rule allow for specification of a single IP value or any one of the 32 prefix lengths. A value of 32 for the IP mask specifies a single IP and a mask length of zero represents a wildcard.

6.3.2 Step 2 : Building the Search Structures and CBVs

The second step in the verification flow is building the search trees and the CBVs. The rule files are used as inputs to the software to produce 2-dimensional B-trees containing the rule lists for the creation of CBVs. A flow diagram for step two is shown in Figure 6-3.

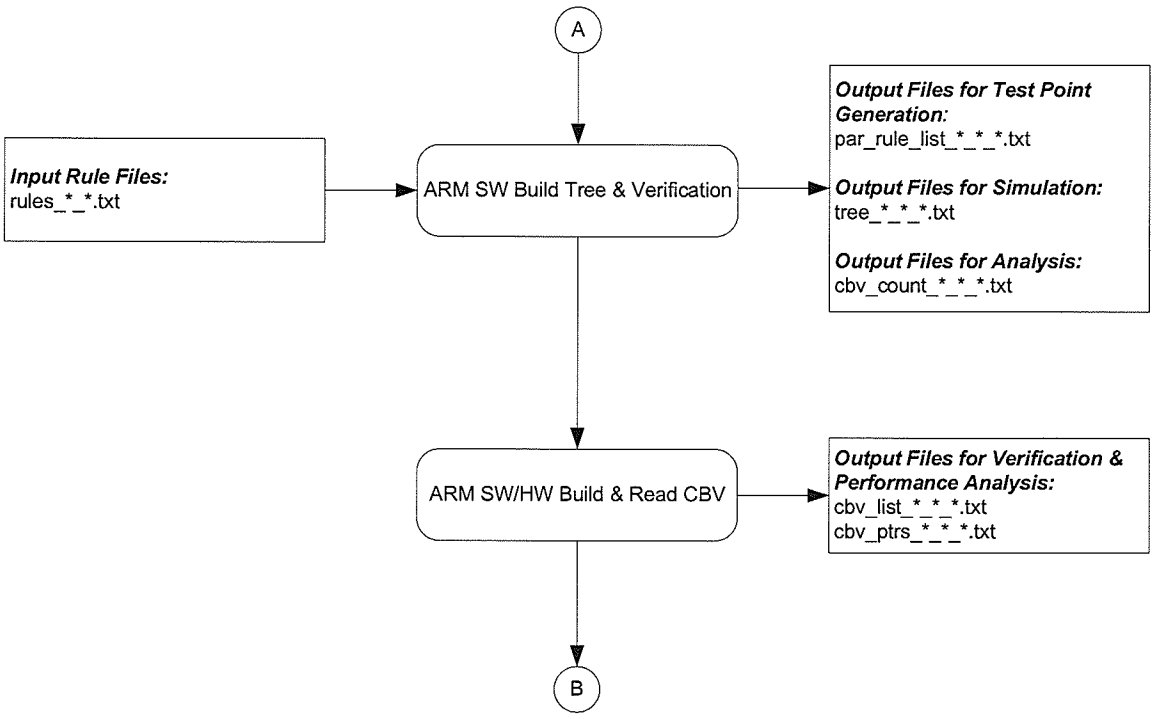

Figure 6-3 : Build Tree and CBVs

Build Tree File Descriptions

1. **par_rule_list_*mode*_*seed*_*rule_size*.txt :** The purpose of this file is to store parsed rules with the start and end of the each of the selected two dimensions. This file is then used as an input to a Tcl script to produce random test points in the specified ranges of each of the rules. This ensures that a number of test points are created for each rule. See Appendix A for a file example.

2. **tree_*mode*_*seed*_*rule_size*.txt :** Contains a text based representation of the search tree. It provides the start and end of each elementary interval in the tree as well as a count and list of rules. See Appendix A for a file example. This was an optional file generated for simulation and Tcl validation.

3. **cbv_count_*mode*_*seed*_*rule_size*.txt:** Contains a list of the number of rules encompassed within each CBV. See Appendix A for a file example.

4. **cbv_list_*mode*_*seed*_*rule_size*.txt :** Contains a list of the CBVs one for each elementary interval in the tree. See Appendix A for a file example.

5. **cbv_ptrs_*mode*_*seed*_*rule_size*.txt:** Contains a list of the RLDRAM pointers one for each CBV. See Appendix A for a file example.

6.3.3 Step 3 : Producing Test Files

In this stage of the verification flow Tcl is used to create a set of stimulus files to exercise the packet filter. Multiple test points are generated from each rule to ensure each rule is tested. The test points are also used with the rule-set to perform a basic linear search. The results of this search are used to validate the final results produced by the packet filter.

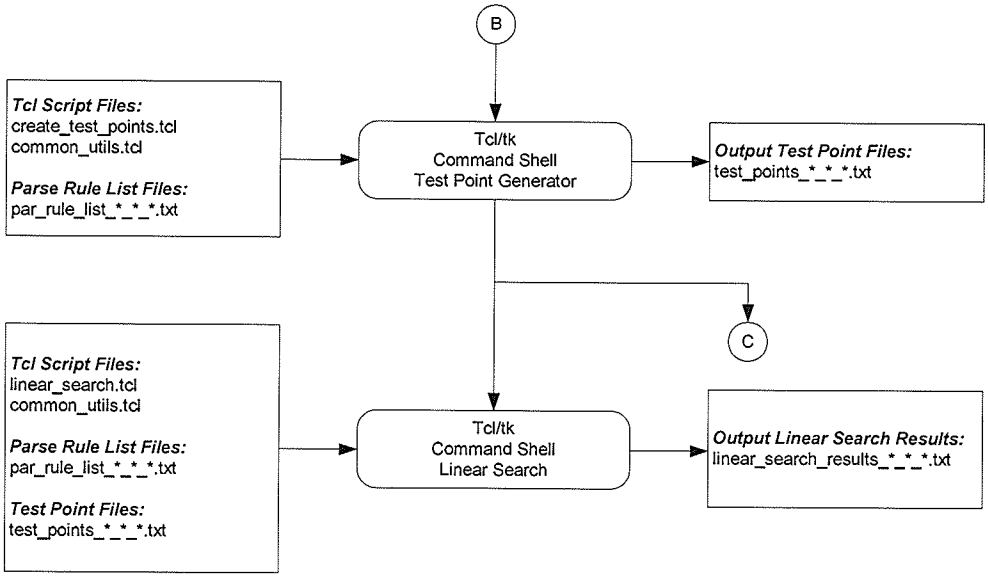

Figure 6-4 : Tcl Operations Linear Search Files and Process

File Descriptions

1. **common_utils.tcl :** Contains common utilities and file I/O procedures used in Tcl packet filter operations.

2. **create_test_points.tcl** : Creates a file of test points for each mode, seed and rule size of a given rule model. Random test points are picked in the specified ranges of each of the rules to ensure that a number of test points are created for each rule.

3. **test_points_*mode*_*seed*_*rule_size*.txt :** Contains random test points for each of the parsed rules in the parsed rule list file (par_rule_list). See the Appendix A for a file example.

4. **linear_search.tcl :** Performs a linear search of the parsed rules list file for each test point and returns a list of matching rule. The list of rules is then converted into CBV format. The CBVs produced by the linear search are used as a set of "golden" test results for the final verification step.

5. **linear_search_results_*mode*_*seed*_*rule_size*.txt :** Contains results from linear search operations converted into CBV format.

6.3.4 Step 4 : Search Operations

The next phase of the verification flow involves using the test points generated to verify the operation of the hardware and software and to determine performance. A flow diagram for step four is shown in Figure 6-5.

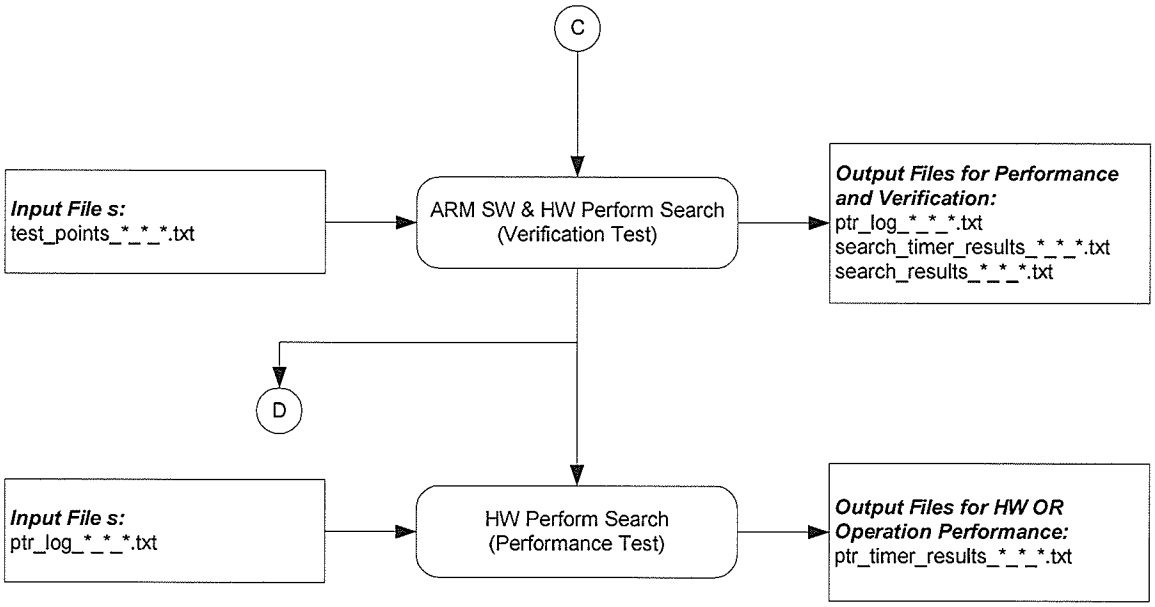

Figure 6-5 : ARM Files and Process

File Descriptions

1. **ptr_log_*mode*_*seed*_*rule_size*.txt :** This is a log of the CBV pointers retrieved from the tree during a test point search. If no pointer is returned from a level in the B-tree then a null pointer 0xFFFFFFFF is stored.

2. **search_timer_results_*mode*_*seed*_*rule_size*.txt:** The purpose of this file is to log statistics during a search operation. See Appendix A for an example of this log file. The three values logged in this file are shown below:

    a. 0x00000CAB : Total time for a search operation from the time the software retrieved the test point. Included the search through the levels of the B-tree until the hardware returns the search result.

    b. 0x0000000C : Count of how many B-tree node keys (elementary intervals) were accessed during as search.

    c. 0x0000000A: Count of how many B-tree nodes were accessed during a search.

3. **search_results_*mode*_*seed*_*rule_size*.txt :** The search_results file contains the resultant CBVs when a 2-dimensional search is performed. The first data word includes the time to perform an OR of the CBVs in the hardware. See Appendix A for an example of this file.

4. **ptr_timer_results_*mode*_*seed*_*rule_size*.txt :** The purpose of the file is to store the results from a performance test in which pre-calculated CBV pointers are read from SRAM and passed to hardware to be retrieved and ORed. Logged values include time for hardware to complete a filter operation and return a value to the response FIFO. This test is performed such that the command FIFO is always full to determine a maximum throughput rate. See Appendix A for an example of this file.

6.3.5 Step 5: Simulation

To assist in debugging and verification a simulation environment was developed to exactly model the entire hardware setup contained on the RLDRAM and LT-XC2V6000+ development boards. The environment was setup such that when problems were found with the actual hardware the log files from tests could be used as inputs to the simulation environment to accurately determine the problem. By doing this hardware can be tested at operational speed to identify problems which can then be diagnosed in simulation. This methodology sometimes referred to as emulation, works exceptionally well for speeding up the verification process. The

simulation and hardware environments both produce CBV list, CBV pointer and OR results files which are compared to ensure correct translation from design files to hardware implementation. A flow diagram for step five is shown in Figure 6-6.

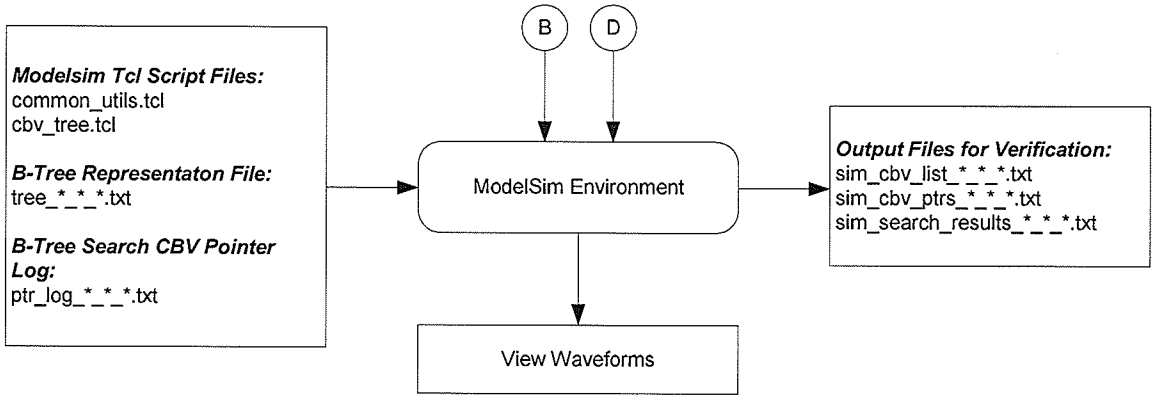

Figure 6-6 : Simulation Files and Process

File Descriptions

1. **cbv_tree.tcl :** Contains procedures for building the CBVs, logging CBV pointers, logging CBV lists, and performing OR operations from lists of pointers.
2. **sim_cbv_list_*mode*_*seed*_*rule_size.txt :** Contains a list of the CBVs for each elementary interval in the B-tree.
3. **sim_cbv_ptrs_*mode*_*seed*_*rule_size*.txt:** Contains a list of the RLDRAM pointers for each CBV.
4. **sim_search_results_*mode*_*seed*_*rule_size*.txt :** The sim_search_results file stores resultant CBVs when a 2-dimensional search is performed in simulation. The first data word includes the time to perform an OR of the CBVs.

6.3.6 Step 6 : Final Verification

During the final verification step, the CBV search file results from simulation, hardware testing, and a simple linear search are compared against each other to ensure there are no differences in results. This step ensures all of the operations in both simulation and hardware testing match up with the "golden" standard test results produced by a basic linear search. A flow diagram for step six is shown in Figure 6-7.

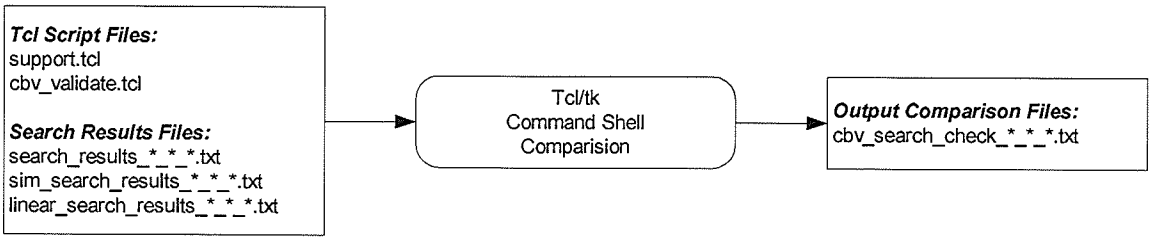

Figure 6-7 : Final Verification Files and Process

File Descriptions

1. **cbv_validate.tcl:** Compares results from the simulation search, linear search and actual test results.

2. **cbv_search_check_*mode*_*seed*_*rule_size*.txt:** Logs results of the CBV validate operation, flagging any differences found in the results.

7 Results

7.1 Perimeter Rule Model

When benchmarking a packet classifier it is desirable to use rule-sets similar to real-life firewall rule-sets. To that end several papers were investigated for possible rule distributions [1][2][8]. Of the papers reviewed the distributions found in the GEM paper [8] are most closely followed because of the specific presentation of rule-set distributions. It should be noted however the GEM model is not always followed as liberties are taken where ambiguities exist. The differences between the GEM model and the developed model are outlined later in this section. The first item of note obtained from the GEM paper is that a large degree of structure is found in most rule-sets. In particular, rule-sets contain a large percentage of rules pertaining to TCP traffic. As a result, benchmarking is performed using TCP fields including two 32-bit IP fields and two 16-bit port fields. These fields are used to mimic inbound and outbound traffic for a network the GEM paper describes as the perimeter firewall.

> "The perimeter firewall assumes a network with two sides: a protected network on the inside, and the Internet on the outside. The inside network consists of 10 class B networks, and the Internet consists of all other IP addresses. Thus, the internal network contains 10*65536 possible IP addresses" [8].

It is also highlighted that organizations large enough to allocate 10 class B networks are quite rare but still should be used for the following reasons:

1. 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 [8].
2. Having a large internal subnet stresses the GEM algorithm since random ranges are selected from the internal ranges. The larger the internal net is, the closer the model is to the Random Model described in section 7.4. [8].

Based on the distributions found in GEM, rule-sets are split between inbound and outbound. Figure 7-1 shows a basic diagram of the network topology.

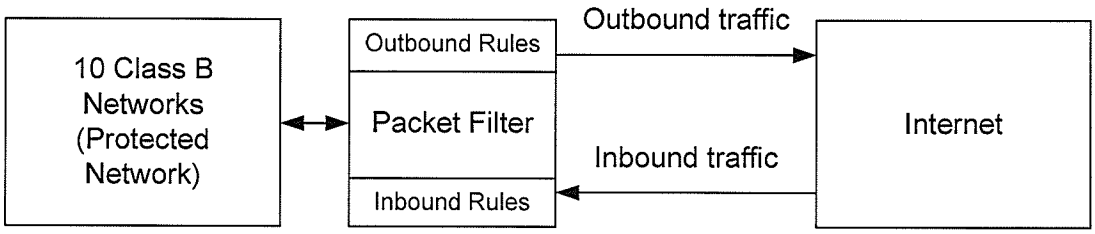

Figure 7-1 : Perimeter Rule Model Network Topology

7.2 Statistical Distributions

Table 7-1 and Table 7-2 provide a summary of the probability distributions found in the GEM paper [8]. The following sections present a summary of the details regarding the inbound and outbound characteristics of the IP address and port rules.

7.2.1 Inbound IP Rules

- source IP addresses are rarely specified in the rules
    - 95% are specified as wild cards
    - 5% are specified as a range uniformly selected from the available IP space
        - Instead a probability distribution is created with increasing probability from Class B to slightly smaller than Class C. This is done, rather than a complete uniform distribution, because it is felt to be more representative of the actual Internet. Figure 7-3 shows the resultant probability distribution function.
- destination addresses for inbound rules are always internal, belonging to the 10 internal class B subnets
    - 45% of the rules have a randomly chosen individual internal IP address as a destination
        - models server machines
    - 15% have a small random range: a range which completely lies inside one of the internal class C networks
    - 30% of the rules have a complete class C as a destination
    - 10% of the rules have a complete class B as a destination

7.2.2 Outbound IP Rules

- destination IP addresses are rarely specified in the rules
    - 90% are specified as wild cards
    - 10% are either specified as a specific address or a range
        - Figure 7-3 shows the chosen probability distribution function for the range
- source IP addresses for outbound rules are always internal, belonging to the ten internal class B subnets
    - 45% of the rules have a randomly chosen individual internal IP address as a destination
        - models server machines
    - 15% have a small random range: a range which completely lies inside one of the internal class C networks
    - 25% of the rules have a complete class C as a destination
    - 10% allow access to a full class B
    - 5% are specified as wild card

7.2.3 Inbound and Outbound Ports

The same statistics are used regardless of whether the direction is inbound or outbound

- source port is rarely specified
    - 98% of the time it is a wildcard, consistent with stateful firewalls which do not need to monitor return traffic
    - 1% specified as a range
    - 1% specified as a single port value
- destination port is precisely specified
    - 96% are specified as single port value from a predefined list
    - 2% are specified as ranges
    - 2% are specified as random single ports

Table 7-1: Statistical distribution for IP address and ports in the perimeter model rule-set. [8]

		Inbound	Outbound
Source address	*	95%	5%
	range	5%	15%
	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 dst	96%	96%
	port is random range	2%	2%
	dst port is single port	2%	2%
Source port	*	98%	98%
	src port is a random range	1%	1%
	src port is from a use port list	0.5%	0.5%
	src port is random 0-65535	0.5%	0.5%

Table 7-2: Statistical Distribution for Ports [8]

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

Destination Port Distribution		
*	0%	
Ranges	4%	
average range size	27030	
single ports	96%	
average number of single ports per rule base	50	
most used ports	80	6.89%
	21	5.65%
	23	4.87%
	443	3.9%
	8080	2.25%

7.2.4 Inbound Rules Detailed Description

Inbound Destination IP

As mentioned in section 6.3.1 a rule-set generator is used to generate rule-sets based on a given distribution. The distributions input into the generator are largely modeled after the information provided in the GEM paper. Table 7-3 shows the three parameters used to determine an inbound destination IP rule. As in a typical probability distribution function, when the probabilities of all of the elements are summed, they equal 1.

Table 7-3: Inbound Rule Destination IP Type Probability

Distribution Type	Probability
0 Random IP Address	0.45
1 Range	0.55
2 Wild Card	0.00

1. **Random IP Address:** To specify the random IP probability distribution function more easily the 32 bits of the address space are broken up into bytes. Each byte can have one of 256 possible values each of which is specified a probability of occurrence. By repeating this procedure for each byte the probability distribution function of the random IP address is fully specified. Using a common notation for an IP Address the bytes are specified as follows: (Byte4).(Byte3).(Byte2).(Byte1).

    - Byte 1: uniform distribution (each value has probability of 1/256)
    - Byte 2: uniform distribution (each value has probability of 1/256)
    - Byte 3: uniform distribution across 10 Class B subnets : [128:137]. Consecutive subnets with same first octet chosen to simplify creation of the generation tool.
    - Byte 4: For a class B network the range of the first octet is between 128 and 191 (179 was selected).

    It should be noted the distribution specified is not only used for single random IP address rules but also to find suitable IP address for prefix ranges. Rules for specifying ranges are also required to be based off valid IP addresses found in the protected network.

2. **Range:** For the purposes of defining ranges with respect to IP addresses, ranges are specified using the prefix method. Range selection begins by first selecting an IP address based on the IP distribution specified and then by selecting a prefix range length based on the range distribution. A value of 32 for the range is used to specify a single IP and a value of zero represents a wildcard. Figure 7-2 shows the probability distribution function chosen for the inbound destination IP address rules. A value of 24 indicates the lower 8 bits of the IP address are used for the range and are 'do not care'. As such, a range is created because the lower 8 bits are masked off for comparison against an incoming IP address.

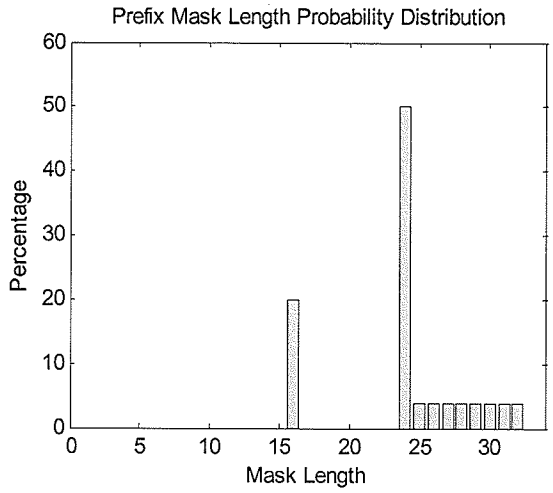

Figure 7-2 : Inbound Destination IP Prefix Mask Length Probability Distribution Function

Notes on Figure 7-2:

- First spike is class B prefix length mask
- Second spike is class C prefix length mask
- Remaining are uniformly spread out to create small networks

3. **Wild Card**: Specifies a condition in which a whole field of the rule is specified as a 'do not care'. In this case zero percent of the rules contain a wild card for the inbound destination IP address.

Inbound Source IP

The tables and distributions in this section describe the probability distribution functions and probabilities used to create the rules for the inbound source IP. Table 7-4 shows the three parameters used to determine the inbound source IP rule.

Table 7-4: Inbound Rule Source IP Type Probability

Distribution Type	Probability
0 Random IP Address	0.00
1 Range	0.05
2 Wild Card	0.95

0. **Random IP Address**: Using a common notation for an IP Address the bytes are specified as follows: (Byte4).(Byte3).(Byte2).(Byte1).

    - Byte 1 ; uniform distribution (each value has probability of 1/256)
    - Byte 2 ; uniform distribution (each value has probability of 1/256)
    - Byte 3: uniform distribution (each value has probability of 1/256)
    - Byte 4 : uniform distribution, has several blocks of IP address removed
        - Specific addresses reserved by the *Internet Assigned Numbers Authority* (IANA) such as 10.0.0.0 - 10.255.255.255 are removed
        - `179.x.x.x` addresses are removed because are they used for the test network

1. **Range**: Figure 7-3 shows the probability distribution function used to define ranges for the inbound source IP rules.

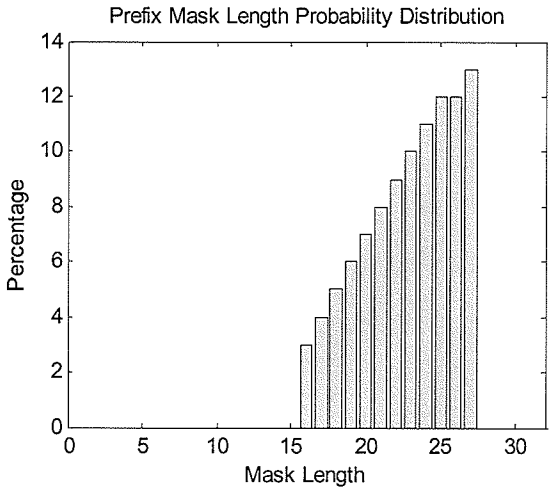

Figure 7-3 : Inbound Source IP Prefix Mask Length Probability Distribution Function

2. **Wild Card**: Specifies a condition in which a portion of the rule is specified as do not care. In this case 95 percent of the rules contain a wild card for the inbound source IP address.

Inbound Source Port

The tables and distributions in this section describe the probability distribution functions and probabilities used to create the rules for the inbound source port.

Table 7-5: Inbound Rule Source Port Type Probability

Distribution Type	Probability
0 Random Port Number	0.5%
1 Used Ports	0.5%
2 Range	1%
3 Wild Card *	98%

Type Descriptions

0. **Single Random Number**: Allows for a small rate of growth in the number of services by adding random generated services, where the port is randomly picked from 0 to 65535 (uniform probability distribution across all ports).

1. **Selected Ports**: Picked from a list of 100 most used ports, ports are selected with the probabilities shown in Table 7-6.

Table 7-6: Most Used Inbound TCP Ports

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

2. **Range**: 1% of source port rules are a range. Common ranges are "all high ports" (1024–65535) and "X11 ports" (6000-6003). As such, most of the weight is placed in extremely low ranges and extremely high. Table 7-7 shows the different probabilities assigned to each range.

Table 7-7: Probability of Range Size for Inbound TCP Port

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

The average range size is calculated by summing up the possible range values for a given range subset, dividing by the number of possible ranges for a given range subset and then multiplying by the given probability. When the values from each of the range subset calculations are summed an average range is obtained, as shown in Equation 7-1. The values shown in Table 7-7 are selected to create an average as close as possible to the one specified in the GEM paper.

Equation 7-1 : Source Port Average Range Size Calculation

$$\left(60000\times 0.45+10000\times 0.05+(3+4+\ldots+30)\times {}^{0.45}\!/_{28}+(100+101+\ldots+1000)\times {}^{0.05}\!/_{901}\right)\approx 27535$$

3. **Wild card:** Specifies a condition in which a portion of the rule is specified as do not care. In this case 98 percent of the rules will contain a wild card for the inbound source port.

Inbound Destination Port

The tables and distributions in this section describe the probability distribution functions and probabilities used to create the rules for the inbound destination port.

Table 7-8: Inbound Rule Destination Port Type Probability

Distribution Type	Probability
0 Random Port Number	2%
1 Used Ports	96%
2 Range	2%
3 Wild Card *	0%

0. **Random Single Number**: Allows a small rate of growth in the number of services by adding of random generated services, where the port is randomly picked from 0 to 65535 (uniform probability distribution across all ports).
1. **Selected Ports**: Picked from a list of 100 most used ports, ports are selected with the probabilities shown in Table 7-9.

Table 7-9: Most Used Outbound TCP Ports

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

2. **Range**: 2 percent of source ports in rules are defined as ranges. Common ranges are "all high ports" (1024–65535) and "X11 ports" (6000-6003). Therefore, most of the weight was placed in extremely low ranges and extremely high.

Table 7-10: Probability of Range Size for Outbound TCP Port

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

Like the source port the goal was to create an average as close as possible to the one specified in the GEM paper. The calculation of the average range size is shown in Equation 7-2

Equation 7-2 : Destination Port Average Range Size Calculation

$$\left(60000\times 0.45+10000\times 0.05+(3+4+\ldots+30)\times {}^{0.45}\!/_{28}+(100+101+\ldots+1000)\times {}^{0.05}\!/_{901}\right)\approx 27535$$

3. **Wild card**: Specifies a condition in which a portion of the rule is specified as do not care. In this case 0 percent of the rules contain a wild card for the inbound source port.

7.2.5 Outbound Rules

With a few slight variations in probability distribution functions the outbound rules are almost the exact opposite of the inbound rules. Other than differences in distribution type probabilities the outbound destination IP matches the inbound source IP and the outbound source IP matches the inbound destination IP. The format for specifying the outbound rules are exactly the same and as with the inbound rules.

Outbound Destination IP

The destination addresses for outbound rules are selected from the Internet with the probabilities shown in Table 7-11.

Table 7-11: Outbound Rule Destination IP Type Probability

Distribution Type	Probability
0 Random IP Address	0.05
1 Range	0.05
2 Wild Card	0.90

Outbound Source IP

The source addresses for outbound rules are selected from the internal addresses with the probabilities shown in Table 7-12.

Table 7-12: Inbound Rule Destination IP Type Probability

Distribution Type	Probability
0 Random IP Address	0.45
1 Range	0.50
2 Wild Card	0.05

Outbound Destination Port

Same as inbound destination port, see section 7.2.4 for details.

Outbound Source Port

Same as inbound source port, see section 7.2.4 for details.

7.3 Performance Analysis

7.3.1 Best Field Order

During development and initial testing it became apparent field order greatly impacted the size of the data structure produced. As such, it became an important objective to compare the performance and data structure size of different field orders. Table 7-13 shows the identifiers for the four fields used for creating the rules.

Table 7-13: Rule Field Identifier

ID	Field	Size
0	Source IP	32-bits
1	Destination IP	32-bits
2	Source Port	16-bits
3	Destination Port	16-bits

For a four dimensional search there is typically twenty four different combinations of field order. However, when combining 2-dimensional search operations which operate in parallel only twelve variations are possible. Only twelve variations exist because the searches are performed in parallel and the order of the pair is inconsequential. The twelve 2-dimensional pairs are outlined in Table 7-14 and the twelve groupings of 2-dimensional pairs are shown in Figure 7-15. The identifiers for these pairs and groupings are used in the result plots in the remainder of this document.

Table 7-14: 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

Table 7-15: 4-Dimensional Field Combinations

File Numbering	4-D Group 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

Results indicate the size of the data structure varies greatly between field orders. As a result it is not possible to obtain test results for all cases because the memory space required for building the structures becomes prohibitive. In an effort to remove the worst field orders an iterative elimination process is used. As a first step small rules sets of size 1024 are generated. The data structures sizes for each of the twelve 2-dimensional field combinations are logged along with the search time results. These results are then used as a basis for elimination of the worst performing field orders. For eliminating a combined analysis is performed in which both 2-dimensional pairs are considered. This is because often one of the 2-dimensional pairs performs exceptionally well while the other performs very poor. In these cases both are removed because the end goal is to find an optimal ordering for a 4-dimensional search.

7.3.2 Plots

The data points for the plots in this section are obtained from generating ten different rule-sets based on the distributions identified in section 7.2. Each rule-set is initialized with a different random seed to obtain a group of rule-sets from which a ninety percent confidence interval can be obtained. The confidence intervals are shown by the line with bars on the top and bottom for each data points in the upcoming plots. If a line is not present it is because the interval is too small to show. Each data point represents the average obtained from the ten runs with the confidence interval indicating ninety percent certainty that the true result lies between the bars. The search time plots also indicate the maximum and minimum values by the lines above and

below the bars. It should be noted the memory usage includes both the CBV and B-tree requirements while the search times represent only the time required for the PFAAE OR operation. Software throughput is evaluated based on memory accesses in section 7.3.4. The result for the 4-dimensional memory usage is created by summing the memory usage results of the two 2-dimensional pairs. By contrast, the result for the 4-dimensional search time is obtained by selecting the worst of the two 2-dimensional results. This is done because in an actual system the 2-dimesional search operations would occur simultaneously.

Outbound Results

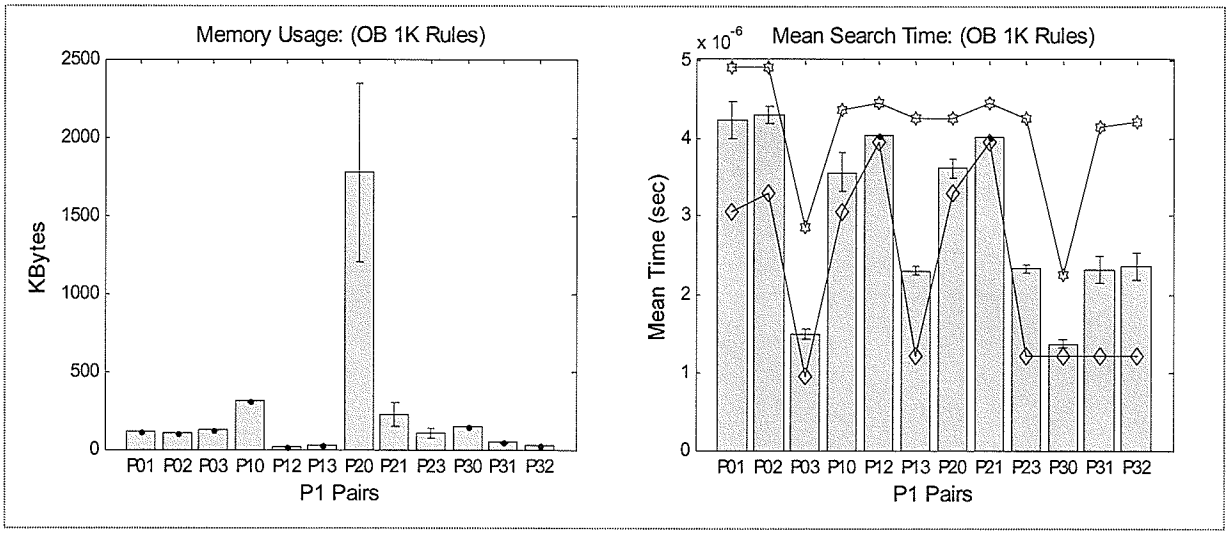

Figure 7-4 : Outbound 1 K Results

Outbound 1 K Results Summary

- Worst by Memory Use: P10, P20, P21
- Worst by Search Time: P01, P02, P12, P21
- Eliminated for 1 K test: None

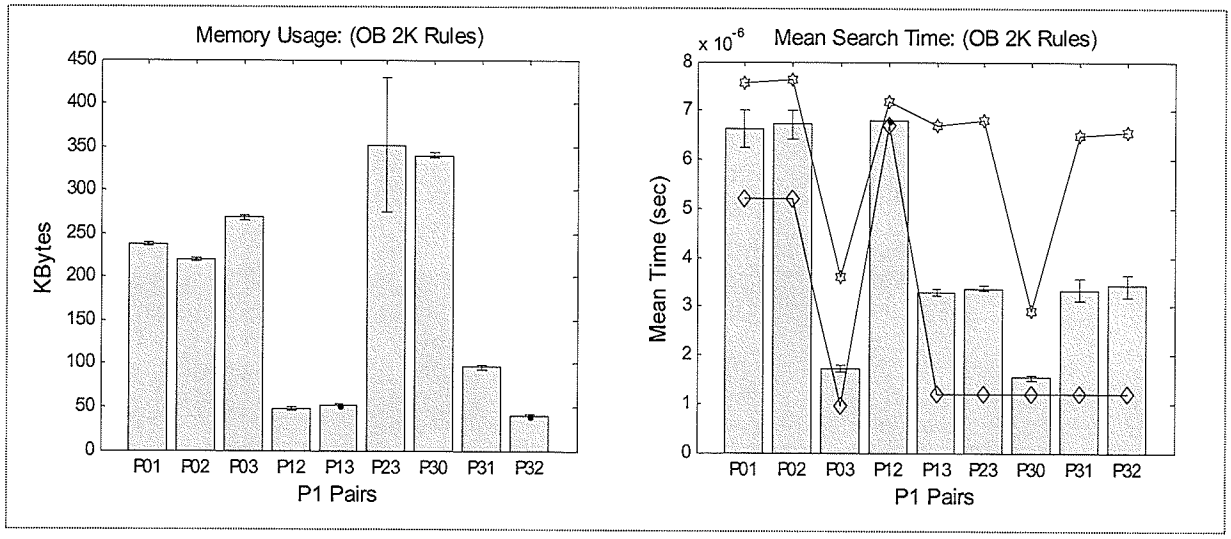

Figure 7-5 : Outbound 2 K Results

Outbound 2 K Results Summary

- Worst by Memory Use: P03, P30, P23
- Worst by Search Time: P01, P02, P12
- Eliminated for 2 K test:
  - P10: previously worst by memory usage
  - P20: previously worst by memory usage
  - P21: previously worst by memory usage

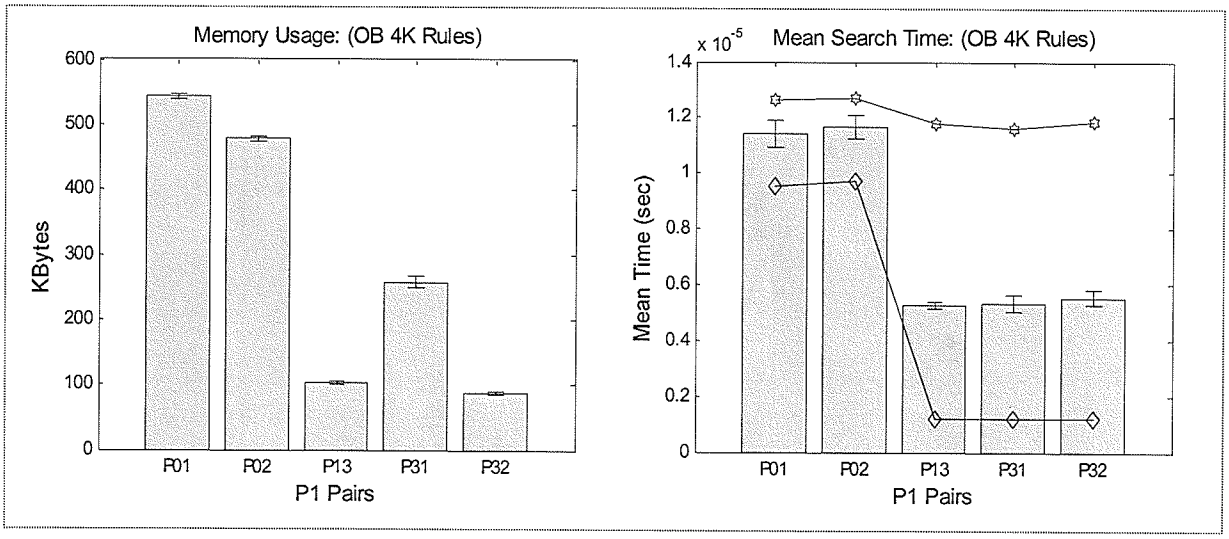

Figure 7-6 : Outbound 4 K Results

Outbound 4 K Results Summary

- Worst by Memory Use: P01, P02

- Worst by Search Time: P01, P02
- Eliminated for 4 K test:
    - P03: previously worst by memory usage
    - P10: previously worst by memory usage
    - P12: removed because it has no 2-dimensional pair left (P03 or P30)
    - P20: previously worst by memory usage
    - P21: previously worst by memory usage
    - P23: previously worst by memory usage
    - P30: previously worst by memory usage

Outbound Results Summary

The final remaining five 2-dimensional pairs (P01, P02, P13, P31, P32) are used in the following sections for analysis of the 4-dimensional groups P01_P32, P02_P13 and P02_P31.

Inbound Results

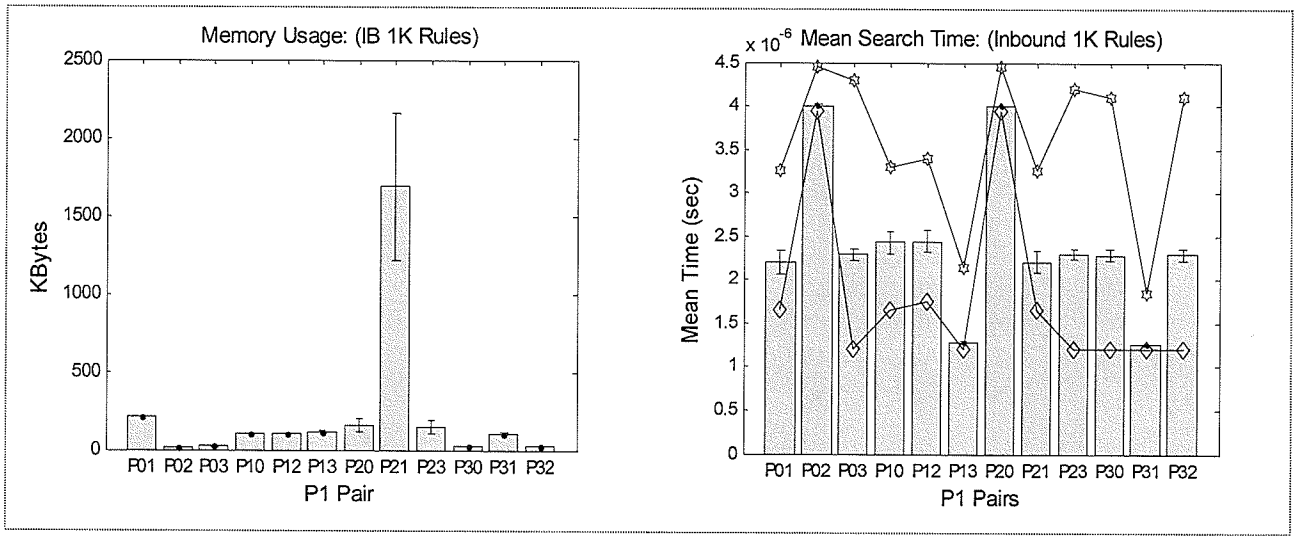

Figure 7-7 : Inbound 1 K Results

Inbound 1 K Results Summary

- Worst by Memory Use: P01, P20, P21
- Worst by Search Time: P02, P20
- Eliminated for 1 K test: None

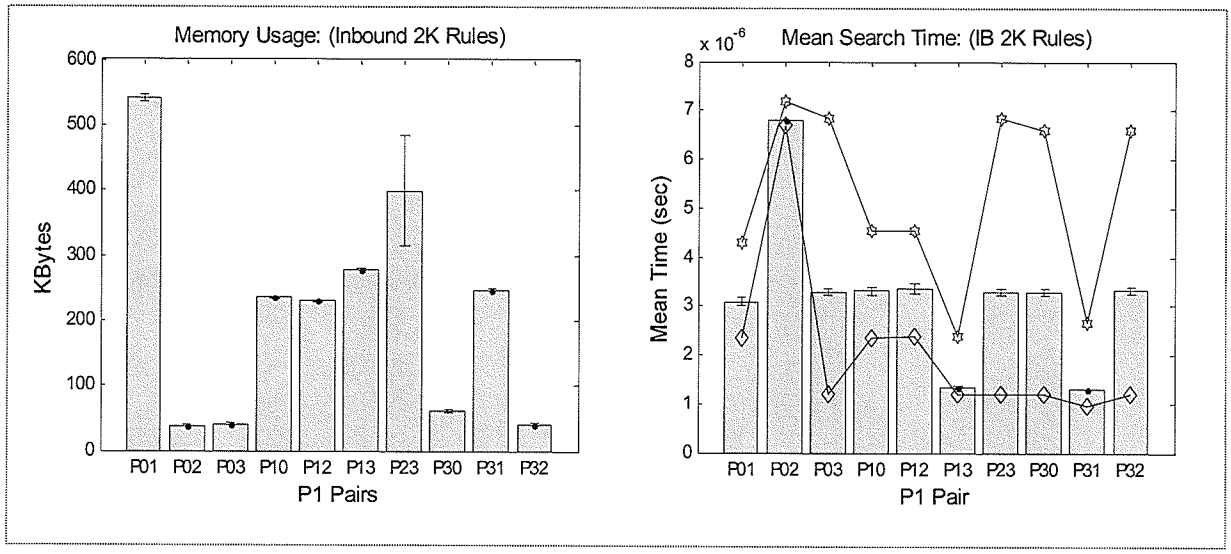

Figure 7-8 : Inbound 2 K Results

Inbound 2 K Results Summary

- Worst by Memory Use: P01, P13, P23
- Worst by Search Time: P02
- Eliminated for 2 K test:
    a. P20: previously worst by search time and memory usage
    b. P21: previously worst by memory usage

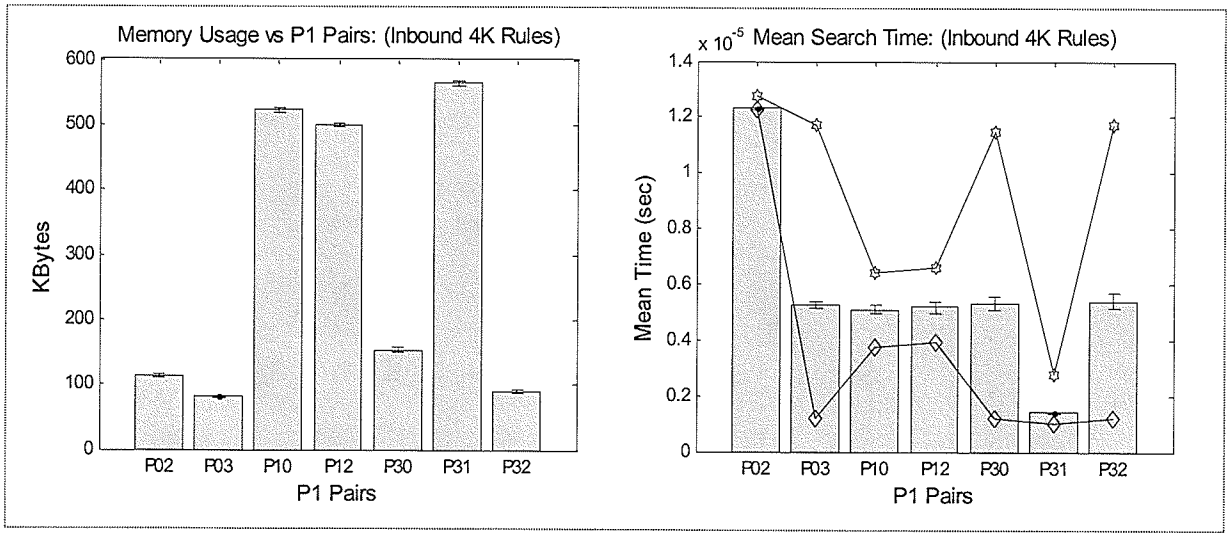

Figure 7-9 : Inbound 4 K Results

Inbound 4 K Results Summary

- Worst by Memory Use: P10, P31

- Worst by Search Time: P02, P32
- Eliminated for 4 K test:
    c. P01: previously worst by memory usage
    d. P13: previously worst by memory usage
    e. P20: previously worst by search time and memory usage
    f. P21: previously worst by memory usage
    g. P23: previously worst by memory usage

Inbound Results Summary

Of the remaining seven 2-dimensional pairs P31 is eliminated due to memory usage. As such, P02 is also eliminated because there are no longer any 2-dimensional pairs left for 4-dimensional analysis. The final remaining five 2-dimensional pairs (P03, P10, P12, P30, P32) are used in the following sections for analysis of the 4-dimensional groups P10_P32, P03_P12 and P30_P12.

7.3.3 Growth Rate

From the results of the inbound and outbound elimination process the remaining 2-dimensional pairs are evaluated up to a rule size of 16 K. The results are used to determine the growth rate of memory usage and search time with respect to rule-set size.

Memory Usage Growth Rate

Based on evaluations of rule-set sizes from 1 K to 16 K the results of the best field order search 2-dimensional pairs and 4-dimensional groups are plotted in Figure 7-10 and in Figure 7-11.

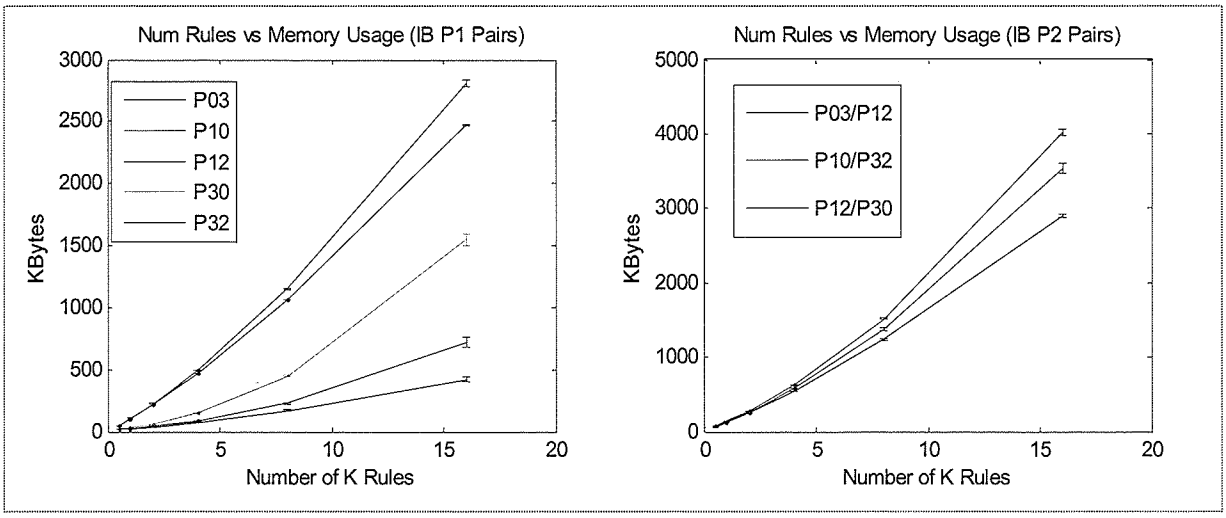

Figure 7-10 : Inbound Memory Growth Rate

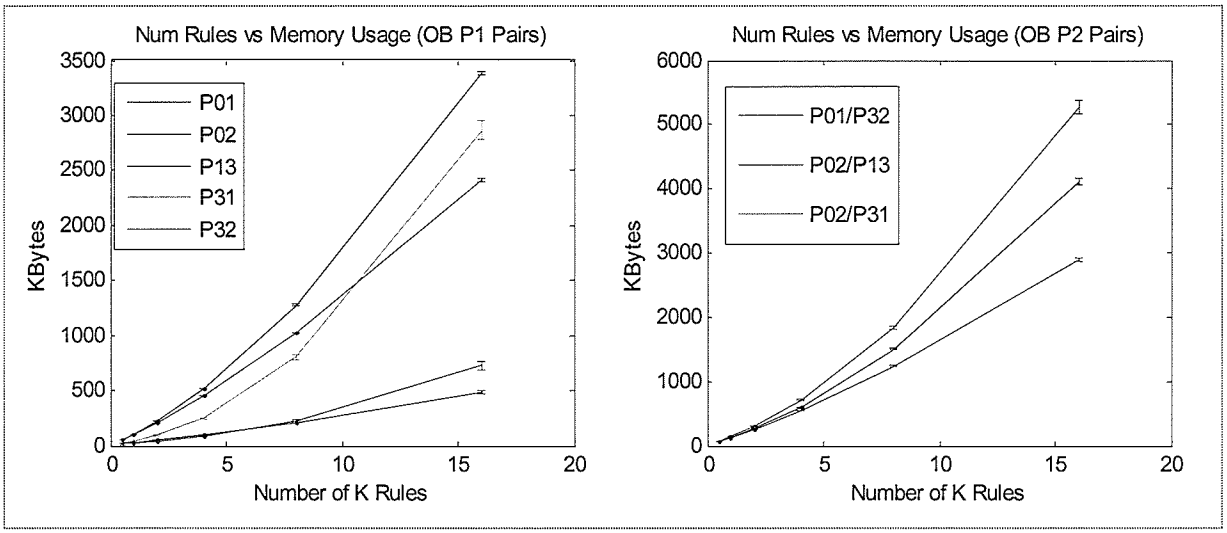

Figure 7-11 : Outbound Memory Growth Rate

From reviewing the resultant memory usage data it is apparent even when rules sizes of 16 K are used the memory required for storage is quite small. No 2-dimensional pair exceeds 3.5 MB and no 4-dimensional group exceeds 6 MB. To determine the actual growth rate of the best performing pair, outbound P02_P13, an analysis is performed using a log-log plot. Figure 7-12 shows the data structure size as a function of rule-set size for the best performing pair. The plot shows the growth on a log-log scale. The green line represents P02_P13 data points, the red line is a curve fit of the worst-case slope of the line, and the blue line represents the plot $y = 10^b x^2$. On a log-log plot equations of the form $y = 10^b x^a$ have a slope equal to a and a y intercept equal

to b. This is clear to see when the log of both sides of $y = 10^b x^a$ is taken to produce $\log_{10} y = a \log_{10} x + b$. Using the Matlab function polyfit the equation of the red line is determined to be $y = 10^{-1.7117} x^{1.2276}$. As such, the data structure size grows almost linearly as a function of rule-set size with a slope of approximately 5/4. This is much lower than the theoretical upper bound of $O(n^3)$. Also evaluated is the outbound pair P02_31, it has a growth rate of approximately 3/2. It is mentioned because it is the pair with the worst growth rate.

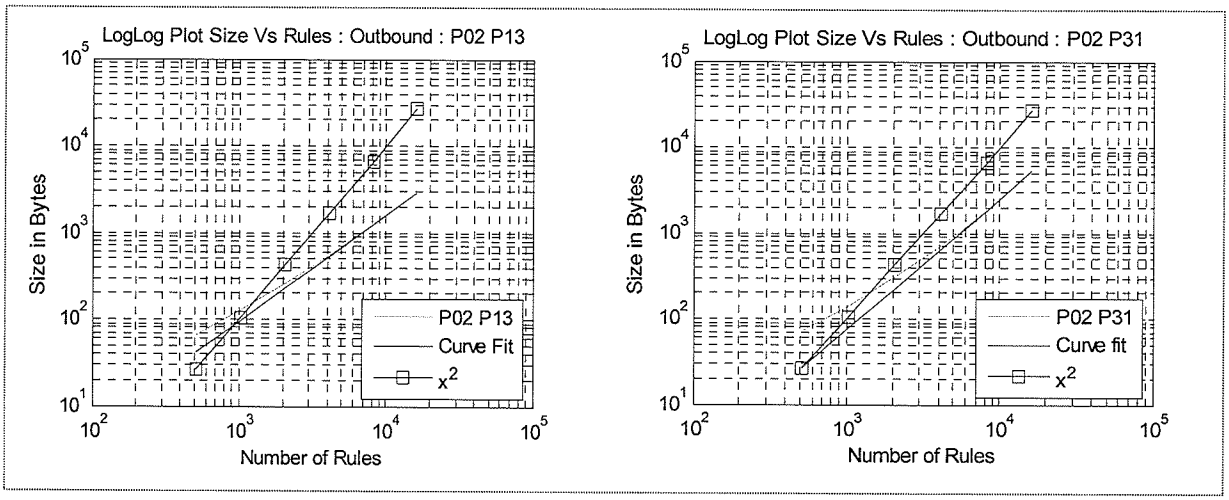

Figure 7-12 : Worst-Case Growth Rate Analysis Plot

Inbound and Outbound CBV Compression Ratio Statistics

To provide insight into the effectiveness of CBV storage as compared to storage without compression Figure 7-13 is provided. The compression ratio is calculated by dividing the number of bits required for a bit-vector without compression by the number of bits required for storage as a CBV. The memory savings are apparent, particularly when the first field is an IP and not a port. Figure 7-14 provides some insight into why the compression ratio difference occurs. As shown by Figure 7-14 the pairs with IP addresses as the first field have much lower number of rules set per CBV as compared to the pairs with a port as the first field. This is because the distributions used for the ports create more clustering as a result of the large percentage of rules assigned to the commonly used port numbers. By contrast, the IP field distributions create more of a spread resulting in CBVs in the second dimension with smaller numbers of rules set. An addition reason for the higher number of rules set in pairs with ports as the first field is because only two levels of B-trees are used. It is obvious the fewer rules are set in a bit-vector the better the compression ratio when creating a CBV.

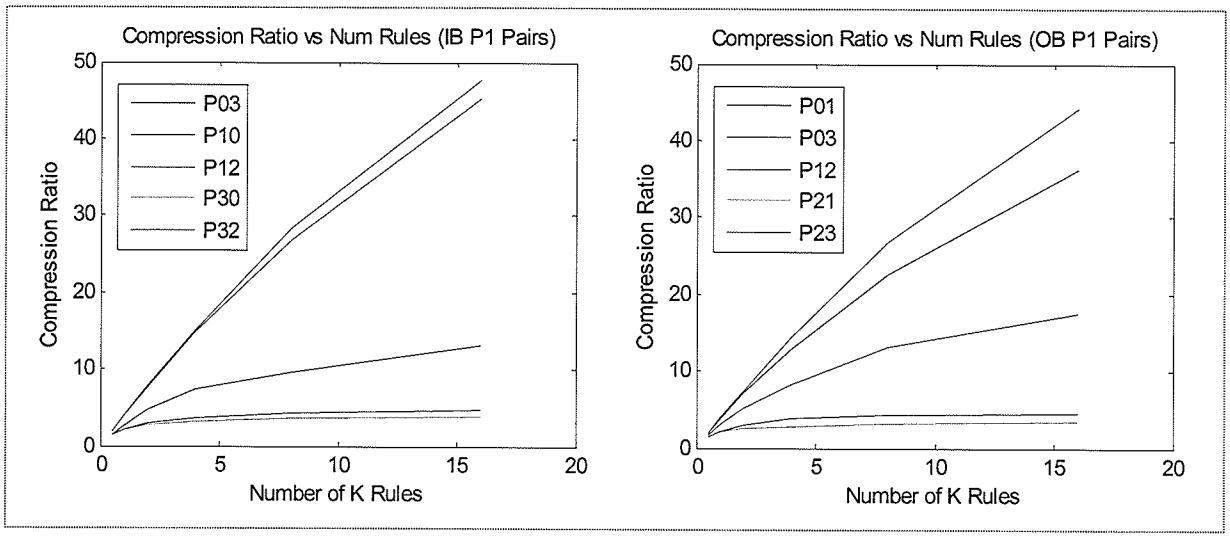

Figure 7-13 : Compression Ratio Statistics

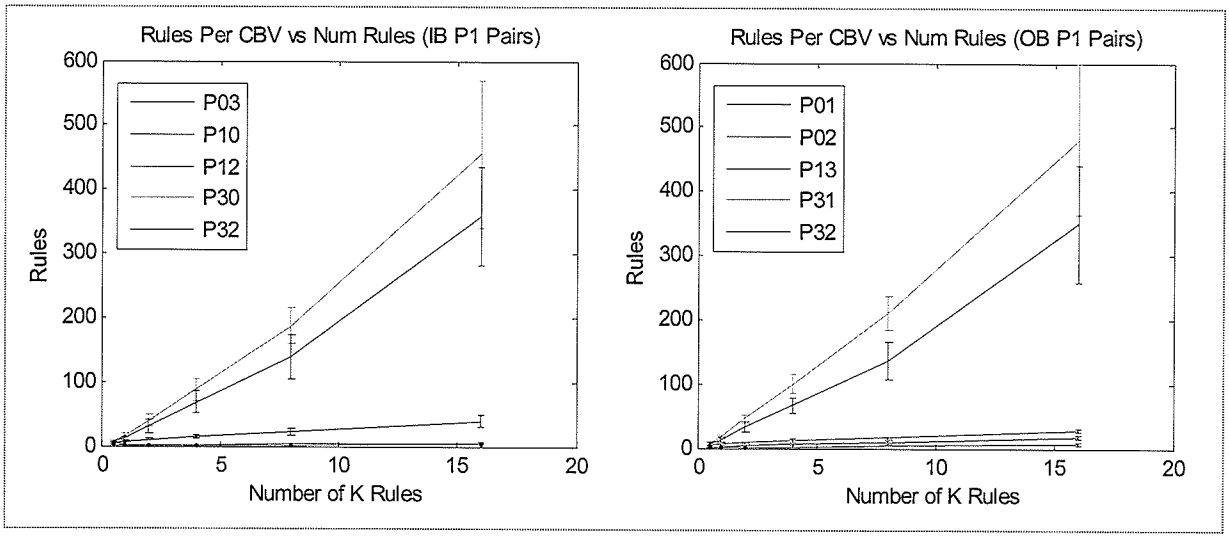

Figure 7-14 : Number of Rules Per CBV

Search Time Growth Rate

Like the memory usage, the search times are also plotted with respect the rule-set size. Figure 7-15 and Figure 7-16 show the inbound and outbound results for the best performing 2-dimensinal pairs and 4-dimensional groups. The result for the 4-dimensional groups is obtained by selecting the worst performing 2-dimensional pair of the available two. This is done because both search operations would be performed in parallel and the worst performing of the two would be the bottleneck for the final AND operation. Once again it should be noted the search times

obtained are only from the PFAAE portion of the search operation. The PFAAE time includes the time to retrieve the CBVs from memory and perform the OR operation. The hardware and software are considered separately because the hardware operations perform orders of magnitude faster than the software. The PFAAE search times are accurately obtained by filling the command FIFO with CBV pointers and then allowing the hardware to run once the FIFO was completely full. In this way the hardware can operate at its maximum speed because it is never waiting for a CBV pointer from software. While this type of operation is not typical, the processor and software are so much slower, this was necessary to test in this manner to obtain an accurate estimate of hardware performance. The results obtained illustrate search time grows linearly with the rule-set size. The reason for the linear growth is attributed to the characteristics of the rules sets used and the fact that the OR operation is $O(n)$. In particular, the probability distribution functions and probabilities for rule types are not a function of rule-set size and as such the characteristics scaled as the rule-set size grew. Research found no mention of the relationship between rule-set characteristics and rule-set size. Unfortunately no evidence was found to prove the theory that the rule-set characteristics likely change with size. Looking at the results the hardware is able to sustain, a throughput of 18 μs/packet, or 56,000 packets per second, for inbound and 24,000 packets per second for outbound traffic. While these results are considerably less than desired one must consider the fact that the algorithm is being run on a development platform. Section 7.5 extrapolates the results obtained for an ASIC to illustrate how the design can be used for Gigabit Ethernet. It should be noted no results are obtained or presented for the final AND operation which intersects the resultant CBVs from each of the 2-dimensional searches. The reason for this is because this operation would be pipelined with the OR operation and is less complex. As such, the bottleneck in the pipelined system is the OR operation.

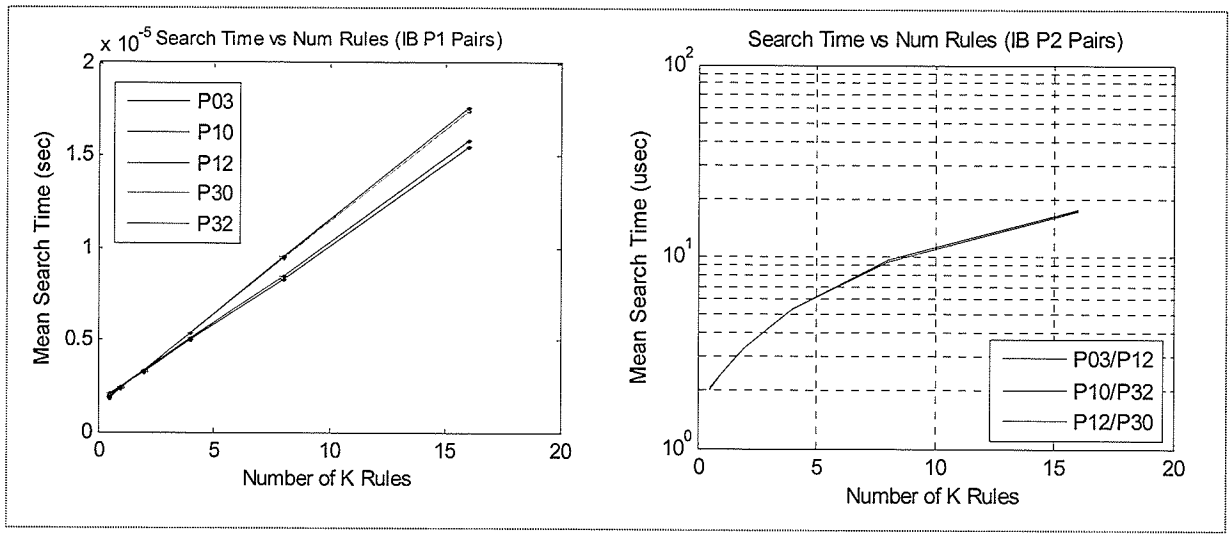

Figure 7-15 : Inbound Search Time Growth Rate

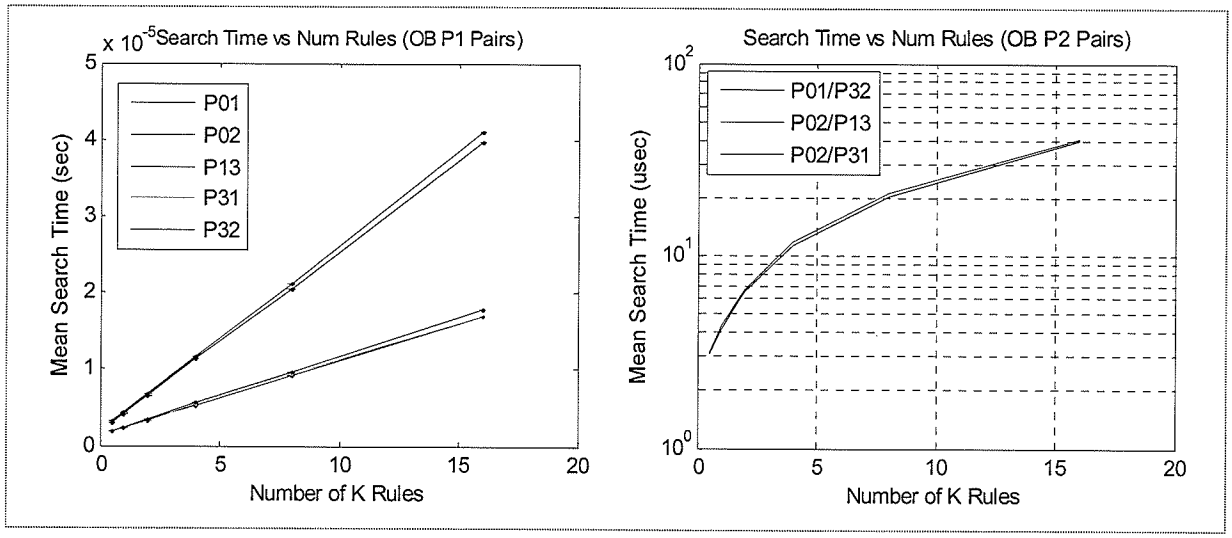

Figure 7-16 : Outbound Search Time Growth Rate

7.3.4 Software B-tree Node Searches

When performing a search the software portion of the system searched through a multi-level B-tree as outlined in the example shown in section 3.3. Due to the limited capabilities of the ARM processor available it was chosen to characterize the software performance based on B-tree node accesses during a search. The number of B-tree nodes accesses during a search also provides a generic metric which allows for what if analysis based on hypothetical systems. Figure 7-17 shows the growth of B-tree nodes access with respect to rule-set size. The figures

indicate average numbers of nodes accessed not minimum or maximum. As expected the growth is logarithmic in nature due to the fact that the height of the B-tree grows as the logarithm of the number of elements inserted. The results obtained from this section are used to determine ASIC performance in section 7.5. The distributions for B-tree nodes accessed during searches for 16 K rule sets are shown in Figure 7-18 and Figure 7-19. These plots are provided as additional information with no analysis provided.

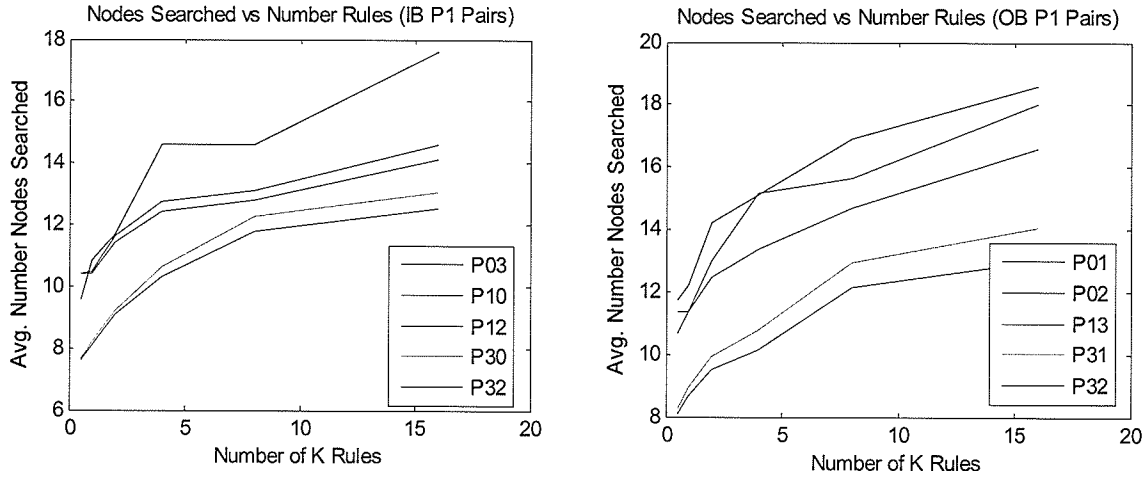

Figure 7-17: Nodes Accesses for Software Search

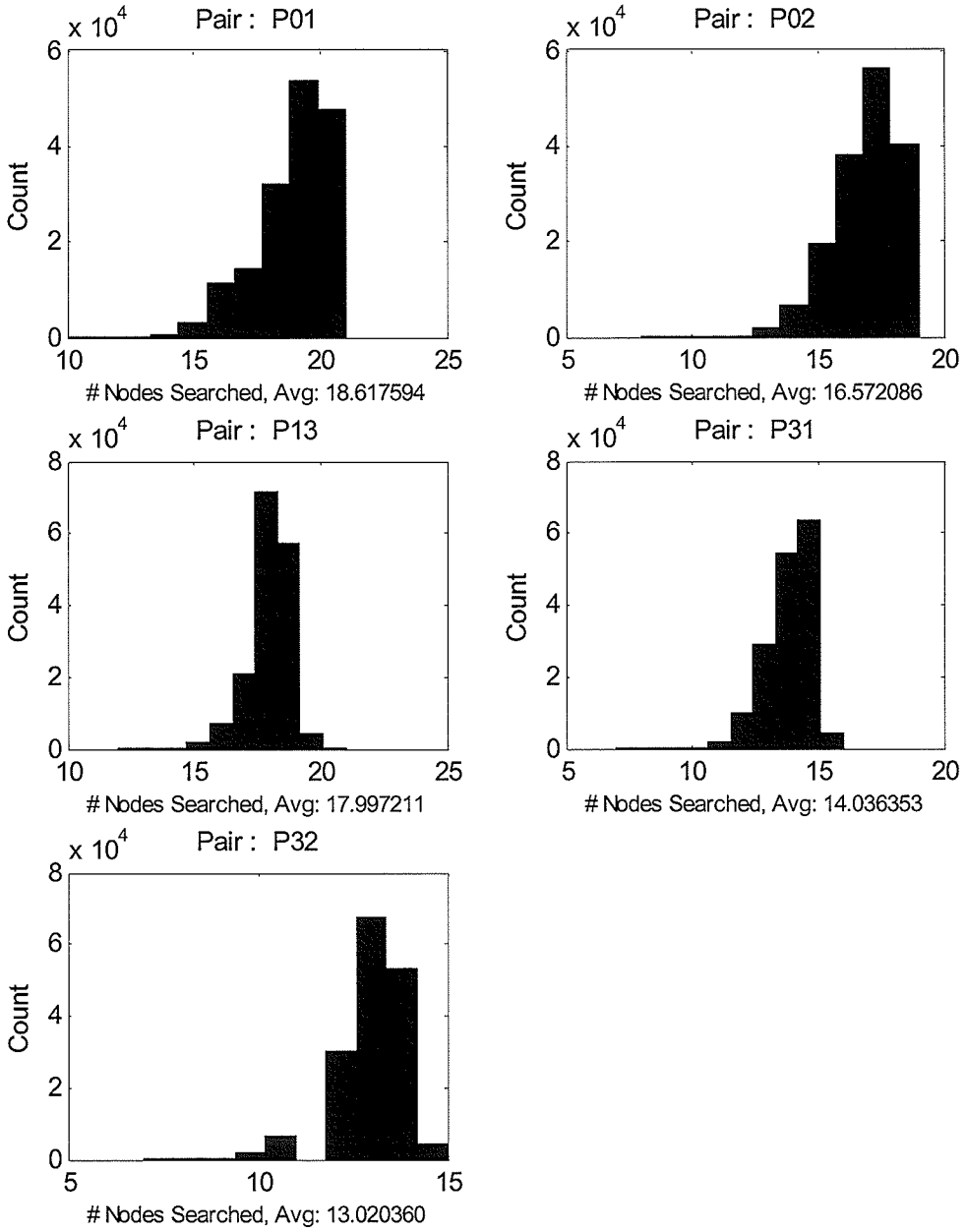

Figure 7-18 : Histograms of Nodes Accesses for Outbound Search Pairs

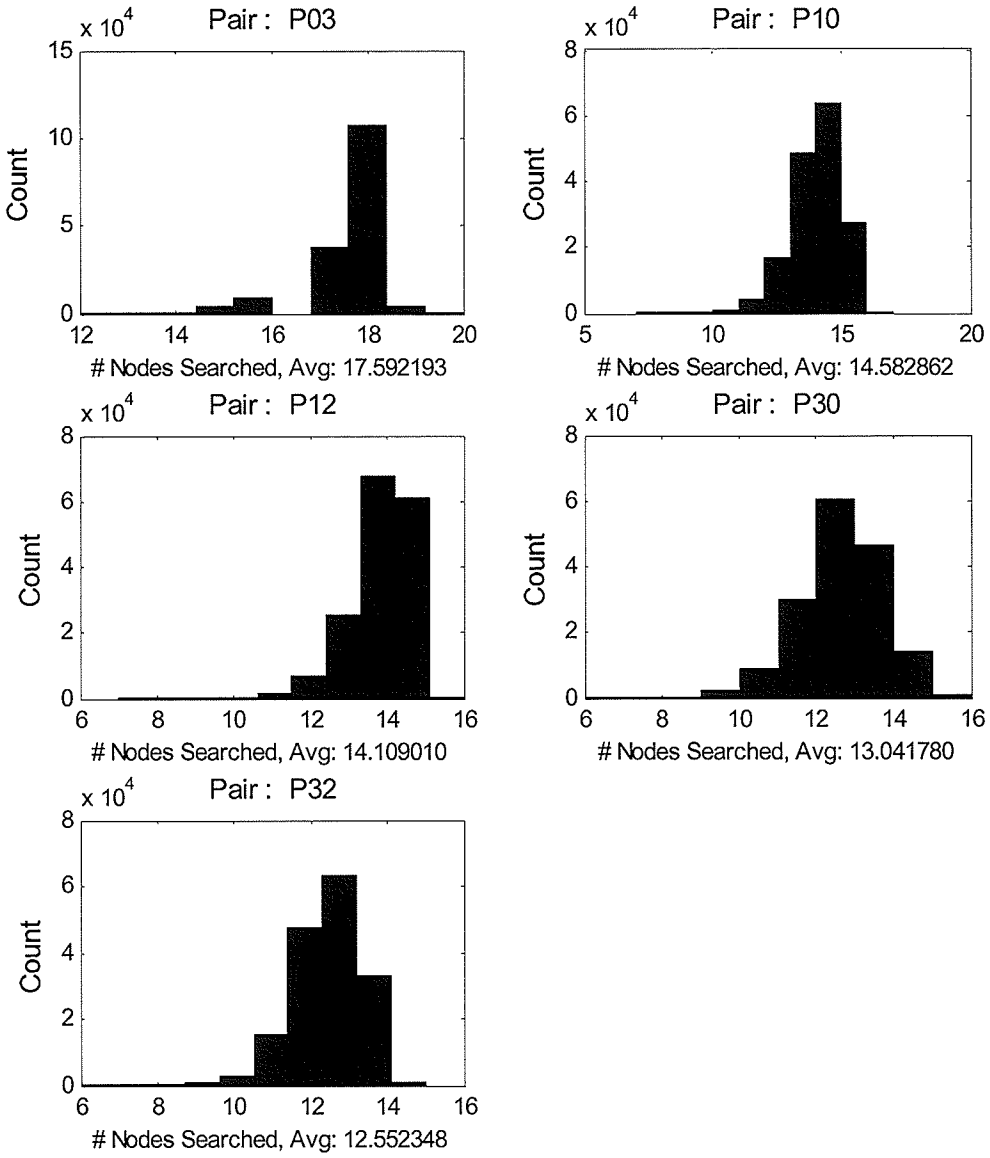

Figure 7-19: Histograms of Nodes Accesses for Inbound Search Pairs

7.3.5 CBV Retrieval Time

When designing a system for optimal throughput it is beneficial to break down the main operations in order to find bottlenecks. To this end the PFAAE filer operation is broken down into the following two operations:

1. Retrieval of the CBV from RLDRAM memory

2. OR operation of CBVs

The primary reason this analysis is done is to verify that the bottleneck of the OR operation is not the RLDRAM memory. Knowing the split of time spent between memory retrieval and actual hardware operations allows for more accurate ASIC performance estimation. Figure 7-20 and Figure 7-21 show the results obtained for CBV retrieval time as a function of rule size. The results show the RLDRAM retrieval time is less than 2 μs for inbound and 3.5 μs for outbound traffic. In either case these results indicate that approximately one tenth of the OR time is spent in retrieval.

Confidence Interval Explanation:

As shown by Figure 7-20 and Figure 7-21 the confidence intervals of the 2-dimensional rule, combinations beginning with a destination port are much larger than the others. The first reason for this is because only two levels of B-trees are used when a port was assigned for use in the first dimension. Only two levels are used because each level is assigned a range width of increasing size. The first level range is assigned values with ranges up to 255 and the second level is assigned values with ranges between 256 and 65535. As the maximum range size for a port is 65535 only two levels are required to contain any possible port rule. Additionally, the probability distribution function for the destination port causes large and small CBVs to be created in the first B-tree level. This is because common single ports apply to many rules resulting in large CBVs and small ranges applying to few rules result in small CBVs. By contrast, the second level of the B-tree only contains small CBVs. These small CBVs result from the small number of rules with ranges large enough to be selected for insertion into the second level. These conclusions are drawn by reviewing the CBV files created. Only the first dimension provides any effect on the final CBVs because the second dimension always contains rules consistent of mostly wild cards. A second dimension containing almost all wild cards provides little or no splitting creating very few elementary intervals. The end result is the following combinations of CBVs:

1. Small CBV from first dimension, Small CBV from second dimension
2. Large CBV from first dimension, Small CBV from second dimension

The combination of these two types of OR operations lead to the larger confidence as compared to the others.

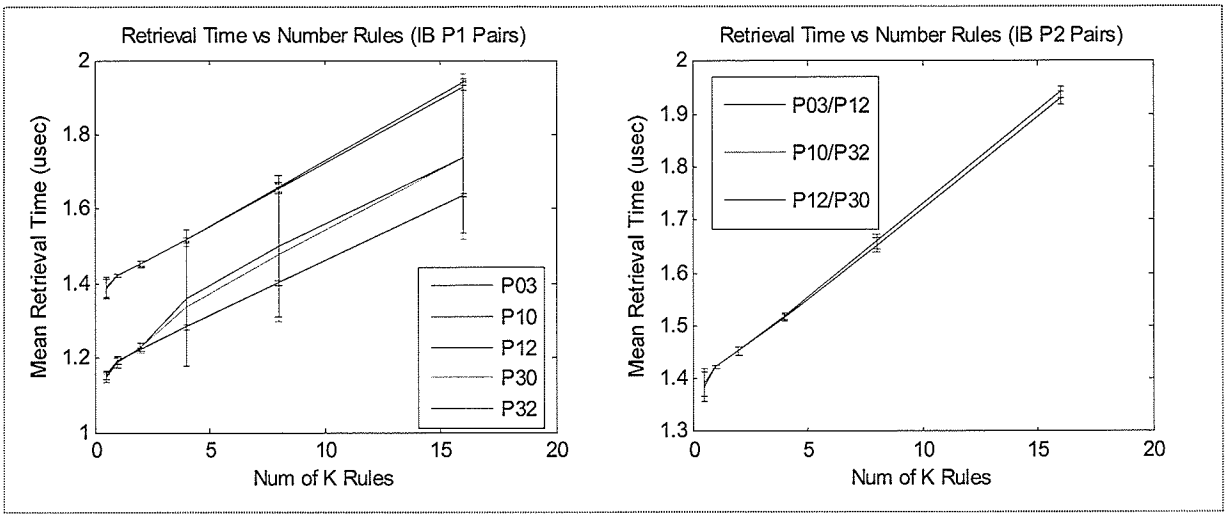

Figure 7-20 : Inbound CBV Retrieval Time

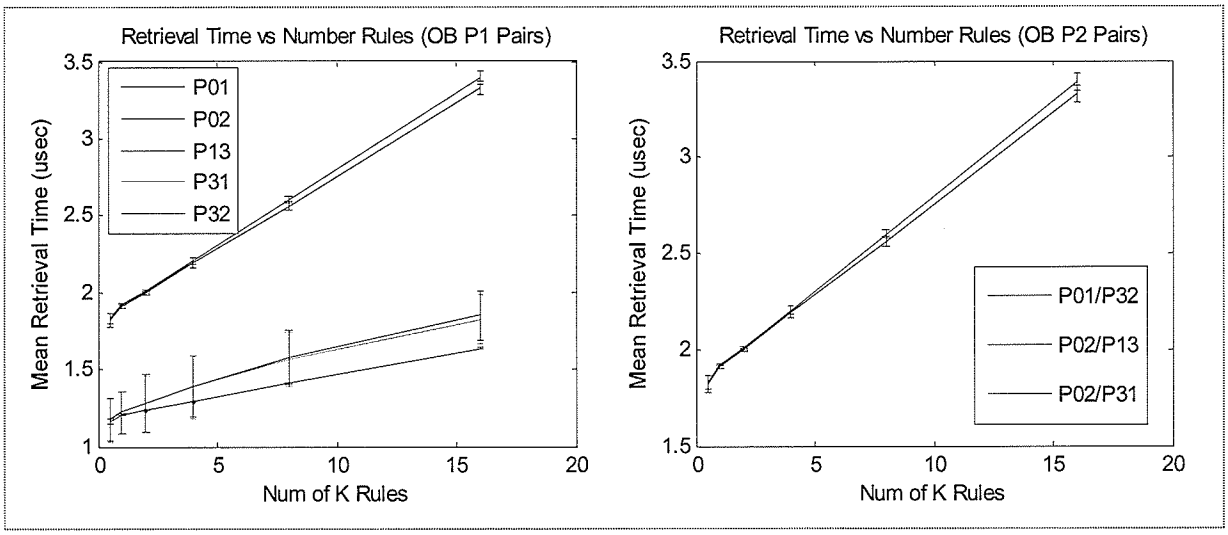

Figure 7-21 : Outbound CBV Retrieval Time

7.4 Random Rule Model

To provide additional insight into the performance of the hardware a rule model was developed based purely on uniformly distributed random rule-sets. This rule model does not consider a particular network structure but rather treats inbound and outbound the same. As inbound and outbound are considered the same the number of possible combinations of pairs is greatly reduced. The reduction is shown in Table 7-16 and Table 7-17 in which the random identifier is shown next to all of the perimeter identifiers it covers. It should be noted these types

of rule-sets are typically not used because they exhibit worst-case memory growth. Actual testing confirmed the expected memory explosion as only one rule-set was able to work at a size of 8 K. A best field order search is not performed as most of the pairs are not able to run because of prohibitive memory requirements. Rather a pair able to run up to the highest possible rule size is chosen to illustrate effects of the random distributions. It is expected the random rule-set will create much larger data structures, resulting in sparse bit-vectors and very fast hardware operations.

Table 7-16: Rule Field Identifier

ID	Field	Size
0	IP Dist	32-bits
1	Port Dist	16-bits

Table 7-17: 2-Dimensional Field Combinations

File Numbering	Random Identifier	Perimeter Identifier	Field 1 (Dimension 1)	Field 2 (Dimension 2)
1	RP00	P01,P10	IP	IP
2	RP01	P02,P03, P12,P13	IP	Port
7	RP10	P20,P21, P30,P31	Port	IP
9	RP11	P23,P32	Port	Port

The probabilities and probability distribution functions for the random IP rules are as follows:

- 50% of the rules contain a random IP selected from the entire IP space
- 50% contain a random IP range based on uniform distribution of prefix lengths

For the random port probabilities are the following:

- 50% of the rules contain a random single number Port 0:65535
- 50% contain a range of ports, the ranges and probabilities are shown in Table 7-18

Table 7-18 : Range Probability Distributions

Range	Probability	
3-30	25%	Allows small ranges of 3-30 in size
100-1000	25%	Allows any range between 100-1000
1000-10000	25%	Allows range of between 1000-10000
10000-60000	25%	Allows range between 10000-60000

7.4.1 Growth Rate & Hardware Search Time

The results for growth rate and hardware search time are obtained for illustrative purposes and are shown in Figure 7-22. The main insights gained from this figure are as follows:

1. Search times are dramatically lower, in fact almost a full order of magnitude. This is likely due to the reduction in wild cards as compared to the perimeter rule model. As there are fewer wildcards, more splitting will occur, creating smaller CBVs, in turn making the hardware operation faster. In the future it would be insightful to perform an analysis of the average number of bits set as a function of rule size.

2. Memory usage is dramatically higher, almost a full order of magnitude. The shape of the plot also appears to be exponentially rising as opposed to being almost linear for the perimeter rule model. This large increase as compared to the perimeter rule model is expected.

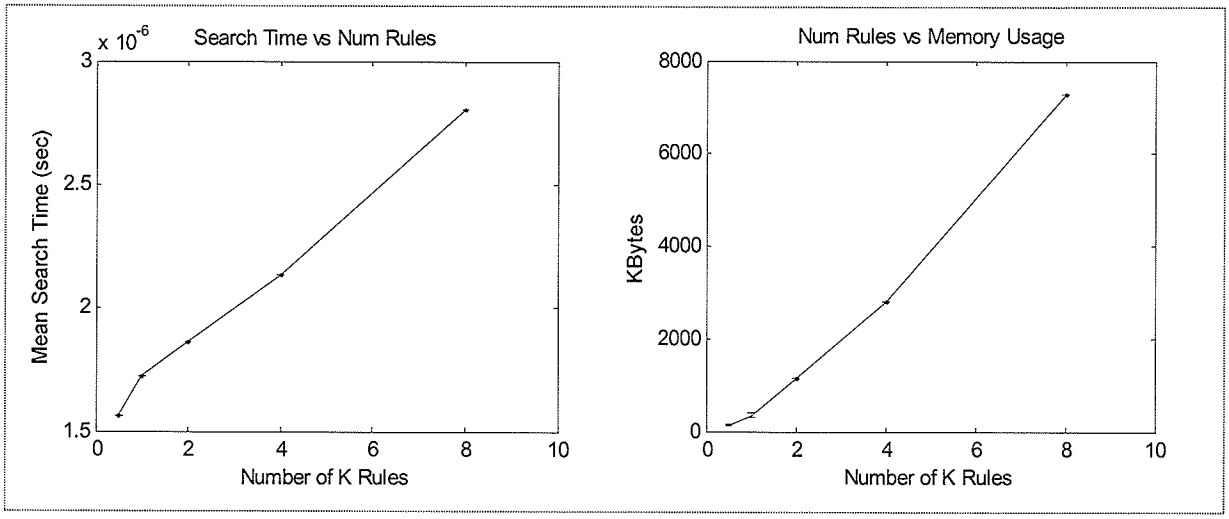

Figure 7-22 : Random Search Time and Memory versus Number of Rules

7.4.2 Software Search Time

Figure 7-23 provides confirmation for the expectation of an increase in the structure size. Clearly the average number of nodes searched has increased dramatically as compared to the results obtained from the perimeter rule model. These results indicate that while the hardware operations are faster the time required for software would be greater.

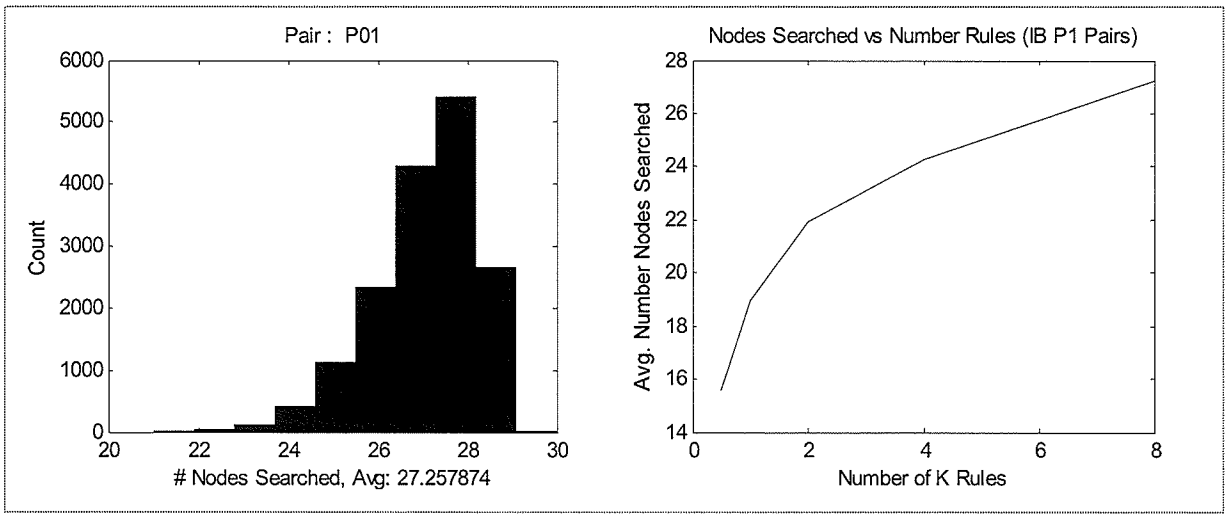

Figure 7-23 : Histogram of Nodes Accessed and Average Number of Nodes Accessed

7.5 Estimated ASIC Performance

The previous studies were an effort meant to validate the algorithm functionality and help produce realistic performance estimates using two different rule-set types. While the development environment was extremely useful for prototyping it had numerous limitations. In particular the bus speeds and processor capabilities turned out to be very restrictive. The addition a more capable processor, DMA functionality and an Ethernet interface would have made the system much faster. Unfortunately, there was simply not enough time to explore these options. To make up for these short comings a performance analysis is presented assuming components selected from common high performance designs. The analysis seeks to prove that with common components the design presented can meet the packet filtering requirements for Gigabit Ethernet. Table 7-19 provides a list of the basic parameters and assumptions made for analysis purposes.

Table 7-19 : ASIC Performance Analysis Parameters

Parameter	Value	Description
Rule-set Size	8 K	4 K for inbound and 4 K for outbound, (n = 4 K)
Bucket number	4	The first dimension contains four buckets for ranges of equal size (b = 4)
Number of Dimensions	4	Two 2-dimensional searches are performed in parallel (d = 2)
B-tree Node Size	96 bytes	Approximate amount of memory required to store ranges in a B-tree with a minimum degree of three (t = 3)
Packet Size	256 bytes	Assumes an average packet size for Gigabit Ethernet

7.5.1 System Throughput Requirement

Assuming a wire speed of 1 Gbps for Ethernet, 12 bytes of interframe gap, 8 bytes of preamble and an average packet size of 256 bytes every stage of the system pipeline must be able to process each packet in approximately 2 μs. Equation 7-3 shows how the value is determined. For the algorithm presented in this thesis there are three stages of the pipeline to consider: software search, PFAAE OR operation, and final intersection operation.

Equation 7-3 : Processing Time per Packet

$$\frac{\text{Wire Speed}}{\text{Average Packet Size}} = \text{Throughput}$$

$$\frac{1\,\text{Gbps}}{276\,\text{bytes} \times 8\,\text{bits/byte}} = 486{,}296\,\text{packets/sec}$$

$$\text{Throughput} = \frac{1}{488281.25\,\text{packets/sec}} = 2\,\text{us processing time per packet}$$

7.5.2 Software Search Performance Requirements

In the following analysis it is assumed a processor exists to process the software nodes fast enough such that the main bottleneck is memory access. The assumption is made that the memory access portion of the software processing is much more time consuming than the effort required to analyze each retrieved B-tree node. This is believed to be a fair assumption as the operations made when processing a B-tree node are quite simple.

Assuming 4 K rules are used for both inbound and outbound rules sets the required memory performance needs to be determined for the software. Given the operations required for packet filtering are run in pipelined fashion 2 μs is available to obtain the memory for a single packet.

Assumptions for Calculations:

1. A four level B-tree is implemented with the rules spread out equally in each of the Trees. In other words each level of the tree contains 1 K rules.
2. A worst-case number of pointers are produced from each search. Every search performed will match to a particular node in each level producing four CBV pointers for the hardware.

Software Complexity:

The search complexity for the software portion of the algorithm is the time taken searching the B-tree structure. Given a t degree B-tree the time taken at each node is $O(t)$ and the total search time is $O(th)$ where h is the height of the tree. Recall from the section 2.4.1 the height of a B-tree can be determined using Equation 2-2. After substitution the formula for the time at a B-tree can be rewritten as $O(t \log_t n)$. Taking into consideration that there is a B-tree in each dimension, where d represents the number of dimensions the equation is rewritten as: $O(d \times t \log_t n)$. Therefore the worst-case depth is really $d \times \log_t n$. Accounting for the multi-

level B-tree structure the tree is now split into b groups of size $\frac{n}{b}$ and to create b 2-dimensional B-trees. This leads to:

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

This indicates the software is required to retrieve a maximum of 16 nodes for each packet. To keep up with the desired line rate the software portion must be able to retrieve 16 nodes of size 96 bytes 0.486 million times a second. The result is memory throughput requirement of 0.75 GBps. A common memory meeting the throughput requirements for memory access is DDR2-800. DDR2-800 with clock speed 400 MHz has a maximum theoretical throughput of 6.4 GBps providing more than enough bandwidth to meet the requirements presented.

Bus Requirement for Transferring CBV Pointers to PFAAE:

Given that b pointers are produced for each packet search the number of pointers required to be moved from processor to the PFAAE is roughly 2 million. Assuming each pointer is 32-bits only a throughput rate of 8 MB per second would be required to transfer the pointers. While little consideration has been made in this thesis with regard to update performance it is clear this would be the driving factor when selecting a suitable bus. Clearly the performance requirements for transferring CBV pointers to the PFAAE are so low that almost any bus would meet the requirements. Given current state of the art it is conceivable the on-chip bus selected would be a full duplex ARM AHB 32-bit bus running at 250 MHz. This bus provides, 8 Gbps of bandwidth, more than the required bandwidth for CBV pointer transfers and potentially enough bandwidth for speedy build and update operations. At this point no analysis is provided for the speed requirements for build and updates operations and is left as a possible area of future research.

7.5.3 Hardware Performance

To meet the throughput requirements the time for CBV retrieval and PFAAE OR operation needs to be less than 2 μs. At this point it is assumed if the CBV retrieval and OR operation are fast enough the final intersection is also fast enough as well. This assumption is made because the intersection operation is typically done as first match and is less complex than the OR operation.

RLDRAM CBV Retrieval Requirement

The development system used for the purposes of this thesis utilized a 32-bit wide DDR RLDRAM interface running at 200 MHz. Current RLDRAM technology, referred to as

RLDRAM II, is capable of running at 400 MHz providing an easy method of doubling the memory bandwidth. Another logical adjustment is to increase the width of accesses from 32 to 128 bits wide. These two improvements combined lead to a factor of eight increase in CBV access performance. Given a retrieval time of 2.25 μs for a 4 K rule-set on the current platform this time needs to be scaled to account for the new memory. As the improvements lead to a factor of eight improvement in performance the retrieval time with the new memory would be 0.28 μs. By subtracting this 0.28 μs from 2 μs, 1.72 μs is left for the PFAAE OR operation.

PFAAE ORing Requirement

The current PFAAE hardware OR time is 10.25 μs for a 4 K rule set size running at 50 MHz. This number includes the CBV retrieval time and when reduced to account for the 2.25 μs of retrieval time produces 8 μs. Based on the assumption that the hardware can be moved to an ASIC with a clock rate of 300 MHz a factor of six improvement could be made. This would result in a PFAAE OR operation in 1.33 μs lower than the required 1.72 μs.

Based on an ASIC implementation including the improvements outlined in this section it is clear the packet classification algorithm outlined in this thesis is capable of operation at a line rate of Gigabit Ethernet assuming an average packet size of 256 bytes.

8 Future Development

Throughout the course of design, verification and implementation for this thesis a number of opportunities for future development became apparent. The opportunities can be categorized into three major types: additional tests to be performed with the system, modifications to the algorithms and modifications to the hardware. These three opportunities are discussed in this chapter.

8.1 Additional Tests

With regard to performing additional testing the major limiting factor in this thesis was time. One of the original goals of the thesis was to test the effect of different bucketing schemes on build time, update time and memory usage. Unfortunately the time was simply not available to allow this testing to be done. The verification phase took longer than expected and cut into this plan.

Additionally, testing could also be performed to check the effect of random test points on the perimeter rule model distributions. While the testing performed did use random points the values were constrained to be selected within the bounds of an existing rule. This was done so performance and verification checks could be done at the same time. The effect of using random tests points constrained by the bounds of existing rules may indicate lower performance as compared to testing with random testing points from the entire field space. This is because selecting test points from existing rules creates a higher probability a search will require an OR operation from more than one CBV. There is a high probability a random test point from the entire field space will only find a match in the fourth level of the B-tree and will not require an OR operation. The fourth level of the B-tree contains rules with large ranges and will likely find a match to any input test point. As such the performance results would likely have a higher average performance.

The effect of the pre-processing step, rule rearrangement, described in the paper on ABV [2] could also be checked. Rearrangement may have a large effect on the CBVs and may provide some interesting results.

8.2 Modifications to Algorithms

The first algorithm modification would be the creation of a hash table to improve the performance of the software search. This involves developing a hash table to find a more

appropriate point in the B-tree to start searching. As such, the search would begin from a node in the B-tree closer to the result than the root reducing the number of nodes to access. The starting search node is located by hashing the input search point with the range limit of the current B-tree level used as a mask. For example, if the search point 0x12345678 was to be searched in a level 1 B-tree the lowest eight bits would be masked off and used as the input into the hash function. Using this method any input starting with the sequence 0x123456XX would hash to the same value. The hash table is then used to return a pointer to the best starting point in the B-tree for searching for a value between 0x12345600 and 0x12345FF. If no pointer is found at a particular hash value then it is immediately known the range is not covered by the B-tree and the search can continue with the next B-tree level.

Secondly, a modification could be made to use different 2-dimensional algorithms instead of the B-tree search algorithm. The effect each algorithm has on the CBVs presents an interesting research topic. Likewise, the use of different algorithms in combination with the scheme developed may produce excellent results. The scheme developed works well when the bit-vectors are sparse created by rule sets with a low percentage of wildcards. This effect was shown by the tests with the random rule-set. There may be other methods the wild cards could be more effectively off loaded to thereby creating sparser bit-vectors.

8.3 Modifications to Hardware

The most obvious modification to hardware would be to implement the software portion on a more capable processor. Additionally, implementation on a platform with a multi-core processor, FPGA, high-speed memory interfaces and Gigabit Ethernet interface would allow for testing of the complete system. Development on a personal computer with an FPGA board for hardware acceleration would be ideal. This type of system should be the target for future development instead of an embedded development platform.

9 Conclusion

In this thesis, the design and verification of a SoC packet classification implementation was discussed. The motivation for this research came from the fact that many schemes have been proposed to solve the multi-dimensional classification problem but none have been shown to scale well beyond two dimensions in terms of speed or rule-set size. Additionally, most schemes disregard update speed in order to increase throughput performance.

To overcome these short comings this thesis introduced three concepts: a compressed bit-vector, 2-dimensional search approach and bucketing. The compressed bit-vector concept was introduced to improve the scalability of a typical bit-vector scheme while still maintaining a hardware amendable implementation. The 2-dimensional search approach was used to reduce the sparseness of the bit-vectors thereby increasing the potential performance of the bit-vector operations. While the effect of the bucketing actually reduced search performance its inclusion in the thesis was done to improve the update and build time. Through the use of pipelining stages for the 2-dimensional search operation, CBV OR operation, and final bit-vector intersection operation performance is further improved.

To test the effectiveness of these concepts a multi-dimensional packet classification scheme was designed and verified using the CMC RPP. The RPP allowed for software and hardware co-design leveraging the benefits of testing at hardware speed to greatly reduce the time required for verification. Having fully testing the hardware and software the next goal was to validate the systems scalability with regard to memory usage and throughput. During initial testing the processor was discovered to be a major performance bottleneck. This required the addition of special hardware to allow the PFAAE to be tested at its maximum rate. As well, it was determined the software would be analyzed with respect to memory accesses rather than actual test performance. To test the scalability of memory usage and throughput synthetic rule-sets were developed based on real firewall database statistics.

Testing first began to determine the best field orders for memory usage and throughput. Once the best fields orders were determined tests were run with rule-sets up to 16 K in size. Testing results showed that the compressed bit-vector concept exhibits a large memory saving as compared to a bit-vectoring scheme without compression. Overall the total data structure was shown to grow at a rate of less than 5/4 as a function of the rule-set size. Compared to the theoretical worst-case growth rate of $O(n^3)$ this is quite good and illustrates the scheme presented is a scalable solution. Testing results showed the PFAAE was able to sustain a throughput of 18

µs/packet, or 56,000 packets per second, for inbound and 24,000 packets per second for outbound traffic. While these results were considerably less than desired, it must be considered that the algorithm was run on a development platform. These results were extrapolated for an ASIC to illustrate how the design can be used for Gigabit Ethernet.

Overall this thesis achieved its primary goal to find a scalable solution to the multi-dimensional packet classification problem. It is believed a reasonable balance was achieved between hardware optimization and programmable flexibility. It should however be noted that additional testing is required to verify the effectiveness of the bucket concept with regard to update and build time performance. Valuable insight was gained on packet classification, embedded system design and verification which can hopefully be built upon in the future.

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] 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.

[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. 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.

[6] P. Gupta, N. McKeown, "Classifying Packets Using Hierarchical Intelligent Cuttings", IEEE Micro vol. 20, no. 1, pp. 34-41, Jan-Feb 2000.

[7] 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.

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

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

[10] 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).

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

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

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

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

[15] ARM Limited,. ARM7TDMI-S Technical Reference Manual (Rev. 4), 2001.

[16] ARM Limted, http://www.arm.com/products/CPUs/ARM7TDMI.html

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

[18] 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.

[19] ARM Limited, *Firmware Suite (Rev 1.4)*, 2002.

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} {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 0x00000001 0x00000800 0x00000000 0x00200000 0x00000000
0x00000000 0x00000001 0x00000001 0x00000001 0x00000800 0x00000000 0x00001000 0x00000000
0x00000000 0x00000001 0x00000001 0x00000002 0x00008201 0x00000000 0x00004000 0x00000400
0x00000040 0x00000000
0x00000000 0x00000001 0x00000001 0x00000001 0x00000408 0x00000000 0x00040000 0x00020000
0x00000000 0x00000001 0x00000001 0x00000005 0x000016ED 0x00000000 0x10000000 0x00200000
0x00200000 0x00010000 0x00000040 0x00440400 0x04000000 0x00010000 0x00000040 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 rule-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
1
:
:
:

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 2-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
0x0000001F 0x00000000 0xFFFFFFFF 0xB3806900 0xB38069FF
     :          :          :          :          :
     :          :          :          :          :
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 {first DW} includes the time to perform
#     an OR of the CBVs in the PFAAE.
#
#--------------------------------------------------------------------------
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
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
0x00002A0F 0x00000000 0x00000001 0x00000001 0x00000004 0x0000692D 0xBB89F7E7 0x00000002
0x20000000 0x08000040 0x00800000 0x80000000 0x00000100 0x02000000 0x80000100
0x00002A0F 0x00000000 0x00000001 0x00000001 0x00000004 0x0000692D 0xBB89F7E7 0x00000002
0x20000000 0x08000040 0x00800000 0x80000000 0x00000100 0x02000000 0x80000100
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
0x0000000B
    :
    :
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 for the hardware to retrieve
#    and OR the CBVs for search operation.  The time is in 20MHz clock cycles.
#
#-------------------------------------------------------------------------
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
0x00000020
0x00000020
0x00000020
    :
    :
    :
0x00000023
0x00000020
0x0000001F
#-------------------------------------------------------------------------
#    Completed Pointer Timer Results LogFile
#
#-------------------------------------------------------------------------
```

A.8 Tree File

```
#-------------------------------------------------------------------------
#    Starting Tree LogFile
#
#    Seed: 0
#    Mode: 0
#    Rule Size: 512
#    Direction: Inbound
#
#-------------------------------------------------------------------------
# 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 Test Points File

```
#-------------------------------------------------------------------------
#    Starting Test Points LogFile
#
#    Seed: 0
#    Mode: 0
#    Rule Size: 512
#    Direction: Inbound
#
#-------------------------------------------------------------------------
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
0x6BC4DD86 0xB38813F2
     :          :
     :          :
     :          :
0xBD4BEEAF 0xB386DFC4
0xE6E96DC1 0xB386DFE7
#-------------------------------------------------------------------------
#    Completed Test Points LogFile
#
#-------------------------------------------------------------------------
```