University of Crete
School of Sciences and Engineering
Computer Science Department

Design and Implementation of Network Packet Classification Engines

Master's Thesis

Vassilios Papaefstathiou

March 2005
Heraklion, Greece

Πανεπιστήμιο Κρήτης
Σχολή Θετικών και Τεχνολογικών Επιστημών
Τμήμα Επιστήμης Υπολογιστών

Σχεδίαση και Υλοποίηση Μηχανών Κατηγοριοποίησης Πακέτων Δικτύου

Εργασία που υποβλήθηκε από τον
Βασίλειο Παπαευσταθίου
ως μερική εκπλήρωση των απαιτήσεων
για την απόκτηση
Μεταπτυχιακού Διπλώματος Ειδίκευσης

Συγγραφέας:

Βασίλειος Δ. Παπαευσταθίου
Τμήμα Επιστήμης Υπολογιστών

Ηράκλειο, Μάρτιος 2005

Design and Implementation of Network Packet Classification Engines

by
Vassilios Papaefstathiou

Master's Thesis

Department of Computer Science
University of Crete

Abstract

Switches and routers are the most important building blocks of today's networks and the Internet. The wide spread and growth of the Internet imposes high performance and efficiency in the network infrastructures in order to support the QoS, demanded by the state-of-the-art network applications, and the ever increasing network traffic. This thesis primarily addresses the searching tasks performed by Internet routers and switches in order to forward packets and provide differentiation of services to packets belonging to particular traffic flows. Considering that these searching tasks must be performed in a per packet basis, the speed and effectiveness of the solutions to these problems determines the efficiency of the overall networks.

We have proposed novel hardware based classification schemes to support QoS in multiple network layers and meet today's high speed links' requirements. Initially, we propose a *Hash Based Classification Engine (HBCE)* to address the problem of classification in the network MAC layer (Data Link Layer). Moving to routers we developed an innovative scheme, *Bitmap Oriented Strides (BOS),* which faces the Longest Prefix Matching problem and supports fast lookups by efficiently managing the routing tables. Striving to enhance the granularity of service differentiation we propose a 5-dimentional packet classification scheme that leverages packet fields from higher network layers. We developed the *Bloom Based Packet Classification (B2PC)* scheme which is an innovative approach for decomposed packet classification that involves Bloom filter data structures.

The proposed implementation of the *Hash Based Classification Engine (HBCE)*, can support up to 64K MAC address rules at aggregate speeds of more than

50 Gbps using only 540KB of memory. Moreover, the hardware implementation of *Bitmap Oriented Strides (BOS)* can handle more than 90K prefixes while requires only 600KB of memory and allows routing decisions for more than 240 million packets per second. Finally, a hardware realization of the *Bloom Based Packet Classification (B2PC)* handles more than 4000 rules by involving 530KB of memory and can classify packets at rates greater than 8Gbps.

Keywords: packet classification, routing lookups, longest prefix matching
Thesis Supervisors: Prof. Manolis Katevenis – Dr. Yiannis Papaefstathiou

Σχεδίαση και Υλοποίηση Μηχανών Κατηγοριοποίησης Πακέτων Δικτύου

Βασίλειος Δ. Παπαευσταθίου

Μεταπτυχιακή Εργασία

Τμήμα Επιστήμης Υπολογιστών
Πανεπιστήμιο Κρήτης

Περίληψη

Οι μεταγωγείς και οι δρομολογητές είναι τα πιο σημαντικά δομικά στοιχεία των σημερινών δικτύων και του Διαδικτύου. Η μεγάλη εξάπλωση και ανάπτυξη του Διαδικτύου απαιτεί υψηλές επιδόσεις και ικανότητες από τις δικτυακές υποδομές ώστε να υποστηρίξει την ποιότητα των υπηρεσιών, που απαιτείται από τις δικτυακές εφαρμογές τελευταίας τεχνολογίας, και την συνεχή αύξηση της δικτυακής κίνησης. Η εργασία αυτή ασχολείται κυρίως με τις λειτουργίες αναζήτησης που εκτελούνται από τους δρομολογητές και τους μεταγωγείς του δικτύου με σκοπό να προωθήσουν πακέτα και να παρέχουν διαφοροποιημένες υπηρεσίες στα πακέτα που ανήκουν σε ιδιαίτερες ροές κίνησης. Θεωρώντας ότι αυτές οι λειτουργίες αναζήτησης πρέπει να διεκπεραιωθούν για κάθε πακέτο, η ταχύτητα και η αποτελεσματικότητα των λύσεων σε αυτά τα προβλήματα καθορίζει την απόδοση των δικτύων.

Προτείνουμε καινοτόμα σχήματα κατηγοριοποίησης πακέτων για υλικό τα οποία υποστηρίζουν ποιότητα υπηρεσίων σε πολλαπλά στρώματα δικτύου και ικανοποιούν τις υψηλές ταχύτητες των σημερινών συνδέσμων. Αρχικά, προτείνουμε μια *Μηχανή Κατηγοριοποίησης Βασισμένη σε Διασπορά (ΜΚΒΔ)* για να διεκπεραιώσει το πρόβλημα της κατηγοριοποίησης στο στρώμα δικτύου MAC (Στρώμα Σύνδεσης Δικτύου). Για τους δρομολογητές αναπτύξαμε ένα καινοτόμο σχήμα, *Δρασκελιές Προσανατολισμένες σε Bitmaps (ΔΠΒ),* το οποίο αντιμετωπίζει το πρόβλημα του Ταιριάσματος Μεγίστου Προθέματος και υποστηρίζει γρήγορες αναζητήσεις, διαχειριζόμενο αποδοτικά τους πίνακες δρομολόγησης. Προσπαθώντας να πετύχουμε καλύτερη λεπτομέρεια στις διαφοροποιημένες υπηρεσίες προτείνουμε ένα 5-διάστατο σχήμα κατηγοριοποίησης πακέτων το οποίο χρησιμοποιεί πεδία πακέτων από υψηλότερα στρώματα του δικτύου. Αναπτύξαμε το σχήμα *Κατηγοριοποίηση Πακέτων Βασιζόμενη σε φίλτρα Bloom (ΚΠΒ2)* το οποίο είναι μια καινοτόμος προσσέγιση για αποσυνθετική κατηγοριοποίηση πακέτων η οποία περιλαμβάνει δομές δεδομένων τύπου Bloom φίλτρων.

Η προτεινόμενη υπολοίηση για την *Μηχανή Κατηγοριοποίησης Βασισμένη σε Διασπορά (ΜΚΒΔ)* μπορεί να υποστηρίξει 64 χιλιάδες κανόνες διευθύνσεων MAC σε συνολικές ταχύτητες μεγαλύτερες από 50 Gbps χρησιμοποιώντας μόνο 540KB μνήμης. Επιπλεόν, η υλοποίηση σε υλικό του σχήματος *Δρασκελιές Προσανατολισμένες σε Bitmaps (ΔΠΒ)* μπορεί να διαχειριστεί περισσότερα από 90 χιλιάδες προθέματα χρησιμοποιώντας μόνο 600KB μνήμης και επιτρέπει αποφάσεις δρομολόγησης για περισσότερα από 240 εκατομμύρια πακέτα ανα δευτερόλεπτο. Τέλος, μια υλοποίηση υλικού του σχήματος *Κατηγοριοποίηση Πακέτων Βασιζόμενη σε φίλτρα Bloom (ΚΠΒ2)* διαχειρίζεται περισσότερους από 4000 κανόνες χρησιμοποιώντας 530KB μνήμης και μπορεί να κατηγοριοποιεί πακέτα σε ρυθμούς υψηλότερους από 8 Gbps.

Επόπτες Εργασίας: Καθ. Μανόλης Κατεβαίνης – Δρ. Ιωάννης Παπαευσταθίου

Acknowledgments

This work was financially supported by the Institute of Computer Science (ICS) of the Foundation for Research & Technology - Hellas (FORTH), Heraklion, Crete, Greece. Besides ICS-FORTH I would also like to thank all those people who helped me throughout this work. First of all, I would like to thank my supervisors, Dr. Yiannis Papaefstathiou and Prof. Manolis Katevenis who gave me the opportunity to work on the subject of packet classification. Our close collaboration, especially with Dr. Yiannis Papaefstathiou, introduced me to the scientific way of thinking and offered me a valuable experience on conducting research.

I would also like to thank the rest of the Computer Architecture and VLSI group at ICS-FORTH for their help. Especially I would like to thank Dr. Christos Sotiriou for his help and guidance in the back-end tools and flow; Dimitris Simos for his valuable help in numerous hardware issues and corrections and suggestions for this report; and Olga Dokianaki and Giorgos Kalokairinos for their valuable support. Moreover, I want to thank Spyros Lyberis for putting me “deep into hardware” and Vanessa Evangelatou for her collaboration in many hardware related projects.

Thanks are also due to all my friends: Chariton, Manos, Dimitris, Takis, Panos, Panos (Castillo), Dimitris (Darth), Lilly, Themis, Vassilis, Ilias, Anestis, Stamatis and Michalis. Deep thanks also to Litsa who stood by me and really supported me all this time.

Finally, I would like to thank my brother Meletis, my sister Aggeliki and my parents Dimitris and Vassiliki for their love and support all these years. All I have accomplished I owe it to my parents and therefore I dedicate this work to them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Nowadays, the Internet has emerged as a global communications service of continuously increasing importance. The ever expanding scope of Internet users and applications require the network infrastructures to exchange large volumes of information, augmenting the already challenging performance constraints. This thesis addresses the searching tasks performed by Internet routers in order to forward packets and apply network services to packets belonging to particular traffic flows. Considering that these searching tasks must be performed for each packet traversing the router, the speed and efficiency of the solutions to these problems determines the performance of the router, and hence the entire Internet.

1.1 Internet and Networking

The building blocks of the Internet are essentially interconnected networks, each consisting of heterogeneous hosts, links, and routers. Hosts produce and consume packets, or datagrams, which contain chunks of data - a part of a file, digitized voice samples, etc. Hosts may be personal computers, workstations, servers and network enabled electronic appliances such as Personal Digital Assistants (PDAs) or mobile phones. Packets indicate the sender and receiver of the data similar to a letter in the postal system. Links connect hosts to routers, and routers to routers. Links may be twisted-pair copper wire, fiber optic cable or a variety of wireless radio technologies. The role of routers is to switch packets from incoming links to the appropriate outgoing links depending on the destination of the packets. Packets may traverse many links, called hops, in order to reach its destination. Due to the impermanent nature of network links (failure, congestion, additions, removals), routing protocols allow the routers to continually exchange information about the state of the network so as to decide the forwarding of packets destined for a particular host, network, or sub-network.

Ethernet Networks

Ethernet is the dominant networking protocol used in Local Area Networks (LANs) over the last decades. It is the most widely adopted protocol in the physical and data link layer of the network. It defines 48-bit addresses, called MAC addresses, that are unique for each network interface, and uses them in order to manage the circulation of packets in the physical medium. Ethernet's speeds started from 10 Mbps and eventually evolved to 100Mbps, 1Gbps and recently to 10Gbps.

IP and TCP Protocols

The original Internet protocol comprises mainly of two protocols: the Internet Protocol (IP) and the Transmission Control Protocol (TCP). The primary function of the Internet Protocol (IP) is to provide an end-to-end packet delivery service. This task is accomplished by including information regarding the sender and receiver inside each packet transmitted through the network. IP protocol specifies the format of this information which is prepended to the content of each packet, namely the packet header. In order to uniquely identify Internet hosts, each host is assigned an Internet Protocol (IP) address. Currently, the vast majority of Internet traffic utilizes Internet Protocol Version 4 (IPv4) [1] which assigns 32-bit addresses to Internet hosts. As shown in Figure 1-1, the IPv4 header of packets includes the IP address of the source and destination host and many other important fields such as the *protocol* which specifies the type of transport protocol used by the sending application. The type of transport protocol determines the format of the transport protocol header following the IP header in the packet.

The second protocol produced by the original Internet Architecture project, the Transmission Control Protocol (TCP), provides a reliable transmission service for IP packets. Through the use of small acknowledgment packets transmitted from the destination host to the source host, TCP detects packet loss and regulates the transmission of packets in order to adjust to network congestion. When the source host detects a packet loss, it retransmits the lost packet or packets. At the destination host, TCP provides in-order delivery of packets to higher level protocols or applications. After the initial development of TCP, a third protocol, the User

Datagram Protocol (UDP), was added to provide additional flexibility. UDP essentially allows applications or higher level protocols to control the transmission behaviour. For example, a streaming video application may wish to ignore packet losses in order to prevent large breaks in the video stream caused by packet retransmissions. Typically, the TCP and UDP transport protocols identify applications using 16-bit port numbers carried in the transport header as shown in Figure 1-1.

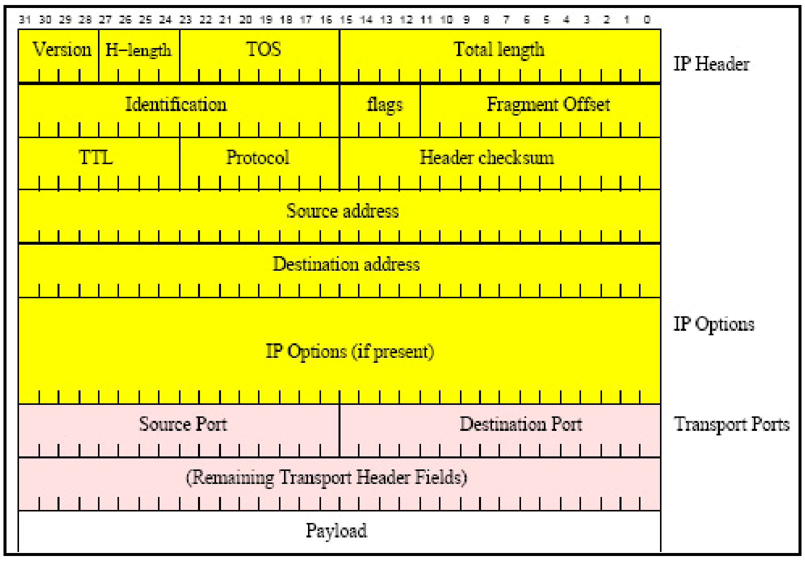


Figure 1-1 IP header format

Internet Addressing

IPv4 addresses were allocated to organizations in contiguous blocks with the intention that all hosts in the same network share a common set of initial bits. This common set of initial bits is referred to as the network address or prefix and the remaining set of bits is called the host address. This allocation strategy provided decentralized control of address allocation and each organization was free to make allocation decisions for the addresses within its assigned block. As shown in Figure 1-2, IPv4 addresses were originally divided into classes, each supporting different sizes of hosts:

- Class A (16 million hosts),
- Class B (64 thousand hosts), and
- Class C (254 hosts).
- Class D addresses for multicast (one-to-many transmission)
- Class E reserved addresses.

Most organizations which required a larger address space than Class C were allocated a block of Class B addresses; however their network nodes are assigned only a small portion of the addresses. This waste of available address space combined with the explosive growth of the Internet resulted in shortage of unassigned IP addresses. Classless Inter-Domain Routing (CIDR) was introduced in order to prolong the life of IPv4 [2]. CIDR essentially allows the "network" part of the address to be an arbitrary length prefix of the IP address, thus a network's address space may span multiple Class C networks. CIDR also allows routing protocols to aggregate network addresses in order to reduce the amount of packet forwarding information stored by each router. The wide adoption of CIDR by the Internet community has slowed the deployment of a more permanent solution, Internet Protocol Version 6 (IPv6) [3].

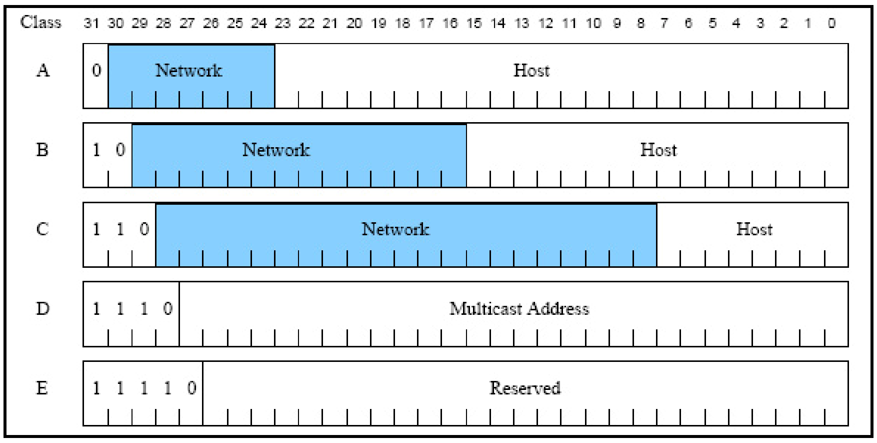


Figure 1-2 Class Based Internet Addressing

1.2 QoS in Ethernet

Ethernet is, by far the most common network, has the highest number of installed ports and provides great cost-performance ratio and thus it is making a breakthrough in MAN and WAN networks. The deployment of Gigabit Ethernet networks and their use beyond the tight borders of LANs motivated the development of QoS mechanisms in the MAC layer of Ethernet networks such as the VLAN scheme [4]. These QoS mechanisms require identification of network flows and the classification of Ethernet packets according to their MAC addresses, VLAN IDs or port numbers. The length of the MAC addresses, namely 48-bits, is what makes the decisions more difficult since exact matches in such a big value it not a trivial task. The advantage of Ethernet networks and equipment is their low cost and thus the classification solutions should also be cost efficient.

1.3 Longest Prefix Matching

The primary task of routers is to forward packets from input links to the appropriate output links. In order to do this, Internet routers consult a *route table* containing a set of network addresses together with the associated output link, or *next hop,* for packets destined for each network. Entries in the route tables change dynamically according to the state of the network and the information exchanged by routing protocols. The task of resolving the next hop from the destination IP address is commonly referred to as *route lookup* or *IP lookup*. Finding the network address given a packet's destination address would not be difficult if the early Internet Protocol (IP) address hierarchy was kept. A simple lookup in three tables, one for each Class of networks, would be sufficient. However, the wide adoption of CIDR allows the network addresses in route tables to have variable lengths (prefixes) and thus performing a search for every possible network address length is not trivial. If we store all the variable-length network addresses in a single table, a route lookup requires finding the longest matching prefix in the table for the given destination address.

A prefix is a set of leftmost bits of a key value, the IP destination address in the case of route lookups. The key values that share a common prefix have the same contiguous set of bits starting at the most significant bit. Given a search key x of size b bits, Longest Prefix Matching (LPM) is a search technique which selects the prefix p_i in the set of prefixes P, such that p_i matches x and p_i has the most specified bits. Prefixes can be represented by simply using the * character to denote the end of the valid bits in the prefix. An example of Longest Prefix Matching (LPM) for a 10-bit search key is illustrated in Figure 1-3. The three shaded prefixes match the search key, but *1000011** is the longest matching prefix. The throughput of an Internet router essentially depends on the speed that Longest Prefix Matching (LPM) operation can be performed.

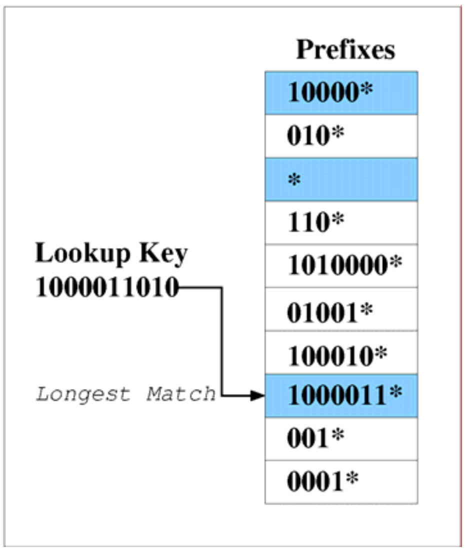


Figure 1-3 Longest Prefix Match Example

1.4 The Packet Classification Problem

If an Internet router is to provide more advanced services than packet forwarding, it must perform more fine grained flow identification. The process of identifying the packets belonging to a specific application session or group of sessions between a source and destination host or sub-network is typically referred as the packet classification problem. The route lookup problem may be also viewed as a sub-problem of the more general packet classification problem. Applications for Quality of Service, security, and monitoring typically operate on flows, thus each packet traversing a router must be classified in order to be assigned a flow identifier, *FlowID*.

Packet classification requires searching a table of filters for the highest priority or the most specific filter that matches the packet. Filters correlate a flow or set of flows to a *FlowID*. Note that filters are also referred as rules in the packet classification literature. Filters contain multiple field values that specify an exact packet header or a set of headers and the associated *FlowID* for packets matching the corresponding field values. The type of field values are typically prefixes for IP address fields, an exact value or wildcard[1] for the transport protocol and ranges for port numbers. An example filter set is shown in Table 1-1. In this simple example, filters contain field values for four packet header fields: 8-bit source (SA) and destination addresses (DA), transport protocol (PRO), and a 4-bit destination port number (PORT). The packet fields most commonly used for packet classification are

[1] Wildcards are used when we don't specify a value and want to represent all the possible values. The symbol used for wildcards is *.

also referred as the IP 5-tuple and include the 8-bit protocol, 32-bit source address, 32-bit destination address from the IPv4 header and the 16-bit source port and 16-bit destination port from the TCP and UDP transport protocol headers.

SA	DA	PORT	PRO	FlowID
11010010	*	[3:15]	TCP	1
10011100	*	[1:1]	*	2
101101*	001110*	[0:15]	*	3
10011100	01101010	[5:5]	UDP	4
*	*	[0:15]	ICMP	5
100111*	011010*	[3:15]	*	6
10010011	*	[3:15]	TCP	7
*	*	[3:15]	UDP	8
11101100	01111010	[0:15]	*	9
111010*	01011000	[6:6]	UDP	10
100110*	11011000	[0:15]	UDP	11
010110*	11011000	[0:15]	UDP	12
01110010	*	[3:15]	TCP	13
10011100	01101010	[0:1]	TCP	14
01110010	*	[3:3]	*	15
100111*	011010*	[1:1]	UDP	16

Table 1-1 Example of a filter set

The packet classification problem may be stated formally as follows:

Given a packet P containing fields P^j and a collection of filters F with each filter F_i containing fields F_i^j, select the highest priority or the most specific filter from the set , where for each filter $\forall j : F_i^j$ matches P^j.

Consider the example of searching Table 1-1 for the best matching filter and for a packet with the following header field values:

- ***SA*:** 1001 1100
- ***DA*:** 0110 1010
- ***PORT*:** 5
- ***PRO*:** UDP

The filters with *FlowIDs* 4, 6 and 8 match the packet, but *FlowID 4* is the most specific filter in all the fields. Hence, the search should return *FlowID* 4.

Packet Classification Challenges

Computational complexity is not the only challenging aspect of the packet classification problem. The increasing traffic in the Internet backbone travels over links with transmission rates in excess of one billion bits per second (1 Gb/s). Current generation fiber optic links can operate at over 40 Gb/s. The combination of transmission rate and packet size define the throughput, in terms of packets per second, routers must support. The majority of the Internet traffic utilizes the Transmission Control Protocol which transmits 40 byte acknowledgment packets. In the worst case, a router could receive a long sequence of TCP acknowledgments, therefore conservative router architects set the throughput target based on the input link rate and 40 byte packet lengths. For example, supporting 10 Gb/s links requires a throughput of 31 million packets per second per port. Modern Internet routers contain tens to thousands of ports. In such high-performance routers, route lookup and packet classification is performed on a per-port basis.

1.5 Contributions of this work

Within this work we have studied the classification tasks required by the modern networks and proposed several hardware solutions to meet the delay sensitive searching tasks required by the network infrastructures. We proposed a classification engine for the MAC layer of the Ethernet networks which uses the techniques of hashing and internal replacement of MAC Vendor IDs; *Hash Based Classification Engine (HBCE)* compacts the MAC address tables and supports high speed decisions using a modest amount of memory. Moreover we proposed a solution for the Longest Prefix Matching (LPM) problem and developed an innovative scheme; *Bitmap Oriented Strides (BOS)* uses bitmaps to compact the prefixes and reaches routing decisions in very high speeds. We have also proposed a novel packet classification scheme for the IP 5-tuple case; *Bloom Based Packet Classification (B2PC)* uses our *BOS* solution to decompose multiple-field packet classification into single fields and combine them in an efficient way by leveraging Bloom filter data structures.

1.6 Outline of the thesis

The remainder of the thesis is organized as follows. Chapter 2 provides an overview of the existing single field search techniques, including Longest Prefix Matching (LPM) techniques and a survey of multi field searching solutions that address the packet classification problem. Chapter 3 presents a classification scheme targeted to MAC Layer of Ethernet networks while a reference hardware design of this scheme is described in Chapter 4. In Chapter 5 we present BOS which is a multi-bit trie algorithmic solution to the Longest Prefix Matching problem. Chapter 6 presents our algorithm for decomposed packet classification, B2PC which utilizes Bloom filter data structures to achieve efficient packet processing. A reference hardware implementation of B2PC is described in Chapter 7. Finally, we provide a summary of the contributions and a discussion of future work in Chapter 8.

Chapter 2

Related Work

In this chapter we present the major algorithms and techniques presented in literature to address the problem of packet classification. We provide an overview of the single field searching techniques, including the longest prefix matching and other types of searches dictated by packet classification. Further, we present the most important algorithms and solutions for multi field searching that are actually used in packet classification.

2.1 Single Field Searching Techniques

A variety of searching problems naturally arise in packet classification due to the structure of packet filters. As discussed in Chapter 1, filter fields specify one of the three different match conditions on the corresponding packet header fields: a fully specified value or exact matching, partially specified value or prefix matching, a range of values or range matching. In this subsection, we provide a summary of the existing algorithmic solutions to these three types of search problems.

2.1.1 Exact Matching

The simplest form of exact matching is the set membership query: determine whether key x belongs to the set of keys X. Often we wish to store associated information with each key $x_i \in X$ such as identifiers or additional information. In such cases, a search where $x \in X$ returns not only a "yes" for the membership query, but also the information associated with the matching entry. Exact match search problems naturally arise in packet classification when filters examine packet fields such as the MAC address in the Data Link Layer. Due to the constraints on exact match searches in the networking context, namely the size of the key sets and the speed at which the search must be performed, non trivial data structures must be used for these applications.

We describe the two classical data structures that attempt to minimize the number of memory accesses per search, B-trees and hash tables. Both data structures are capable of supporting set membership queries as well as storing additional information with each key. We also provide a brief introduction to Bloom filters, a data structure designed to efficiently represent a set of keys.

2.1.1.1 B-Trees

B-Trees were originally designed to limit the number of accesses to direct access storage units such as disks [5]. The reduction in I/O operations is achieved by organizing keys in a tree data structure where the nodes of the tree may have many children. The maximum number of children of each node is referred as the *degree* of the tree. The number of keys stored in any tree node (except the root node) is bounded by the *minimum degree* of the B-Tree. Specifically, each node in the tree must contain at least $(B - 1)$ keys and at most $(2B - 1)$ keys, where $B \geq 2$. An example of a B-Tree storing the integer multiples of three is shown in Figure 2-1. The keys stored in a node are arranged in non-decreasing order and each internal node also stores a set of pointers between the keys. The child nodes that store keys greater than the parent key are pointed by the parent's "left" pointer and the children with value less or equal to the parent key are pointed by the parent's "right" pointer. Finally, the height h of a B-Tree containing n keys is bounded by:

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

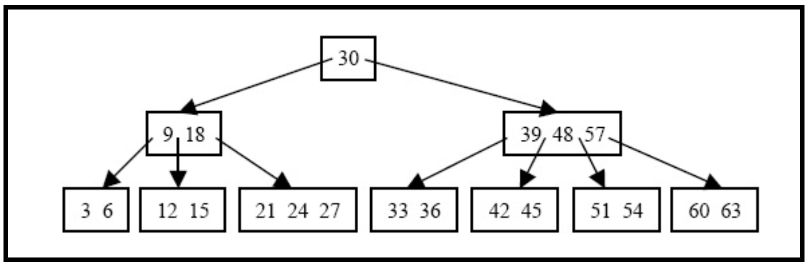


Figure 2-1 Example B-Tree data structure

2.1.1.2 Hashing

Hashing is a technique that can provide excellent average performance when the number of keys, n, in the set X is much less than the maximum number of possible keys *K.* Assume a set *X* that contains 100 keys where the keys may take any value in the range [0 : 65535], i.e. a 16-bit unsigned integer. We could simply allocate a table with 65,536 entries and use the value of the key **x** as an index into the table, but obviously this is very wasteful. This technique, *direct addressing*, is only efficient when the number of keys *n* in the set *X* approaches the number of possible key values *K***.**

The classical solution to this problem is to map the key value **x** to a narrower range of values that can be used to index a smaller table. In order to perform the mapping function, a *hash function*, *h(x)*, is computed on the key value. The resulting value is used as an index into a *hash table* of size *[0: m − 1]* where *m<<K*. Ideally, the hash function uniformly distributes all *n* keys across the *m* slots in the hash table. This search method, called *hashing*, has been extensively studied and is given thorough treatment by a number of computer science textbooks [5].

There is a variety of methods for constructing hash functions. Often, the low-order bits of key values are uniform in distribution such that the *hash index* may be constructed by selecting the low order bits of the key. Such hash functions are trivial to construct in hardware. Figure 2-2 illustrates an example of using the four low-order bits of the key as a hash index for the same integer multiples of three used in the B-Tree example in Figure 2-1.

Note that when *n* is greater than *m* or the distribution of keys across the hash table is not uniform, then *collisions* occur. In our example, we use a common collision resolution technique called *chaining*, where keys that map to the same *hash index* form a linked list. The ratio of keys to hash table slots is referred to as the *load factor*, $a = \frac{m}{n}$, which specifies the average number of keys in a chain. Thus, the average number of probes in a hash table where chaining is used for collision resolution is $1 + \alpha$. Moreover, there is a variety of much more sophisticated hash functions and collision resolution techniques presented literature and in textbooks [5].

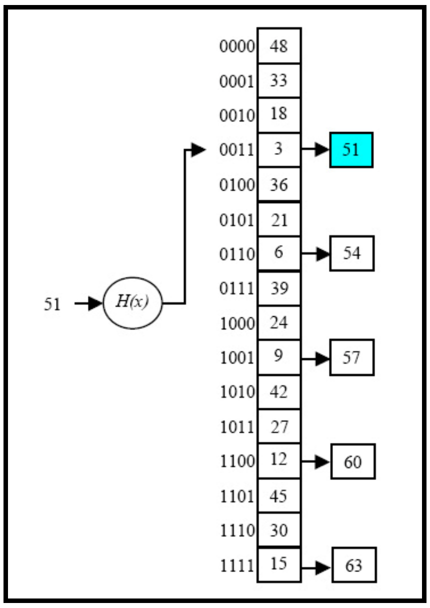


Figure 2-2 Hash function example

2.1.1.3 Bloom Filters

A Bloom filter is a data structure used for efficiently representing a set of keys. Via implicit representations of the keys in the set, the data structure supports membership queries but is not capable of storing additional information for each stored key. This technique was formulated by Burton H. Bloom in 1970 [6], and has received renewed attention in the research community for various applications such as web caching, intrusion detection, and content based routing [7].

A Bloom filter is essentially a bit-vector of length m where a key x is represented by a subset of the m bits. Given a set of keys X with n members, we insert a key $x_i \in X$ into the Bloom filter as follows. We compute k hash functions on x_i, producing k values in the range $[0 : m-1]$. Each of these values addresses a single bit in the m-bit vector, hence each key x_i causes k bits in the m-bit vector to be set to 1. Figure 2-3 provides an example of inserting two keys into a Bloom filter. Note that if one of the ***k*** hash values specifies a bit that is already set to 1, that bit is not changed.

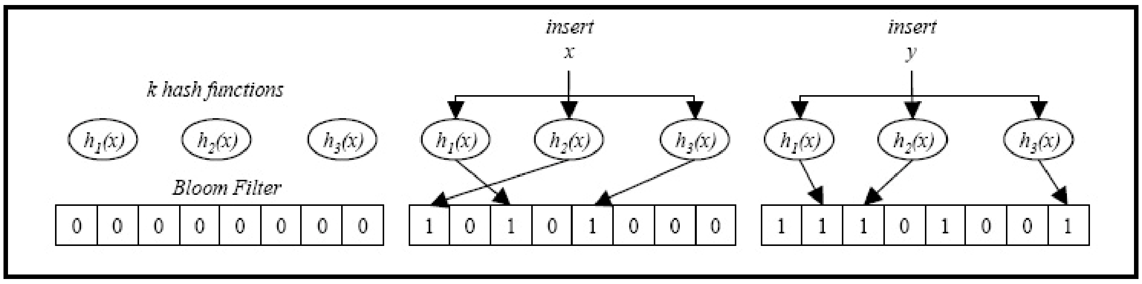


Figure 2-3 Bloom Filter Example

Querying the filter in order to determine if a given key *x* belongs to the set *X* is similar to the insertion process. Given key *x*, we generate *k* hash indices using the same hash functions used to insert keys into the filter. We check the bit locations corresponding to the *k* hash indices in the *m*-bit vector. If at least one of the *k* bits is 0, then it denotes that the key is not a member of the set. If all the bits are found to be 1, then we claim that the key belongs to the set with a certain probability. If we find all *k* bits to be 1 and *x* is not a member of *X*, then it is said to be a *false positive* match. This ambiguity in membership comes from the fact that the *k* bits in the *m*-bit vector can be set by any of the *n* members of *X*. Thus, finding a bit set to 1 does not necessarily imply that it was set by the particular key being queried. However, finding a 0 bit certainly implies that the key does not belong to the set, since if it was a member then all *k*-bits would have been set to 1 when the key was inserted into the Bloom filter.

The following is a derivation of the probability of a false positive match, *f*. The probability that a random bit of the *m*-bit vector is set to 1 by a hash function is simply $\frac{1}{m}$. The probability that it is not set is $1-\frac{1}{m}$. The probability that it is not set by any of the *n* members of *X* is $\left(1-\frac{1}{m}\right)^{nk}$. Hence, the probability that this bit is set is $1-\left(1-\frac{1}{m}\right)^{nk}$. For a key to be declared a possible member of the set, all *k* bit locations generated by the hash functions need to be 1. The probability that this happens, *f*, is given by

$$f = \left(1-\left(1-\frac{1}{m}\right)^{nk}\right)^{k}$$

For large values of *m* the above equation reduces to

$$f \approx \left(1-e^{\frac{-nk}{m}}\right)^{k}$$

Since this probability is independent of the input key, it is termed the *false positive* probability. The false positive probability can be reduced by choosing appropriate values for *m* and *k* for a given size of the member set, *n*. For a given ratio of $\frac{m}{n}$, the false positive probability can be reduced by adjusting the number of hash functions, *k*.

In the optimal case, when false positive probability is minimized with respect to *k*, we get the following relationship:

$$k = \left\{ \left\lfloor \frac{m}{n} \ln 2 \right\rfloor, \left\lceil \frac{m}{n} \ln 2 \right\rceil \right\}$$

The false positive probability at this optimal point is given by

$$f = \left(\frac{1}{2} \right)^k$$

It should be noted that if the false positive probability is to be tuned, then the size of the filter, *m*, needs to scale linearly with the size of the key set, *n*.

One property of Bloom filters is that it is not possible to delete a key stored in the filter. Deleting a particular entry requires that the corresponding *k* hashed bits in the bit vector be set to zero, which would disturb other keys programmed into the filter which hash to any of these bits. In order to solve this problem the idea of the *Counting Bloom Filter* was proposed by Fan, et.al. [8]. A Counting Bloom Filter maintains a vector of counters corresponding to each bit in the bit-vector. Whenever a key is added to or deleted from the filter, the counters corresponding to the *k* hash values are incremented or decremented, respectively. When a counter changes from one to zero, the corresponding bit in the bit-vector is cleared. Note that maintaining counters significantly increases the storage requirements.

2.1.2 Longest Prefix Match

Longest Prefix Matching (LPM) has received significant attention in the literature over the past ten years. This is due to the fundamental role it plays in the performance of Internet routers. Due to the explosive growth of the Internet, Classless Inter-Domain Routing (CIDR) was widely adopted to prolong the life of Internet Protocol Version 4 (IPv4) [2]. Use of this protocol requires Internet routers to search variable-length address prefixes in order to find the longest matching prefix of the IP destination address and retrieve the corresponding forwarding information, or “next hop”, for each packet traversing the router. This computationally intensive task, commonly referred to as IP Lookup, is often the performance bottleneck in high-performance Internet routers.

2.1.2.1 Linear Search

If the set of prefixes is small, a linear search through a list of the prefixes sorted in order of decreasing length is sufficient. The sorting step guarantees that the first matching prefix in the list is the longest matching prefix for the given search key. Linear search is the most memory efficient of all LPM techniques and the memory requirements are **O(N)** where **N** is the number of prefixes in the table. Note that the search time is also **O(N)**, thus linear search is not practical for IP lookup when the set of prefixes is relatively large.

2.1.2.2 Content Addressable Memory (CAM)

Many commercial router designers have chosen to use Content Addressable Memory (CAMs) for IP address lookups in order to keep up with the latest optical link speeds despite their larger size, cost, and power consumption relative to Static Random Access Memory (SRAM). CAMs minimize the number of memory accesses required to locate an entry by comparing the input key against all memory words in parallel; hence, a lookup effectively requires one clock cycle. While binary CAMs perform well for exact match operations and can be used for route lookups in strictly hierarchical addressing schemes [9], the wide use of address aggregation techniques like CIDR requires storing and searching entries with arbitrary prefix lengths. In response, Ternary Content Addressable Memories (TCAMs) were developed with the ability to store an additional “Don’t Care” state which allows them to ensure single clock cycle lookups for arbitrary prefix lengths.

2.1.2.3 Trie Based Schemes

Search techniques which build decision trees use the bits of prefixes to make branching decisions and allow the worst-case search time to be independent of the number of prefixes in the set. An example of a binary trie constructed from a set of prefixes is shown in Figure 2-4. Shaded nodes denote a stored prefix with the corresponding next hop shown next to the node. A search is conducted by traversing the trie using the bits of the address, starting with the most significant bit. Note that the worst-case search time is now **O(W)**, where **W** is the length of the address and maximum prefix size in bits.

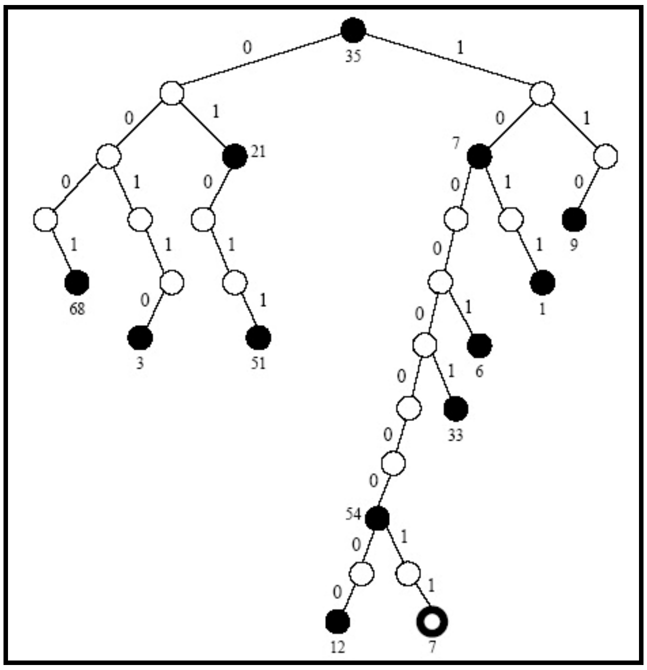


Figure 2-4 Binary Trie example

One of the first IP lookup techniques to employ *tries*[2] is Sklower's implementation of a Patricia trie in the BSD kernel [10]. The Patricia trie is a binary radix tree that compresses paths with one-way branching into a single node. It assumes contiguous masks and bounds the worst case lookup time to **O(W)**. While paths may be compressed, only one bit of the address is examined at a time during a search resulting in search rates that do not meet the needs of high-performance routers.

In order to speed up the lookup process, multi-bit trie schemes were developed which perform a search using multiple bits of the address at a time. Srinivasan and Varghese introduced two important techniques for multi-bit trie searches, *Controlled Prefix Expansion* (CPE) and *Leaf Pushing* [11]. *Controlled Prefix Expansion* restricts the set of distinct prefix lengths by "expanding" prefixes shorter than the next distinct length into multiple prefixes. This allows the lookup to proceed as a direct index lookup into tables corresponding to the distinct prefix length, or stride length, until the longest match is found. The technique of *Leaf Pushing* reduces the amount of information stored in each table entry by "pushing" information about the best matching prefix along the paths to leaf nodes. As a result each leaf node needs only to store a pointer or next hop information. While this technique reduces memory usage,

[2] Trie is the term used for trees in information retrieval data structures. It originates from the word re**trie**val.

it also increases incremental update overhead. The authors also discuss variable length stride lengths, optimal selection of stride lengths, and dynamic programming techniques.

Gupta, Lin, and McKeown simultaneously developed a special case of CPE specifically targeted to hardware implementation [12]. Arguing that DRAM is such a plentiful and inexpensive resource, their technique spends large amounts of memory in order to limit the number of off-chip memory accesses to two or three. Their basic scheme is a two level "expanded" trie with an initial stride length of 24 and second level tables of stride length eight. Given that random accesses to DRAM may require up to eight clock cycles and current DRAMs operate at less than half the speed of SRAMs, this technique can be out-performed by techniques utilizing SRAM and requiring less than 10 memory accesses.

Other techniques such as *Lulea* [13] and Eatherton and Dittia's *Tree Bitmap* [14] employ multi-bit tries with compressed nodes. The *Lulea* scheme essentially compresses an expanded, leaf-pushed trie with stride lengths 16, 8, and 8. In the worst case, the scheme requires 12 memory accesses; however, the data structure only requires a few bytes per entry. While extremely compact, the *Lulea* scheme's update performance suffers from its implicit use of leaf pushing. The *Tree Bitmap* technique avoids leaf pushing by maintaining compressed representations of the prefixes stored in each multi-bit node. It also employs a clever indexing scheme to reduce pointer storage to two pointers per multi-bit node.

2.1.2.4 Multiway and Multicolumn Search

Several other algorithms exist with attractive properties that are not based on tries. The *Multiway and Multicolumn Search* techniques presented by Lampson, Srinivasan, and Varghese are designed to optimize performance for software implementations on general purpose processors [15]. The primary contribution of this work is mapping the longest matching prefix problem to a binary search over the fixed-length endpoints of the ranges defined by the prefixes. By specifying a set of contiguous initial bits, prefixes define ranges of values. For example, if *10** is a prefix for a four bit field, then it defines the range [1000:1011]. Prefixes never define overlapping ranges, only nested ranges. For example, [0:3] and [2:4] are overlapping ranges, whereas [0:3] and [1:2] are nested ranges. The authors use this property to develop a binary search technique over the endpoints of the ranges defined by the prefixes.

2.1.2.5 Binary Search on Prefix Lengths

The most efficient lookup algorithm known, from a theoretical perspective, is *Binary Search on Prefix Lengths* which was introduced by Waldvogel, et. al.[16]. The number of steps required by this algorithm grows logarithmically with the length of the address, making it particularly attractive for IPv6, where address lengths increase to 128 bits. However, the algorithm is relatively complex to implement, making it more suitable for software rather than hardware implementation. It also does not readily support incremental updates.

This technique bounds the number of memory accesses via significant pre-computation of the route table. First, the prefixes are sorted into sets based on prefix length, resulting in a maximum of *W* sets to examine for the best matching prefix. A hash table is built for each set, and it is assumed that examination of a set requires one hash probe. The basic scheme selects the sequence of sets to probe using a binary search on the sets beginning with the median length set. For example: for an IPv4 database with prefixes of all 32 lengths, the search begins by probing the set with length 16 prefixes. Prefixes of longer lengths direct the search to its set by placing "markers" in the shorter sets along the binary search path. Accordingly, a 24-length prefix would have a "marker" in the length 16 set. Therefore, at each set the search selects the longer set on the binary search path if there is a matching marker directing it lower. If there is no matching prefix or marker, then the search continues at the shorter set on the binary search path.

The use of markers introduces the problem of "backtracking": having to search the upper half of the trie because the search followed a marker for which there is no matching prefix in a longer set for the given address. In order to prevent this, the best-matching prefix for the marker is computed and stored with the marker. If a search terminates without finding a match, the best-matching prefix stored with the most recent marker is used to make the routing decision. The authors also propose methods of optimizing the data structure based on the statistical characteristics of the route table. For all versions of the algorithm, the worst case bounds are **O(logWdist)**time and **O(N×logWdist)** space where **Wdist** is the number of unique prefix lengths.

Empirical measurements using an IPv4 route table resulted in memory requirement of about 42 bytes per entry.

2.1.3 All Prefix Matching (APM)

Longest Prefix Matching (LPM) is a special case of the general All Prefix Matching (APM) problem. Instead of returning just the longest matching prefix, the APM problem requires that all matching prefixes are returned. This problem arises when multi-filed search techniques are decomposed into several instances of single-field search techniques.

Note that most trie-based algorithms easily map to the APM problem. The algorithm can simply return all matching prefixes along the path to the longest matching prefix. While the trie-based algorithms easily map to APM, it is important to note that the *Binary Search on Prefix Lengths* and *Multiway and Multicolumn Search* techniques do not readily support APM. The use of markers in *Binary Search on Prefix Lengths* naturally directs searches to longer prefixes before examining shorter length prefixes. The same consequence is experienced by the *Multiway and Multicolumn Search* due to the binary search over range endpoints. In order to support APM searches using these techniques, we must use a general technique that allows any LPM algorithm to perform APM.

2.1.4 Range Matching

Range matching problems naturally arise in many searching problems in the areas of networking and database design, and there are several forms of range matching problems. In this subsection we describe the most widely used approaches to address the following problem that arises in packet classification: Given a set *X* of closed intervals *[i, j]* and a point *p*, find all the intervals in *X* that contain *p*. This task is an essential part of packet classification, as packet filters may specify ranges for the source and destination port numbers in packet headers in order to identify a set of applications. Solutions to this problem typically employ a variant of the Interval Tree [17] or convert each closed interval *[i,j]* into a set of prefixes and then employ one of the Longest Prefix Matching (LPM) algorithms.

2.1.4.1 Interval Tree

An Interval Tree stores a set of closed intervals *X* using a balanced binary tree as the underlying data structure [5]. Each node in the Interval Tree stores an interval *x* ∊ *X*. The low endpoint of the interval is used as the key for the node in the balanced binary search tree. In order to facilitate faster searches, tree nodes typically store additional variables such as the maximum value of all the endpoints of the ranges stored in their sub-tree. An example of an Interval Tree is shown in Figure 2-5.

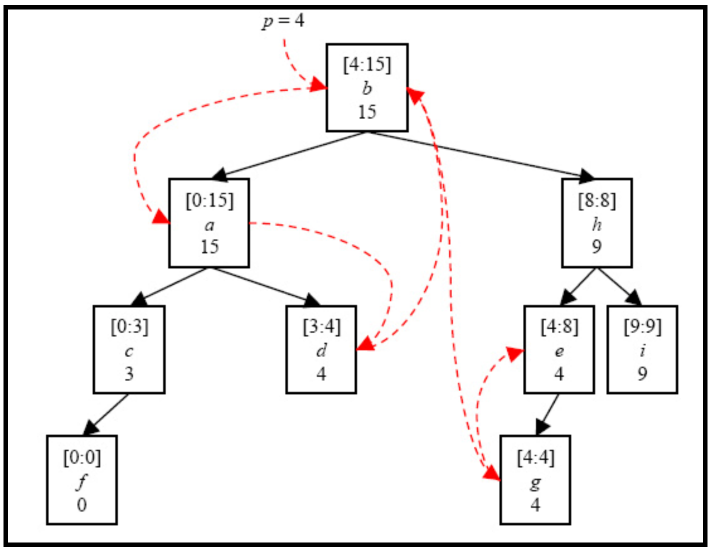


Figure 2-5 Interval Tree example

Searching for one matching interval for a given point *p* is straight-forward, but returning the set *S* of all matching intervals for *p* requires a few extra steps. We first locate the matching interval for p that is stored at the leftmost node in the tree. From this node, we perform an in-order walk of the tree nodes, stopping when we arrive at the last node in the tree or a node whose key is greater than *p*. An example search for *p =4* is shown in Figure 2-5. Letting *S* be the number of matching intervals, the search requires **O(logX + S)** time.

2.1.4.2 Range to Prefix Conversion

Prefixes define exactly one range on the real numbers. The low and high endpoints of the range defined by a prefix are the minimum and maximum points covered by the prefix. For binary numbers, this translates to replacing the masked bits of the prefix with zeros and ones, respectively. For example, the four bit prefix *11** defines the range [1100:1111] or [12:15]. This transform operation is not symmetric, as an arbitrary range may specify multiple prefixes. Specifically, a range defined on the set of b-bit numbers will specify at most $[2 \times (b - 1)]$ prefixes.

For a single-field search on a reasonable number of ranges, this expansion factor is not prohibitive. As a result, several packet classification techniques use the range to prefix conversion technique to solve the range matching sub-problem [18], [19]. Finally, we note that Feldman and Muthukrishnan [17] provide a range to prefix conversion technique for the special case of searching *elementary intervals* by converting them into prefixes. They show that a set of $(n - 1)$ *elementary intervals* can be converted into a set prefixes containing at most $2n$ prefixes, where an LPM search is used to select the *elementary interval* containing a given point p.

2.2 Multi Field Searching Techniques

In this subsection we provide a summary of the major multiple field search techniques aimed at packet classification. Due to the complexity of the search, packet classification is often a performance bottleneck in network infrastructure and thus it has received significant attention by the research community. Many algorithms and classification schemes have been proposed with numerous different approaches. These techniques can be categorized according to the high level approach of the classification solution. We can consider that there are three main different high-level approaches:

- **Exhaustive Search**: examines all entries in the filter set.
- **Decision Tree**: construct decision trees from the filters in the filter set and use the packet fields to traverse the decision trees.
- **Decomposition**: decompose the multiple field search into instances of single field searches, perform independent searches on each packet field and then combine the results.

2.2.1 Exhaustive Search

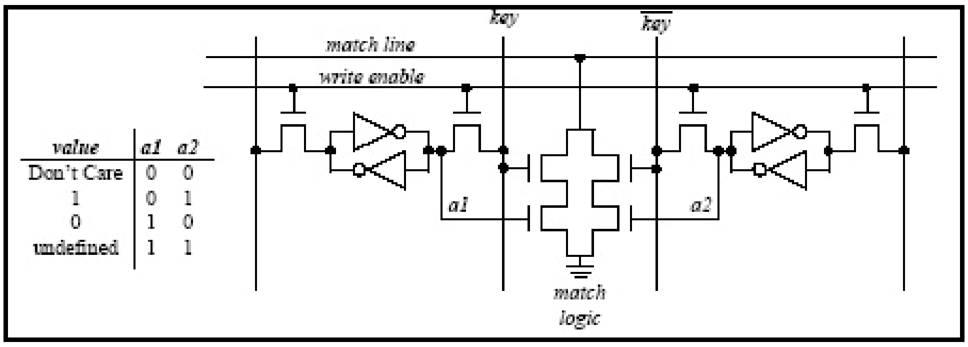
The fundamental solution to any searching problem is simply to search through all the entries in the set. The two most common exhaustive search approaches for packet classification are a linear search through a list of filters or a parallel search over the set assuming that it is divided into a number of subsets. These are extreme solutions, where the lowest performance option, linear search, does not divide the set into subsets and the highest performance option, Ternary Content Addressable Memory (TCAM), completely divides the set into subsets containing only one entry. We discuss both of these solutions in detail below.

2.2.1.1 Linear Search

Performing a linear search through a list of filters has **O(N)** storage requirements, but it also requires **O(N)** memory accesses per lookup. Even in the smaller filter sets, linear search becomes very slow. It is possible to reduce the number of memory accesses per lookup by partitioning the list into sub-lists and pipelining the search where each stage searches a sub-list. Note that linear search can be popular solution for the final stage of a lookup when the set of possible matching filters has been drastically reduced [19][20][21].

2.2.1.2 Ternary Content Addressable Memory (TCAM)

Alike fully-associative cache memories, Ternary Content Addressable Memory (TCAM) devices perform a parallel search over all filters in the filter set. TCAMs were developed with the ability to store a "Don't Care" state in addition to a binary digit. A typical TCAM cell is shown in Figure 2-6. Input keys are compared against every TCAM entry which enables them to ensure single clock cycle lookups for arbitrary bit mask matches.



value	a1	a2
Don't Care	0	0
1	0	1
0	1	0
undefined	1	1

Figure 2-6 A typical TCAM cell

Despite their astonishing efficiency, TCAMs have four primary drawbacks:

1. high cost per bit relative to other memory technologies; current TCAMs cost about 20 times more per bit of storage than DDR SRAMs.
2. storage waste, in addition to the six transistors required for binary digit storage, a typical TCAM cell requires an additional six transistors to store the

mask bit and four transistors for the match logic, resulting in a total of 16 transistors; some very efficient solutions use 14 transistors.

3. high power consumption; the massive parallelism in TCAM architectures is the main source of high power consumption. Each “bit” of TCAM match logic must drive a match word line which signals a match for the given key. The extra logic and capacitive loading results in access times approximately three times longer than SRAM. Specifically, TCAMs consume 150 times more power per bit than SRAM.
4. limited scalability to long input keys; TCAMs can only match keys of maximum length equal to the word size.

2.2.2 Decision Trees

Another popular approach to packet classification on multiple fields is to construct a decision tree where the leaves of the tree contain filters or subsets of filters. In order to perform a search using a decision tree, we construct a search key from the packet header fields. We traverse the decision tree by using individual bits or subsets of bits from the search key to take branching decisions at each node of the tree. The search continues until we reach a leaf node storing the best matching filter or subset of filters. Decision tree construction is complicated due to the fact that a filter may specify several different types of searches. The mix of Longest Prefix Match, arbitrary range match, and exact match filter fields significantly complicates the branching decisions at each node of the decision tree. A common solution to this problem is to convert all the filter fields to a single type of match.

2.2.2.1 Grid of Tries

Srinivasan, Varghese, Suri, and Waldvogel introduced the original *Grid-of-Tries* algorithm for packet classification [22]. *Grid-of-Tries* applies a decision tree approach to the problem of packet classification on source and destination address prefixes. For filters defined by source and destination prefixes, *Grid-of-Tries* improves the directed acyclic graph (DAG) technique introduced by Decasper, Dittia, Parulkar, and Plattner [23]. This technique is also called set pruning trees because redundant sub-trees can be “pruned” from the tree by allowing multiple incoming edges at a node. While this optimization does eliminate redundant sub-trees, it does not completely eliminate replication as filters may be stored at multiple nodes in the tree. *Grid-of-Tries* eliminates this replication by storing filters at a single node and using *switch pointers* to direct searches to potentially matching filters.

Consider the filter set shown in Table 2-1 where source and destination address prefixes for each rule are defined. Moreover, assume we are searching for the best matching filter for a packet with source and destination addresses equal to 0011.

Filter	Source Address	Destination Address
F1	0*	10*
F2	0*	01*
F3	0*	1*
F4	00*	1*
F5	00*	11*
F6	10*	1*
F7	*	00*
F8	0*	10*
F9	0*	1*
F10	0*	10*
F11	111*	000*

Table 2-1 Example filter set for Grid of Tries

In the Grid-of-Tries structure shown in Figure 2-7, we find the longest matching source address prefix 00* and follow the pointer to the destination address tree. Since there is no 0 branch at the root node, we follow the switch pointer to the 0* node in the destination address tree for source address prefix 0*. Since there is no branch for 00* in this tree, we follow the switch pointer to the 00* node in the destination address tree for source address prefix *. Here we find a stored filter F7 which is the best matching filter for the packet.

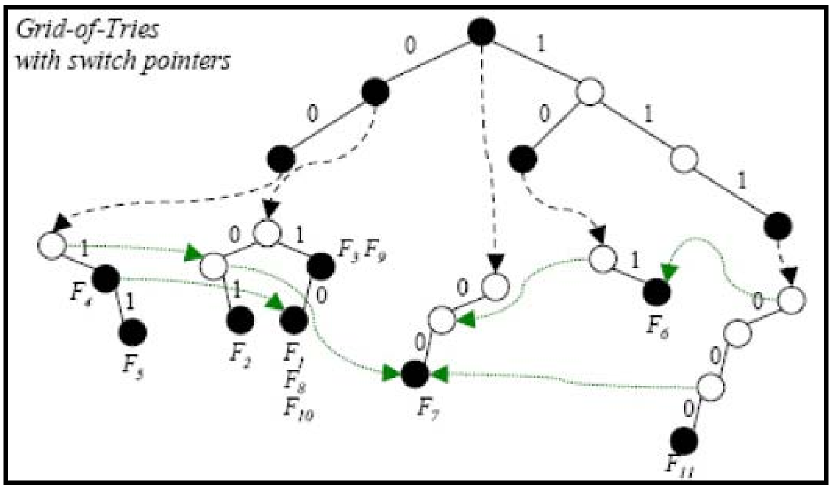


Figure 2-7 Grid of Tries data structure

Grid-of-Tries bounds memory usage to **O(NW)** while achieving search time of **O(W)**, where *N* is the number of filters and *W* is the maximum number of bits specified in the source or destination fields. For the case of searching on IPv4 source and destination address prefixes, the measured implementation uses multi-bit tries sampling 8 bits at a time for the destination trie; each of the source tries starts with a 12 bit node, followed by 5 bit trie nodes. This yields a worst case of 9 memory accesses; the authors claim that this could be reduced to 8 with an increase in storage.

2.2.2.2 Hierarchical Intelligent Cuttings (HiCuts)

Gupta and McKeown introduced an innovative technique called *Hierarchical Intelligent Cuttings* (*Hi-Cuts*) [20]. The concept of "cutting" comes from viewing the packet classification problem geometrically. Each filter in the set defines a *d*-dimensional rectangle in *d*-dimensional space, where *d* is the number of fields in the filter. Selecting the decision criteria translates into choosing a partitioning, or "cutting", of the space. Consider the example filter set in Table 2-2 consisting of filters with two fields: a 4-bit address prefix and a port range covering 4-bit port numbers. This set is shown geometrically in Figure 2-8.

Filter	Address	Port
a	1010	2
b	1100	5
c	0101	8
d	*	6
e	11*	0-15
f	001*	9-15
g	00*	0-4
h	0*	0-3
i	0110	0-15
j	1*	7-15
k	0*	11

Table 2-2 Example filter set for HiCuts

HiCuts pre-processes the filter set in order to build a decision tree with leaves containing a small number of filters bounded by a threshold. Packet header fields are used to traverse the decision tree until a leaf is reached. The filters stored in that leaf are then linearly searched for a match. *HiCuts* converts all filter fields to arbitrary ranges, avoiding filter replication. The algorithm uses various heuristics to select decision criteria at each node that minimizes the depth of the tree while controlling the amount of memory used.

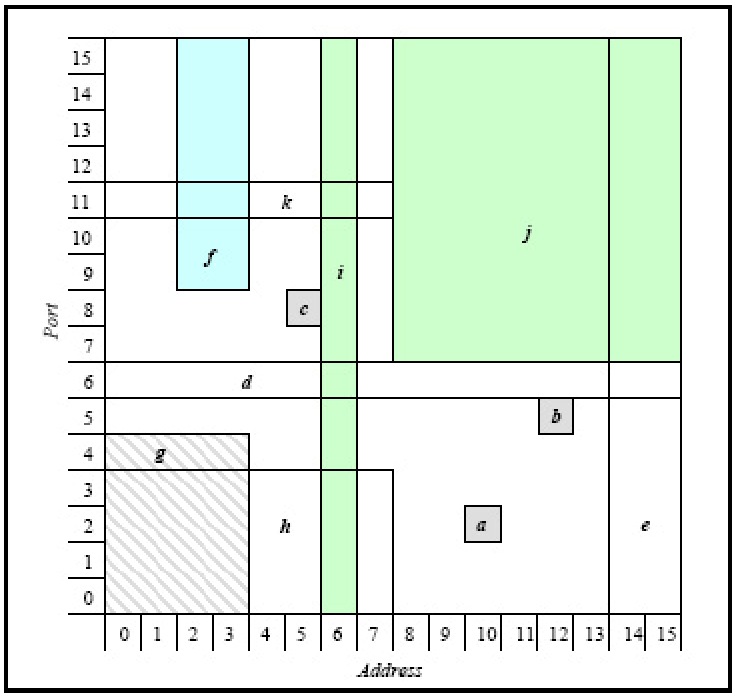


Figure 2-8 HiCuts geometric representation

A *HiCuts* data structure for the example filter set in Table 2-2 is shown in Figure 2-9. Each tree node covers a portion of the *d*-dimensional space and the root node covers the entire space. In order to keep the decisions at each node simple, each node is cut into equal sized partitions along a single dimension. For example, the root node in Figure 2-9 is cut into four partitions along the *Address* dimension. In this example, we have set the thresholds such that a leaf contains at most two filters and a node may contain at most four children. The authors describe a number of more sophisticated heuristics and optimizations for minimizing the depth of the tree and the memory resource requirement.

Experimental results in the two-dimensional case show that a filter set of 20k filters requires 1.3MB with a tree depth of 4 in the worst case and 2.3 on average. Experiments with four-dimensional classifiers used filter sets ranging in size from approximately 100 to 2000 filters. Memory consumption ranged from less than 10KB to 1MB, with associated worst case tree depths of 12 (20 memory accesses). Due to the considerable pre-processing required, this scheme does not readily support incremental updates.

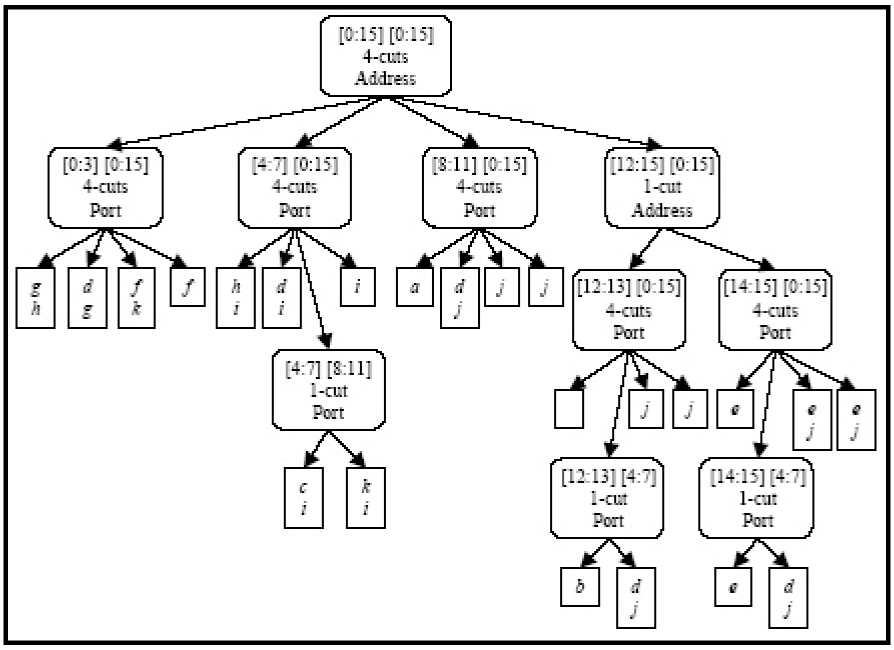


Figure 2-9 HiCuts Data Structure

2.2.2.3 Fat Inverted Segment (FIS) Trees

Feldman and Muthukrishnan introduced a scheme for packet classification using independent field searches on Fat Inverted Segment (FIS) Trees [17]. FIS Trees utilize a geometric view of the filter set and map filters into *d*-dimensional space. Projections from the "edges" of the *d*-dimensional rectangles specified by the filters define elementary intervals on the axes. N filters will define a maximum of $I = (2N + 1)$ elementary intervals on each axis. A FIS Tree is a balanced *t*-ary tree with *k* levels that stores a set of segments, or ranges. Note that $t = (2I + 1)^{1/k}$ is the maximum number of children a node may have. The leaf nodes of the tree correspond to the elementary intervals on the axis. Each node in the tree stores a canonical set of ranges such that the union of the canonical sets at the nodes visited on the path from the leaf node associated with the elementary interval. Covering a point *p* to the root node is the set of ranges containing *p*.

Using the example filter set shown in Table 2-2 we present an overview of FIS in Figure 2-10. The scheme starts by building an FIS Tree on one axis. For each node with a non-empty canonical set of filters, we construct an FIS Tree for the elementary intervals formed by the projections of the filters in the canonical set on the next axis (filter field) in the search. The authors propose a method of using a Longest Prefix Matching technique to locate the elementary interval covering a given point. This method requires at most *2I* prefixes.

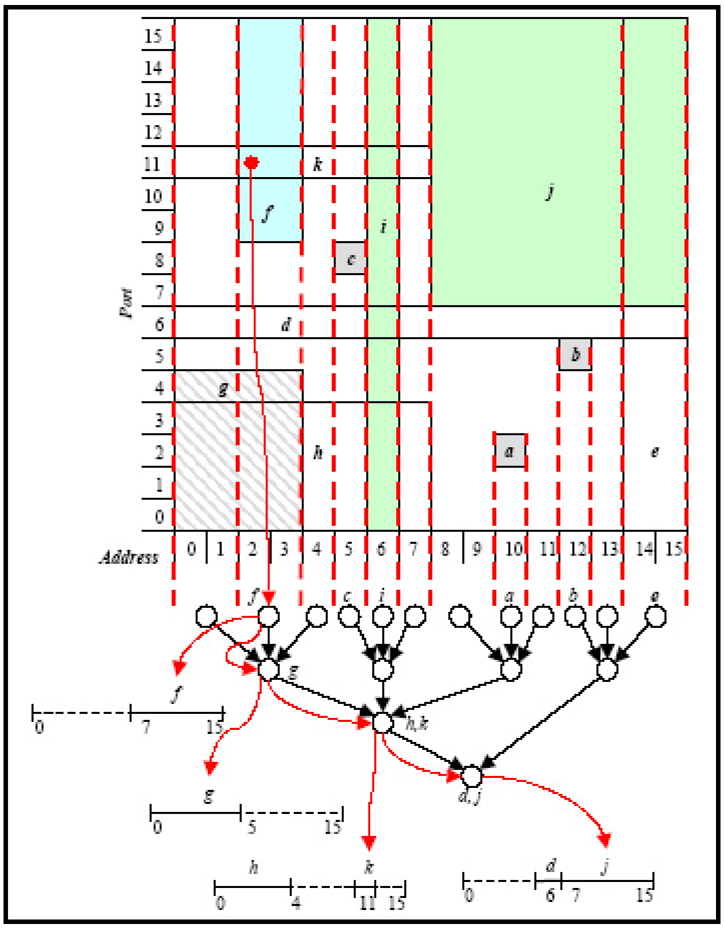


Figure 2-10 FIS example

Figure 2-10 also provides an example search for a packet with address 2, and port number 11. A search begins by locating the elementary interval covering the first packet field, interval [2:3] on the Address axis in our example. The search proceeds by following the parent pointers in the FIS Tree from leaf to root node. Along the path, we follow pointers to the sets of elementary intervals formed by the Port projections and search for the covering interval. Throughout the search, we remember the highest priority matching filter. The authors performed simulations with real and synthetic 78 filter sets containing filters classifying on source and destination address prefixes. For filter sets ranging in size from 1K to 1M filters, memory requirements ranged from 100 to 60 bytes per filter. Lookups required between 10 and 21 cache-line accesses which amounts to 80 to 168 word accesses, assuming 8 words per cache line.

2.2.3 Decomposition

Given the option of efficient single field search techniques, decomposing a multiple field search problem into several instances of a single field search problem is a practical approach. Employing this high-level approach has several advantages. First, each single field search engine operates independently, thus we have the opportunity to exploit the parallelism offered by modern hardware. Performing each search independently also offers more degrees of freedom in optimizing each type of search on the packet field.

Despite these advantages, decomposing a multi-field search problem creates other complicated issues. The primary challenge is to efficiently aggregate and combine the results of the single field searches. Moreover, the longest matching prefix for a given filter field is not sufficient as a result from the single field search engines. The best matching filter may contain a field which is not necessarily the longest matching prefix relative to other filters; it may be more specific or have higher priority in other fields. As a result, techniques employing decomposition try to take advantage of filter set characteristics that allow them to limit the number of intermediate results. In general, solutions using decomposition provide high throughput due to their parallel hardware implementations. The high level of lookup performance often comes at the cost of memory waste.

2.2.3.1 Parallel Bit Vectors (BV)

Lakshman and Stiliadis introduced one of the first multiple field packet classification algorithms targeted to a hardware implementation. Their technique is commonly referred to as the Lucent bit-vector scheme or *Parallel Bit-Vectors* (*BV*) [24]. The authors make the initial assumption that the filters are sorted according to priority. *Parallel BV* utilizes a geometric view of the filter set and maps filters into *d*-dimensional space. As shown in Figure 2-11, projections from the "edges" of the *d*-dimensional rectangles specified by the filters define elementary intervals on the axes. Note that we are using the example filter set shown in Table 2-2 where filters contain two fields: a 4-bit address prefix and a range covering 4-bit port numbers. N filters define at maximum *(2N+1)* elementary intervals on each axis.

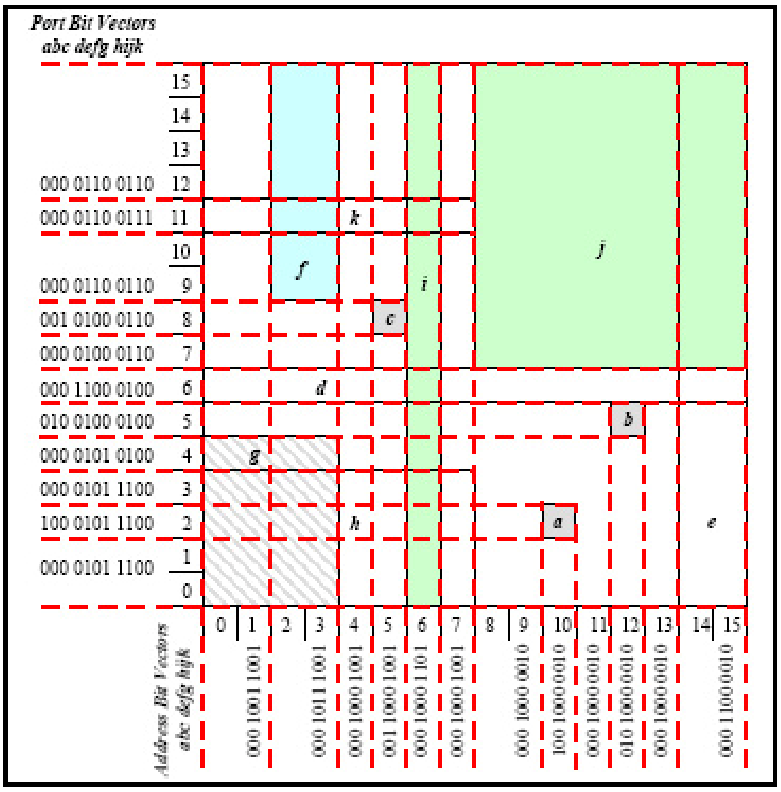


Figure 2-11 Parallel Bit Vectors example

For each elementary interval on each axis an N-bit bit-vector is defined. Each bit position corresponds to a filter in the filter set, sorted by priority. All bit-vectors are initialized to all ‘0’s. For each bit-vector, we set the bits corresponding to the filters that overlap the associated elementary interval. Consider the interval [12:15] on the *Port* axis in Figure 2-11. Assume that sorting the filters according to priority places them in alphabetical order. Filters e, f, i, and j overlap this elementary interval; therefore, the bit-vector for that elementary interval is 00001100110 where the bits correspond to filters a through k in alphabetical order. For each dimension *d*, we construct an independent data structure that locates the elementary interval covering a given point, then we return the bit-vector associated with that interval. The authors utilize binary search, but any range location algorithm is suitable.

Once we compute all the bit-vectors and construct the *d* data structures, searches are relatively simple. We search the *d* data structures with the corresponding packet fields independently. Once we have all *d* bit vectors from the field searches, we simply perform the bit-wise *AND* of all the vectors. The most significant ‘1’ bit in the result denotes the highest priority matching filter. Multiple matches are easily supported by examining the most significant set of bits in the resulting bit vector.

The authors implemented a five field version with five 128Kbyte SRAMs. This configuration supports 512 filters and performs one million lookups per second. Assuming a binary search technique over the elementary intervals, the general *Parallel BV* approach has **O(lgN)** search time and **O(N^2)** memory requirement. The authors have further proposed an algorithm to reduce the memory requirement to **O(NlogN)** using incremental reads.

2.2.3.2 Aggregated Bit-Vector (ABV)

Baboescu and Varghese introduced the *Aggregated Bit-Vector* (ABV) algorithm which seeks to improve the performance of the *Parallel BV* technique by using statistical observations of real filter sets [25]. Conceptually, *ABV* starts with *d* sets of *N*-bit vectors constructed in the same manner as in *Parallel BV*. The authors leverage the widely known property that the maximum number of filters matching a packet is inherently limited in real filter sets. This property causes the *N*-bit vectors to be sparse. In order to reduce the number of memory accesses, *ABV* essentially partitions the *N*-bit vectors into *A* chunks and only retrieves chunks containing '1' bits. Each chunk is *N / A* bits in size and has an associated bit in an *A*-bit aggregate bit-vector. If any of the bits in the chunk are set to '1', then the corresponding bit in the aggregate bit-vector is set to '1'. Figure 2-12 provides an example using the filter set in Table 2-2.

Each independent search on the *d* packet fields returns an *A*-bit aggregate bit-vector. We perform the bit-wise *AND* on the aggregate bit-vectors. For each '1' bit in the resulting bit-vector, we retrieve the *d* chunks of the original *N*-bit bit-vectors from memory and perform a bit-wise *AND*. Each '1' bit in the resulting bit-vector denotes a matching filter for the packet. *ABV* also removes the strict priority ordering of filters by storing each filter's priority in an array. This allows us to reorder the filter in order to cluster '1' bits in the bit-vectors. This in turn reduces the number of memory accesses. Simulations with real filter sets show that *ABV* reduced the number of memory accesses relative to *Parallel BV* by a factor of a four. Simulations with synthetic filter sets show more dramatic reductions by a factor of 20 or more when the filters sets do not contain any wildcards. As wildcards increase, the reductions become much more modest.

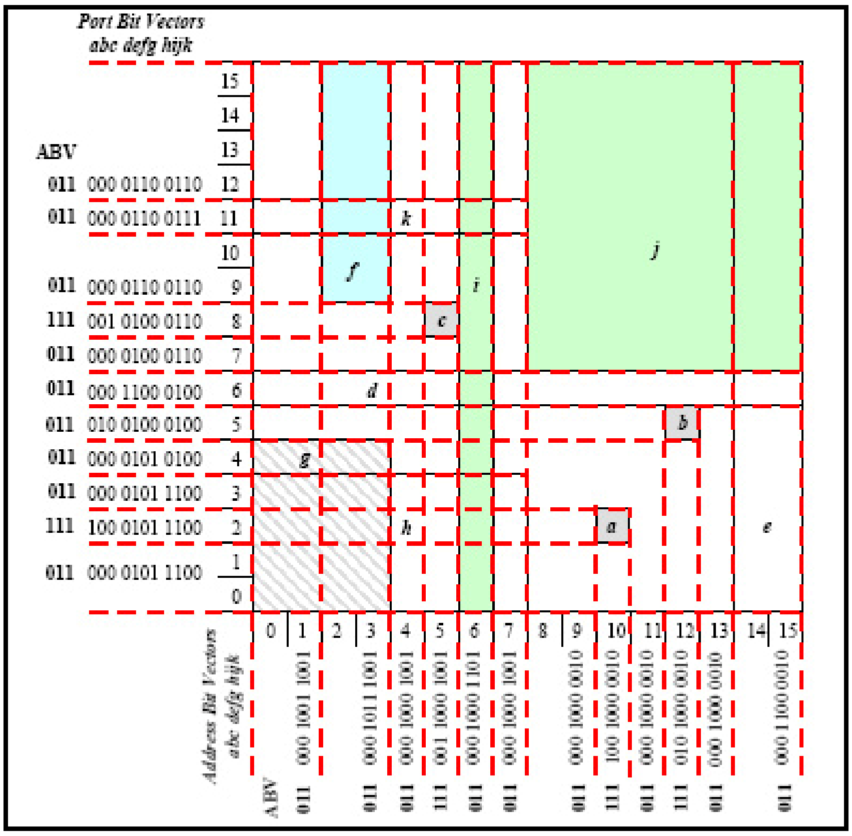


Figure 2-12 Aggregated Bit Vector example

2.2.3.3 Recursive Flow Classification (RFC)

Leveraging observations on real filter sets, Gupta and McKeown introduced *Recursive Flow Classification* (RFC) which provides high lookup rates at the cost of memory inefficiency [26]. The authors introduced a unique high-level view of the packet classification problem. Essentially, packet classification can be viewed as the reduction of an *m*-bit string defined by the packet fields to a *k*-bit string specifying the set of matching filters for the packet or action to apply to the packet. For classification on the IPv4 5-tuple, *m* is 104 bits and *k* is typically on the order of 10 bits. The authors also performed a rather comprehensive and widely cited study of real filter sets and extracted several useful properties. Specifically, they noted that filter overlap and the associated number of distinct regions created in multi-dimensional space is much smaller than the worst case of $\mathbf{O(n^d)}$. For a filter set with 1734 filters the number of distinct overlapping regions in four-dimensional space was found to be 4316, as compared to the worst case which is approximately 10^{13}.

RFC performs independent, parallel searches on "chunks" of the packet header, where "chunks" may or may not correspond to packet header fields. The results of the "chunk" searches are combined in multiple phases. The result of each "chunk" lookup and aggregation step in RFC is an equivalence class identifier (classID) which represents the set of potentially matching filters for the packet. The number of classIDs in RFC depends upon the number of distinct sets of filters that can be matched by a packet. The number of classIDs in an aggregation step scales with the number of unique overlapping regions formed by filter projections.

RFC lookups in "chunk" and aggregation tables utilize indexing; the address for the table lookup is formed by concatenating the classIDs from the previous stages as shown in Figure 2-13. The resulting classID has fewer number of bits than the address, thus RFC performs a multi-stage reduction to a final classID that specifies the action to apply to the packet. The use of indexing simplifies the lookup process at each stage and allows RFC to provide high throughput. This simplicity and performance comes at the cost of memory inefficiency. The memory usage for less than 1000 filters ranged from a few hundred kilobytes to over one gigabyte of memory depending on the number of stages. The authors propose a hardware architecture using two 64MB SDRAMs and two 4Mb SRAMs that could perform 30 million lookups per second when operating at 125MHz. The index tables used for aggregation also require significant pre-computation in order to assign the proper classID for the combination of the classIDs of the previous phases. Such extensive pre-computation prohibits dynamic updates at high rates.

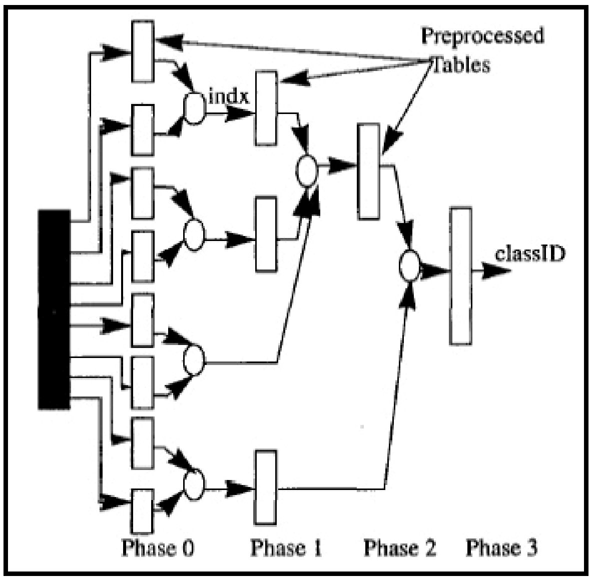


Figure 2-13 RFC aggregation scheme

Chapter 3

MAC Layer Classification

In this chapter we present our solution for MAC layer switching and classification in Ethernet networks. We developed a scheme suitable for hardware implementation that can facilitate the support of forwarding, switching, filtering, classification and QoS in Layer 2 (Data Link Layer). Our hardware solution aims at Ethernet switches or Bridges. We design a Hash Based Classification Engine (HBSE) that can support fast and storage efficient classification of many multi-gigabit links.

3.1 Ethernet Switching

Layer 2 (Data Link Layer) switching allows packets to be switched in the network based on their Media Access Control (MAC) address. The MAC sub-layer is part of the Data Link Layer and it is responsible to move the data packets from one Network Interface Card (NIC) to another across a channel. When a packet arrives at the switch, the switch checks the packet's destination MAC address and, if known, it sends the packet to the output port where the destination MAC is connected. The format of the Ethernet packets is shown in Figure 3-1.

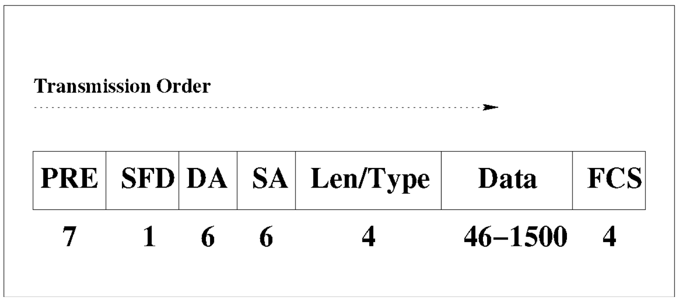


Figure 3-1 Ethernet Frame Format

The field lengths are in bytes and are the following:

- *PRE = Preamble*
- *SFD = Start-of-frame delimiter*
- *DA = Destination Address*
- *SA = Source Address*
- *Len/Type = Data Length of frame or frame Type*
- *FCS = Frame Check Sequence*

The three fundamental elements in Ethernet L2 switching are the MAC addresses, the ports of the switch and the Virtual LANs (VLANs). Since Ethernet switching is making a breakthrough in MAN and WAN networks, these elements are critical in mechanisms that provide QoS.

MAC Addresses

The MAC address is a 48-bit(6 bytes) value that uniquely identifies a NIC. The first 24-bits(3 bytes) of the address identify the vendor of the card and the last 24-bits identify the card itself. Every NIC has a MAC address that is hardwired and cannot be changed.

Ports

The ports are the physical interfaces where the NICs are connected to the switch. Each port can be identified by a number assigned by the manufacturer of the switch and provides all the communication from and to the attached NIC.

VLANs

VLAN tagging was introduced in IEEE 802.1q [4] and defines how an Ethernet frame is tagged with a VLAN ID. This tagging is a MAC option that provides some important capabilities not previously available to Ethernet network users and network managers. VLANs provide a mechanism to handle time-critical network traffic by setting transmission priorities to outgoing frames according to IEEE 802.1p [27]. Moreover VLANs allow network stations to be assigned to logical groups, and then communicate across multiple LANs as if they were on a single LAN. Bridges and switches filter destination addresses and forward VLAN frames only to ports that serve the specific VLAN traffic.

A VLAN-tagged frame is simply a basic MAC data frame that has a 4-byte extra header inserted between the SA and Length/Type fields as shown in Figure 3-2. The VLAN header consists of two fields:

- *A reserved 16-bit value to indicate that this is a VLAN frame(0x8100)*
- *A 16-bit Tag Control Info field:*
  - *The first 3-bits indicate the priority according to IEEE 802.1p (8 possible)*
  - *The next 1 bit is CFI (Canonical Format Indication)*
  - *The last 12-bits indicate the VLAN Identifier (4096 possible).*

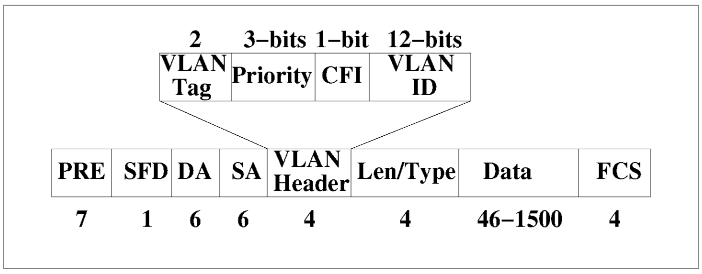


Figure 3-2 VLAN Ethernet Frame

Typically, there are two types of VLANs, port-based and MAC address-based. On port-based VLANs the logical grouping is done by assigning some specific ports to constitute a VLAN. When a data frame is received on a port, the switch or bridge determines the associated VLAN based on the port of the reception. Using the forwarding database information, the data frame is sent to the appropriate port(s). The other option is to specify VLANs using MAC addresses. MAC-based VLANs can be created by the MAC addresses of all devices on a network. VLANs of this type provide better device mobility and privacy for the users.

3.2 Hardware Based Classifiers

L2 switching, forwarding and filtering require the fields of each packet to be examined and the appropriate action to be performed. For example, given a packet’s destination MAC address, the packet should be forwarded to the appropriate output port. Therefore, the switches need to store some information and consult it for their decisions. The information about the MAC addresses, the VLANs and the Ports is stored in internal data structures and for each packet a search is conducted using the packet header fields.

Switches and bridges have integrated hardware solutions for the L2 classification task. They place the MAC address tables in internal or external memories and all operations access the tables to find the exact match. Today’s switches support at most 32K MAC addresses [28] and 4096 VLANs, hence the size of memories is relatively small.

The nature of L2 classification requires exact matches and many implementations use CAMs that provide single access matching. CAM solutions are simple but are expensive and power consuming. Trie based solutions have poor performance since the 48-bits of the MAC address are relatively long to be resolved with partial matches in subparts of the address. Moreover, trie based solutions may require several memory accesses and massive storage in pointers.

Another popular solution is hashing of the MAC address bits [29] and storing the data in SRAM based lookup tables. The 48-bits are hashed with a specific hashing function and an index for the lookup table is generated. Possible collisions due to hashing are usually resolved with linked lists of entries. Hashing 48-bits into a small, say 16-bit, value requires a good function that generates differentiated values by taking into account all the information bits. Many solutions use the CRC polynomials for hashing since they have been proved very efficient [30] or others use direct mapping by the least significant bits of the MAC address.

3.3 Hash Based Classification Engine

Our solution for L2 classification is based on hashing like many commercial products but we propose a hashing scheme that exactly matches certain requirements in terms of both memory accesses and storage. We propose a Hash Based Classification Engine (HBCE) with internal MAC Vendor replacement. HBCE is designed to support up to 64K MAC-address rules, 4096 VLANs and 1024 ports. Every rule in HBCE is uniquely identified by a number that can be called Flow ID, in our case we consider that 32K Flow IDs would be enough for a LAN.

The most essential part of our scheme is the MAC address table that will hold the associated information. The length of MAC addresses, namely 48-bits, is what makes this part the most critical in terms of both speed and storage. VLANs and ports are relatively small in size and can be directly mapped into tables, as it will be described in the next sections.

3.3.1 MAC Address Hashing

We developed a hashing function to map the MAC addresses into a table that will hold the Flow ID of the associated rule. MAC addresses are stored in a 64K table called MAC_TBL and the indexes to it are generated by the MAC address bits using our hashing function. The collisions due to hashing are handled by pointers to variable size blocks. Handling variable size blocks requires dynamic memory management implementation and is discussed in subsection 3.3.5. The number of entries of each variable size block is defined by the number of MACs that collide in a specific entry.

Indexes in MAC_TBL are generated by the use of the XOR function in all the 48-bits of the MAC address and the16-bit address is produced as follows:

$$MAC\_TBL_{index} = \{\ MAC[47:40]\ \mathbf{xor}\ MAC[31:24]\ \mathbf{xor}\ MAC[15:8]\ ,\ MAC[39:32]\ \mathbf{xor}\ MAC[23:16]\ \mathbf{xor}\ MAC[7:0]\ \}$$

To identify a certain MAC address in the block we also need to save some additional information so as to be able to distinguish those that collide. Fortunately, we don't need to save all 48-bits and we take advantage of the fact that the address has been produced by the actual MAC-address field. Therefore a MAC located in address A of MAC_TBL can be reproduced by the 16-bits of A and the last 32-bits (H_{val}) of the MAC address as follows:

$$MAC[47:40] = A[15:8]\ \mathbf{xor}\ H_{val}[31:24]\ \mathbf{xor}\ H_{val}[15:8]$$
$$MAC[39:32] = A[7:0]\ \mathbf{xor}\ H_{val}[23:16]\ \mathbf{xor}\ H_{val}[7:0]$$
$$MAC[31:0] = H_{val}(31:0)$$

The bits saved in H_{val} are unique for every possible MAC address located in address A and can be used to identify it. If we use CRC-16, like popular schemes, to produce 16-bit indexes then we should store the complete 48-bits of the MAC address because there is no inverse CRC function. Moreover, CRC polynomials don't have one-to-one correspondence between input and generated values. The speed and storage performance of our hashing function is discussed in section 3.4

3.3.2 MAC Vendor Replacement

The official IEEE OUI and Company ID assignments [31] has published all the assigned MAC vendor IDs of 24-bits and the associated company names. We collect them and observe that the 24-bit vendor address space of the MAC addresses is not fully occupied. The available list shows that fewer than 8000 vendors are active instead of the 2^{24}= 16777216 possible. Therefore we can replace the 24-bit vendor ID with a 13-bit internally assigned vendor ID; 13 bits are enough for the 8000 vendors.

The last 24-bits of the MAC address that uniquely identify a device of a vendor can remain unchanged. We decide to have internally replaced the vendor ID part of a MAC in order to reduce the storage requirements for each MAC address, at the cost obviously of the replacement operation. Consequently, every incoming MAC address need to be translated before the actual processing begins.

We can now consider that each MAC address handled by our system is 37-bits long. Naturally, this replacement means that we keep a small table with 8192 entries called VID_RPL that matches the existing 24-bit Vendor ID values with the internally assigned 13-bit Vendor ID values. This table can be easily constructed since all Vendor IDs are sequentially assigned by IEEE and a few ‘holes’ that exist in the address space can be handled by a 24-to-13 decoder. Despite this table is constant and can be kept in a ROM, we can use a method that learns the connected MAC addresses and assigns incrementally an internal ID. The first time an unknown MAC vendor ID appears in the system we can assign it with a new ID.

After this replacement we define a new hashing function on the 37-bits of the MAC address. Now, the 16-bit indexes in MAC_TBL are generated as follows:

$$\begin{aligned} MAC\_TBL_{index} = \{\ & MAC[31:24]\ \textbf{xor}\ MAC[15:8]\ , \\ & MAC[23:16]\ \textbf{xor}\ MAC[7:0]\ \} \end{aligned}$$

Notice that we don’t use the 6 MSB of the replaced Vendor ID in order to have a byte balanced hashing function. The new H_{val} is now 21-bits and is defined as follows:

$$H_{val} = \{\ MAC[36:24]\ ,\ \ MAC[7:0]\ \}$$

Now, a MAC located in address A of MAC_TBL can be reproduced by the 16-bits of the address and H_{val} as follows:

$$\begin{aligned} MAC[36:24] &= H_{val}[20:8] \\ MAC[23:16] &= A[15:8]\ \textbf{xor}\ H_{val}[7:0] \\ MAC[15:8] &= A[15:8]\ \textbf{xor}\ H_{val}[15:8] \\ MAC[7:0] &= H_{val}(7:0) \end{aligned}$$

3.3.3 MAC_TBL and Data Structure

MAC_TBL is a table with 64K entries and stores the Flow ID of each MAC address. Indexes in MAC_TBL are generated with hashing and therefore collisions may occur. To support resolving these collisions we define a complex data structure associated with each entry of the MAC_TBL. Each MAC address stored in an entry of the table needs 21-bits (H_{val}) to be fully identified (as described above) and along with this value we have to store the Flow ID which needs 15-bits. This information sums to

36-bits and should be stored in the memory. These 36-bits force the memory word to be at least 36-bits. If we use on-chip memories the word size is not a problem but in case of off-chip memories we have to find a commercial solution that matches our requirements. Fortunately, 36-bits is a popular word size of many SRAM vendors.

In the case where only one MAC address is saved in a table entry we can save the Flow ID in the 15 MSB of the word and H_{val} in the 21 LSB. However, a table entry might be empty which means that is not mapped to any MAC address, therefore we reserve the Flow ID number 0 for this purpose. The 15 MSB of the memory word should be set to 0 in empty entries. Moreover, a table entry may be mapped to many MAC addresses. In the latter case, where collisions occur, we have to store a pointer to the variable size block and the number of MACs that collide. The number of colliding MACs can also indicates the size of the block. For the cases of collisions we have reversed the Flow ID number 1 and store it in the 15 MSB of the word. The last 17-bits of the word are used to store the pointer to the block and the remaining 4-bits are used to keep the number of MACs mapped in this table entry. 4-bits are enough for the maximum number of collisions of our system as explained in subsection 3.4.1. The format of the memory words in each case is shown in Figure 3-3.

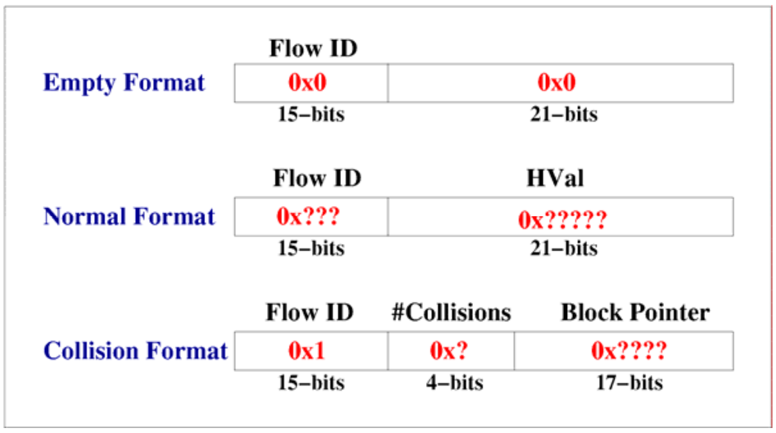


Figure 3-3 MAC_TBL entries format

The variable size blocks also use 36-bit memory words and the format of their entries is the same with the normal format of Figure 3-3. An example that shows the form of the data structure for some hypothetical MAC addresses is depicted in Figure 3-4.

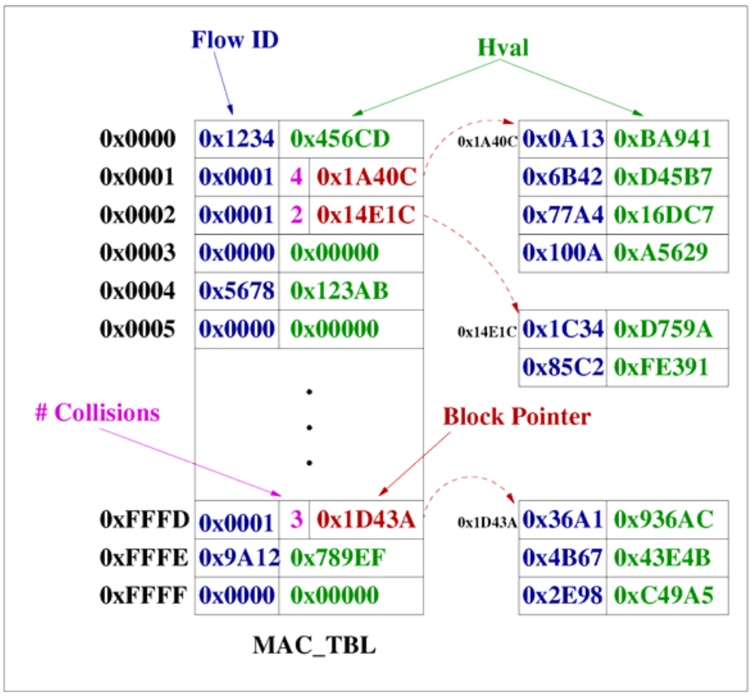


Figure 3-4 MAC_TBL Data structure example

Insert Operation

An insert operation in HBCE is a relatively simple task and needs a specific number of steps. Once a 48-bit MAC address is handled by our scheme we have first to replace the Vendor value with our internally assigned one by accessing VID_RPL. Then, the new MAC address of 37-bits is hashed to generate MAC_TBL_{index} and H_{val}. The generated index is used to access MAC_TBL and get the contents of the specific entry. The next step is to decode the Flow ID field and make the appropriate actions. Depending on the FlowID we may just insert the MAC address or allocate extra memory words to host the new MAC address. The complete specification of required steps is presented in subsection 4.3.2.

Lookup Operation

The lookup operation requires to examine a specific entry in MAC_TBL and follow the block pointer, if applicable, to locate the specific MAC address. Locating a MAC requires to check all the existing H_{val} fields. Once a 48-bit MAC address should be looked up by our scheme we have to replace the Vendor value with our internally assigned one by accessing VID_RPL. Then, the new MAC address of 37-bits is hashed to generate MAC_TBL_{index} and H_{val}. The generated index is used to access MAC_TBL and get the contents of the specific entry. The next step is to decode the Flow ID field and make the appropriate actions. Depending on the FlowID we may find the MAC address at once or we may follow pointers and sequentially search a block of colliding MAC addresses. The complete specification of required steps is presented in subsection 4.3.3.

Delete Operation

Delete operation requires to examine a specific entry in MAC_TBL and follow the block pointer, if applicable, to locate the MAC address and remove it. Locating a MAC requires to check all the existing H_{val} fields. Once a 48-bit MAC address should be deleted we have to first replace the Vendor value with our internally assigned one by accessing VID_RPL. Then, the new MAC address of 37-bits is hashed to generate MAC_TBL_{index} and H_{val}. The generated index is used to access MAC_TBL and get the contents of the specific entry. The next step is to decode the Flow ID field and make the appropriate actions. Depending on the FlowID we may delete the MAC address easily or we may follow pointers and remove it from a block of colliding MAC addresses. The complete specification of required steps is presented in subsection 4.3.4.

3.3.4 VLAN and Port Tables

Handling VLAN and Port fields is simple and requires storing the associated 15-bit Flow ID for each of the fields. VLAN is defined as a 12-bit identifier and can be directly mapped in a 4K table called VLAN_TBL. Similartly, the port field is defined as a 10-bit identifier and is directly mapped in a 1K table called PORT_TBL.

3.3.5 Dynamic Memory Management

Dynamic memory management in our system is needed to support the variable blocks described when collisions occur. This mechanism handles requests for memory allocation and deallocation of variable sizes. We have a pool of 64K adjacent memory words intended to be used for anti-collision purposes. An operation may require allocation of a certain number of memory words and our mechanism has to provide the address of the first of these words. The current dynamic memory management mechanism provides support for 2-word and 4-word blocks and is extensively described in subsection 4.4.

In case we need larger blocks, we cannot have adjacent memory words but we can link internally 2 or 4-word blocks by using the collision format discussed before. This decision does not significantly degrade the performance of our design because in both cases of adjacent and linked blocks we need to access all the memory words. The main disadvantage of this implementation is that if we need block sizes not multiples of 2 or 4 then we have to pay a small fragmentation overhead. Figure 3-5 depicts how blocks can be linked together and used in HBCE.

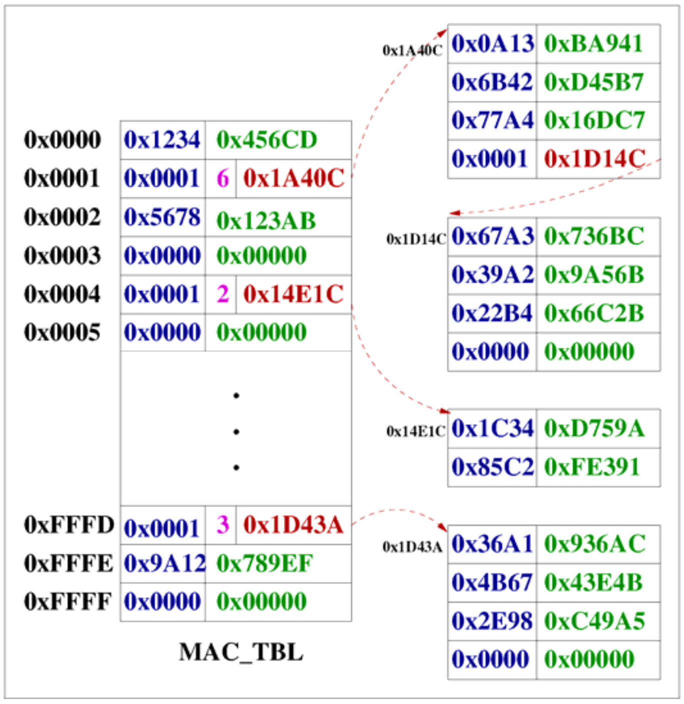


Figure 3-5 Data structure example with linked blocks

3.4 Simulation Results and Performance

In this subsection we discuss simulation results based on synthetic MAC address tables and present our results on storage and speed complexity. We calculate and analyze the performance of HBCE and compare it with the traditional CRC-16 and direct mapped solutions. HBCE storage and speed performance is based on certain assumptions for the underlying hardware and memory architecture.

3.4.1 Indexing MAC_TBL with a hashing function

Indexing MAC_TBL in HBCE is based on the hashing function proposed in subsection 3.3.2 which hashes the modified MAC address bits. We illustrate the performance of our function by using synthetic MAC address databases with existing MAC vendor IDs. We generated 32K, 48K and 64K MAC address databases with variable number of active vendor IDs, such as 256, 1500 and 4000, to test the behaviour of our function. For the generation of the databases we used real MAC Vendor IDs from the subset provided by OUI and appended random uniformly distributed 24-bit values that can represent the real network cards' serial numbers. We calculate the maximum and average number of collisions for our scheme and compare it with CRC-16 and direct mapping of the 16 LSBs. The simulation results are presented in Table 3-1.

Database Size (Active Vendors)	**Index Function**	**Maximum Collisions**	**Average Collisions**
32K (256)	CRC-16	5	1,495
	Direct Mapping	6	1,542
	HBCE	6	**1,490**
32K (1500)	CRC-16	6	1,476
	Direct Mapping	7	1,527
	HBCE	5	**1,483**
32K (4000)	CRC-16	5	1,482
	Direct Mapping	6	1,532
	HBCE	5	**1,481**
48K (256)	CRC-16	6	1,732
	Direct Mapping	8	1,822
	HBCE	6	**1,737**
48K (1500)	CRC-16	7	1,730
	Direct Mapping	8	1,821
	HBCE	7	**1,732**
48K (4000)	CRC-16	6	1,728
	Direct Mapping	8	1,818
	HBCE	7	**1,735**
64K (256)	CRC-16	8	2,631
	Direct Mapping	9	2,792
	HBCE	7	**2,642**
64K (1500)	CRC-16	7	2,630
	Direct Mapping	8	2,765
	HBCE	7	**2,637**
64K (4000)	CRC-16	7	2,618
	Direct Mapping	9	2,771
	HBCE	8	**2,642**

Table 3-1 Indexing simulation results

The results show that the HBCE seems a good hash function that approaches CRC-16 performance and is better that direct mapping. The XOR function used by both CRC-16 and HBCE provides better collisions results because in generates more uniformly distributed indexes. The advantage of HBCE is that it requires only a small portion from the original MAC address to be stored instead of the total 48-bits required by CRC. It is also much simpler and less expensive to implement the HBCE hash function in hardware. The results also show that when the number of MAC addresses stored in MAC_TBL grows to the limits of the table, namely 64K, the average number of collisions increases but fortunately it remains in tolerable levels. The number of active vendors in the dataset seems that it does not influence the performance. Moreover, the maximum number of collisions appeared during simulations allows us to assume that 4-bits are enough for the #Collisions field which currently supports up to 15 collisions.

Additionally to the synthetic MAC databases we use real MAC addresses from ICS-FORTHs network and Computer Laboratory of University of Cambridge[3]. We concatenate these MAC addresses to create a real database and provide the simulation results in Table 3-2.

Database Size (Active Vendors)	**Index Function**	**Maximum Collisions**	**Average Collisions**
1611 (195)	CRC-16	2	1,023
	Direct Mapping	2	1,031
	HBCE	2	**1,019**

Table 3-2 Real database simulation results

This small sample of real MAC addresses still shows that our hashing function is performing very well and can be efficiently used on a real system such as a central L2 switch of a big institution.

3.4.2 Storage Requirements

We calculate the total storage requirements of HBCE for the synthetic databases based on the collisions produced in each case and assume that all the rules are stored in 36-bit wide words. The collisions are handled by the dynamic memory management system described in subsection 3.3.5 and thus apart from the static tables used we have to calculate the number of 2-word and 4-word blocks required. The size of the static tables is demonstrated in Table 3-3.

Table	**Entries**	**Total Bytes**
MAC_TBL	65536	294912
VLAN_TBL	4096	18432
PORT_TBL	1024	4608
VID_RPL	8192	36864
Total	**78848**	**354816 (346 Kb)**

Table 3-3 HBCE static tables memory

[3] We kindly thank the network administrators for providing us with this valuable information.

In Table 3-4 we present the final storage requirements of HBCE for each database, and include in our calculations the collision blocks linked in MAC_TBL. We also present the storage requirements if CRC-16 was the hashing function for the same databases. Note that in the CRC case we need two memory words for each MAC address because we need to keep the 37-bit internal MAC address and the corresponding 15-bit FlowID.

Database Size (Active Vendors)	**Static Tables (Kbytes)**	**Collision Blocks (Kbytes)**	**HBCE Total (Kbytes)**	**CRC Total (Kbytes)**
32K (256)	346	58	404	634
32K (1500)	346	57	403	634
32K (4000)	346	58	**404**	**634**
48K (256)	346	120	466	788
48K (1500)	346	120	466	787
48K (4000)	346	120	**466**	**787**
64K (256)	346	194	540	947
64K (1500)	346	194	540	948
64K (4000)	346	195	**541**	**946**
1611(195)	346	0,1	**346,1**	**360**

Table 3-4 HBCE final storage requirements

We can see that half megabyte is enough for HBCE to store 64K MAC addresses and support QoS. Moreover, we have 36% - 42% better storage requirements than the equivalent CRC-16 solution. Note also, that although we have assigned 64K adjacent memory words (288 Kbytes) for collision resolving only 70% of this space is actually used which means that it is possible for our scheme to support more than 64K MAC addresses. The cost of supporting even more MAC addresses would naturally be an increase in the average number of collisions.

3.4.3 Lookup performance

The lookup performance of HBCE is based on the total number of memory accesses required to find a match in the tables. This a performance metric very frequently used in such schemes. VLAN_TBL and PORT_TBL are direct mapped and therefore the Flow ID can be found with a single access in the appropriate table. MAC_TBL is the most critical table for the performance of HBCE since collisions may occur and we have to lookup sequentially all the colliding MAC addresses. For every incoming MAC address we have first to replace the original vendor ID with our internally assigned one. Therefore we need a single memory access in VID_RPL, then the MAC_TBL index is generated based on the modified MAC address. The number of accesses required to resolve a MAC address also depends on the number of collisions that have occurred. According to Table 3-1 the worst case number of memory accesses for 64K MAC addresses is 8 but the average number is 2,64 which is fairly smaller. In Table 3-5 we present the summary of worst and average case memory accesses for each case.

Active MAC Addresses	**Average Case**	**Worst Case**
32K	2,49	7
48K	2,73	8
64K	3,64	9

Table 3-5 HBCE total number of memory accesses

Supported Link Speeds

According to our lookup performance we can calculate the efficiency of HBCE as a classification engine in a high speed L2 switch. To calculate the network performance we have to assume a certain speed for the memory we use and a pipelined hardware implementation that can provide one memory access per cycle. The results we present assume 2 possible memory configurations:

- 200Mhz off-chip synchronous SRAM
- 400Mhz on-chip synchronous SRAM

We also assume that the worst case scenario for HBCE is when L2 transports minimum sized Ethernet packets (64 bytes). The summary of the supported link speeds are presented in Table 3-6.

Active MAC Addresses	**Off-chip SRAM 200Mhz**		**On-Chip SRAM 400Mhz**	
	Average (Gbps)	Worst Case (Gbps)	Average (Gbps)	Worst Case (Gbps)
32K	**41,2**	14,6	**82,2**	29,3
48K	**37,5**	12,8	**75,0**	25,6
64K	**28,13**	11,4	**56,2**	22,8

Table 3-6 HBCE network performance

The network performance presented in Table 3-6 allows HBCE to be used in a high speed switch that can support many high speed ports. The average case of a 64K MAC database demonstrates that our scheme can be used in a switch/concentrator consisting of 36 x 1Gbit ports and 2 x 10Gbit port or other combinations such as 16 x 1Gbit ports and 4 x 10Gbit ports.

Chapter 4

Hardware Implementation of HBCE

In this chapter we present a reference hardware implementation of the HBCE MAC layer classification scheme that was described in Chapter 3. We provide a detailed description of all the internal blocks of the system and the hardware resources utilized. We also present the speed and silicon area estimations of the final design. We decide to implement the final design in an FPGA platform to prove the feasibility and scalability of the architecture, even when limited hardware resources are available. The FPGA platform we use is a Xilinx Virtex II Pro [32] with an external Cypress NoBL (ZBT) SSRAM [33].

4.1 HBCE Organization

HBCE involves many internal blocks to implement the required functionalities. Figure 4-1 illustrates the internal organization of HBCE and the external interfaces. The central operation of the system is handled by a Main Control Block (HBCE_MCB) which receives commands from the OPB_INF block. OPB_INF is an implementation of Xilinx OPB Bus slave interface [34]. Upon a reception of a command HBCE_MCB instructs the MAC_VID block to make the vendor ID replacement and then provides the modified MAC address to MAC_HSH in order to perform hashing in the data. When the hashed values are ready then HBCE_MCB performs the appropriate actions so as to insert, lookup or delete a MAC address or a VLAN or a Port in the data structure. HBCE_MCB interfaces with the memory through the memory handler (MEM_HDLR) and the memory controller (MEM_CTRL). The MEM_HDLR implements the dynamic memory management scheme described in section 4.4 by employing several free-lists and the MEM_CTRL is the actual low level memory interface. When the final FlowID is resolved then it is returned through the OPB_INF block to the instructor of the initial command.

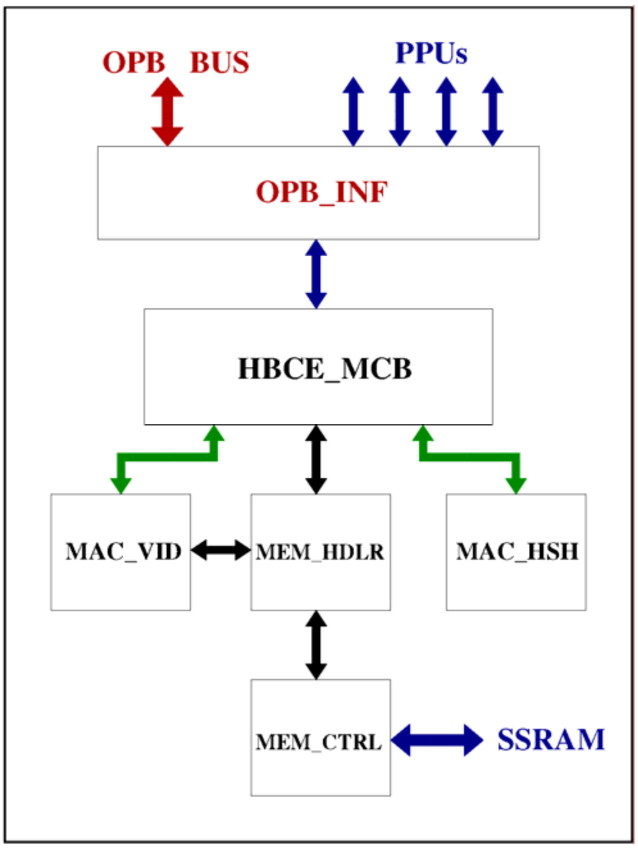


Figure 4-1 HBCE Internal organization and block diagram

4.2 OPB_INF

OPB_INF has an FSM to implement the OPB Bus slave interface timings in order to have interconnection with the peripheral Bus that is widely used in Xilinx FPGA platforms. This interface has a 32-bit address bus and a 32-bit data bus and supports read and write operations on specific addresses that correspond to actual block registers. OPB_INF receives read and write commands to internal registers from four parallel processing units (PPUs) and provides the result to the corresponding instructor unit through BRAM interfaces. The signals of the interface and their descriptions are shown in Table 4-1.

Signal	Length	In/Out	Description
i_opb_select	1	I	Initiates the transaction.
i_opb_rnw	1	I	Indicates read or write.
i_opb_be	4	I	Byte enable for the data.
i_opb_seqaddr	1	I	Sequential address transactions.
i_opb_abus	32	I	Incoming Address
i_opb_dbus	32	I	Incoming Data
o_opb_xferack	1	O	Transaction acknowledge.

o_opb_errack	1	O	Error acknowledge.
o_opb_toutsup	1	O	Timeout suppress.
o_opb_retry	1	O	Request retry.
o_opb_dbus	32	O	Outgoing Data
o_hbce_req	1	O	Request for HBCE operation
o_hbce_opcode	3	O	Opcode of operation
o_hbce_addr	48	O	MAC address data
o_hbce_flow_id	15	O	Flow ID data
o_hbce_vlan	12	O	VLAN data
o_hbce_port	10	O	Port data
o_hbce_fld_bmp	3	O	Bitmap to indicate the valid data

Table 4-1 OPB_INF signals with the bus and HBCE

HBCE needs several data to start working on a MAC address, a VLAN or a Port and all of them need to pass over the OPB bus. For this purpose, we define some control registers that each instructor unit should fill before it starts an operation. The control registers defined are the following:

- **ConfReg0** : It contains the PPU number that instructs the commands and the valid parts of the command. The fields of the register are:

  Address: ADDRHI & 0x10000

31:11	10:8	7:2	1:0
Reserved	**Rule Bitmap**	Reserved	**PPU number**

  PPU number: Is a 2-bit field that indicates which of the 4 PPUs instructed the command.

  Rule Bitmap : Is a 3-bit that indicates which parts of the incoming rule are valid. Bit(10) indicates that MAC is valid, Bit(9) indicates that VLAN is valid and Bit(8) indicates that port is valid.

- **ConfReg1** : It contains the 32 MSB of the incoming 48-bit MAC Address. The fields of the register are:

  Address: ADDRHI & 0x10001

31:0
MAC Address [47:16]

- **ConfReg2** : It contains the 16 LSB of the incoming 48-bit MAC Address. The fields of the register are:

  Address: ADDRHI & 0x10002

31:16	15:0
Reserved	**MAC Address [15:0]**

- **ConfReg3 :** It contains the Flow ID of the rule to be inserted. The fields of the register are:

  Address: ADDRHI & 0x10003

31:15	14:0
Reserved	**Flow ID**

- **ConfReg4 :** It contains the values of the incoming VLAN and Port. The fields of the register are:

  Address: ADDRHI & 0x10004

31:28	27:16	15:10	9:0
Reserved	**VLAN**	Reserved	**Port**

Access to these registers is achieved with normal OPB reads or writes to the address of each register. Using these registers we also define the commands for HBCE that can be given through the OPB Bus. The commands are the following:

- **InsertKey:** This command aims to be used for rule insertion in the database and results in insert operation requests to HBCE. Before this command is initiated the appropriate configuration registers (ConfReg0-4) should be written with the desired values.

  Address: ADDRHI & 0xA0000

  OPB Command: Read

- **SearchKey:** This command should be used to lookup a given set of MAC, VLAN, PORT values in the data structure and results in lookup operation requests to HBCE. Before this command is initiated the appropriate configuration registers (ConfReg0-4) should be written with the desired values.

  Address: ADDRHI & 0xC0000

  OPB Command: Read

- **DelKey:** This command should be used to delete a rule given the MAC address, or VLAN or Port of the rule and results in delete operation requests to HBCE. Before this command is initiated the appropriate configuration registers (ConfReg0-4) should be written with the desired values.

  Address: ADDRHI & 0xC0000

  OPB Command: Write

- **WrVendor:** This command aims to be used on the initialization of the block to fill the MAC vendor replacement tables with the appropriate values. The range of valid addresses is: 0xA0000 - 0xA0EC8

  Address: ADDRHI & 0xA0000

  OPB Command: Write

4.3 HBCE_MCB

HBCE_MCB has several internal blocks that handle the operations of the HBCE scheme as described in Chapter 3. The internal organization of HBCE_MCB is depicted in Figure 4-2. HBCE_MCB interfaces with OPB_INF block to receive commands and notifies it when it completes an operation. It also communicates with MAC_VID to receive the internally modified MAC address and with MAC_HSH to get the hashed values. Moreover the required memory communication is done over the MEM_HDLR block where requests for read, write, memory allocation and deallocation are given.

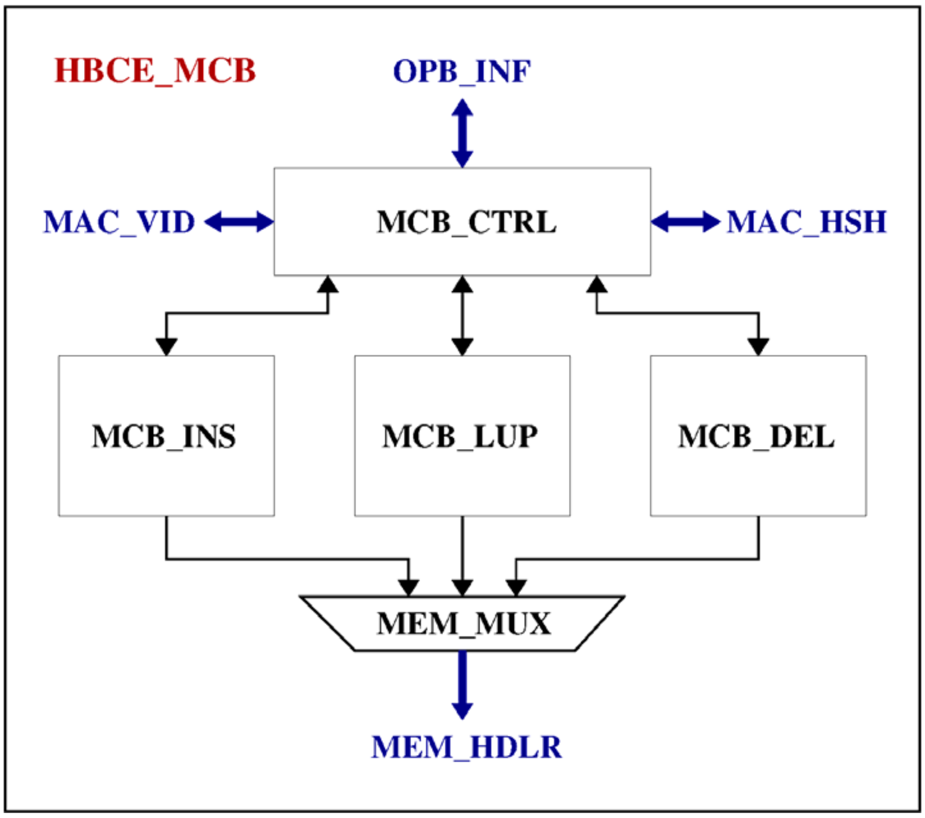


Figure 4-2 HBCE_MCB internal organization

Memory Organization and Tables

The current HBCE implementation is based on sequential accesses to MAC_TBL and follows the pointers to the dynamically allocated nodes. Moreover in the memory we have stored the VLAN table (VLAN_TBL), the Port table (PORT_TBL) and the vendor assignment table (VID_RPL). All these tables and the free memory addresses are stored in the same SSRAM. The memory word we use is 36-bits and we use at most 128K words which have been found enough during the simulations of subsection 3.4.2. The organization of the tables in the memory and the pool of free memory words for dynamic memory management is shown in Figure 4-3. The first 64K words are used for MAC_TBL, the next 8K words are for VID_RPL, the next 4K words are for VLAN_TBL and the next 1K words are for PORT_TBL. The remaining 52224 memory words are used by the memory handler (MEM_HDLR) to provide dynamic allocation and deallocation of memory blocks.

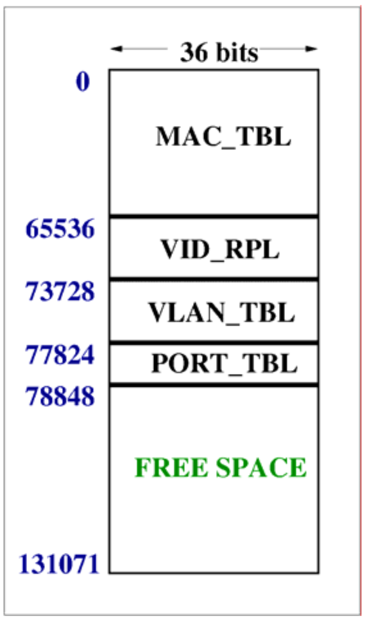


Figure 4-3 HBCE Memory Organization

4.3.1 MCB_CTRL

MCB_CTRL is responsible to manage the block's operations and involves an FSM to handle the requests for the insert, lookup and delete defined by the following opcodes:

- *2'b00 : Lookup*
- *2'b01 : Insert*
- *2'b10 : Delete*
- *2'b11 : Reserved*

For each operation there is a sub-block responsible to complete it. MCB_INS is responsible for the inserts, MCB_LUP for the lookups and MCB_DEL for the deletes. Upon a reception of a command MCB_CTRL generates a request to MAC_VID block in order the MAC vendor ID to be replaced and then instructs MAC_HSH to generate the proper hash values. Then it orders one of the MCB_INS, MCB_LUP and MCB_DEL blocks to start its operation and sets the MEM_MUX to output the appropriate block's requests to the memory handler.

4.3.2 MCB_INS

MCB_INS sub-block handles all the insertions in the appropriate table depending on whether a MAC address, a VLAN or a Port rule is to be inserted. VLAN and port insertions require a single memory access and are trivial, however inserting a MAC address is the most complex operation and has an FSM to handle the possible cases. After the vendor replacement and the hashing we access MAC_TBL in the address indicated by $\mathbf{T_{index}}$ and decode the FlowID field:

- *If **FlowID** field is 0 we write the given Flow ID and the generated $\mathbf{H_{val}}$.*
- *If **FlowID** field has value 1 we proceed to the following steps:*
  - *allocate a memory block of size **#Collisions** + 1 ,*
  - *we copy the contents of the old block specified by the block pointer to the newly allocated block,*
  - *add the new entry in the last word of the block by writing the given **FlowID** and the generated $\mathbf{H_{val}}$,*
  - *deallocate the old block,*
  - *update the MAC_TBL entry with the new **#Collisions** and the new block pointer.*
- *If **FlowID** field has value other than 0 or 1 we do the following:*
  - *allocate a memory block of size 2*
  - *write the data read from MAC_TBL to the first word of the block,*
  - *add the new entry in the second word of the block by writing the given **FlowID** and the generated $\mathbf{H_{val}}$,*
  - *update the MAC_TBL entry by writing the **FlowID** field with 1, the **#Collisions** field with 2 and the **Block Pointer** field with the address of the allocated block.*

4.3.3 MCB_LUP

MCB_LUP sub-block handles the lookups in the appropriate table depending on whether a MAC address, a VLAN or a Port rule is to be searched. VLAN and port lookups require a single memory access and are trivial, however looking for a MAC address is a more complex operation and has an FSM to handle the possible cases. After the vendor replacement and the hashing we access MAC_TBL in the address indicated by $\mathbf{T}_{\mathbf{index}}$ and decode the FlowID field:

- *If the **FlowID** field is 0 then we have not a match.*
- *If the **FlowID** field is has a value 1 we follow the **Block Pointer** and read as many words as the **#Collisions** field says. During each word access we compare the H_{val} field of the word with the generated one.*
  - *If we find a match in one of the words then we return associated the **FlowID** field,*
  - *Otherwise, when the words finish and we don't have found a match.*
- *If the **FlowID** field has value other than 0 or 1 then we compare the H_{val} field of the entry with the generated one.*
  - *If the values match we return the **FlowID** field of the entry,*
  - *Otherwise we don't have a match.*

4.3.4 MCB_DEL

MCB_DEL sub-block handles the deletions in the appropriate table depending on whether a MAC address, a VLAN or a Port rule is to be deleted. VLAN and port deletes require a single memory access and are trivial, however deleting a MAC address is more a complex operation and has an FSM to handle the possible cases. After the vendor replacement and the hashing, we access MAC_TBL in the address indicated by $\mathbf{T}_{\mathbf{index}}$ and decode the FlowID field:

- *If the **FlowID** field has value 0 then delete fails.*
- *If the **FlowID** field has a value 1 we check the **#Collisions** Field*
  - *If it is 2 then we find which word matches, we move the other word to the specific TBL_MAC entry and deallocate the block. If none of the words match then delete fails.*
  - *If it is not 2 we follow the Block Pointer and read as many words as the **#Collisions** field says. During each word access we compare the H_{val} field of the word with the generated one.*
    - *If we find a match in one of the words then we substitute this word with the last word of the block and remove the last word.*
    - *Otherwise, the words finish and delete fails.*
- *If the **FlowID** field has value other than 0 or 1 then we compare the H_{val} field of the entry with the generated one.*
    - *If the values match we substitute it with a word of empty format.*
    - *Otherwise delete fails.*

4.3.5 MAC_VID

This sub-block receives the vendor ID value in 24-bits and finds the corresponding internally assigned ones in 13-bits. It holds a small number of special cases into an internal lookup table and consults the VID_RPL table if the vendor ID does not belong to the special cases. VID_RPL is located inside the external SSRAM and the final internal ID is found in a defined table offset and in the sub-offset specified by the last 8-bits of the MAC vendor.

4.3.6 MAC_HSH

This sub-block receives the modified vendor ID value from MAC_VID and the last 24-bits of the original MAC address and calculates $\mathbf{T_{index}}$ and $\mathbf{H_{val}}$ as defined in subsection 3.3.2. This sub-block is of minor complexity since it has only a few XOR gates and has single cycle latency. It can be easily modified to implement a new hashing scheme of variable latency without affecting the rest of the system.

4.4 MEM_HDLR

The MEM_HDLR sub-block provides the dynamic memory management in our system and supports variable size blocks. MEM_HDLR is the intermediate layer between the blocks and the memory controller MEM_CTRL to support requests for allocation and deallocation of variable size blocks. Requests for reads or writes in the memory are immediately forwarded to the memory controller MEM_CTRL.

We have a pool of 64K adjacent memory words intended for dynamic operations. To support this management we use a head pointer to the pool of these addresses, a tail pointer to the last address of this pool and a current pointer to keep the state of the already used words. During allocation from the pool we increment the current pointer. The deallocated blocks are placed into free-lists where each free-list holds all the deallocated blocks of a certain size. For every free-list we keep a head, tail pointer and a counter to keep the number of linked blocks. Linking between multiple blocks is implemented by writing the address of the next block inside the data of the previous block. We decide not to support unlimited free-lists for blocks of different sizes but limit allocation and deallocation into blocks of 2 and 4 words. During requests for allocation of a specific size block we first check if we have available blocks in the corresponding free-list and if not then we allocate from the pool. Upon deallocation, we add the deallocated block in the tail of the corresponding free-list and increment the appropriate counter. Figure 4-4 illustrates the mechanism of memory pool and the free-lists.

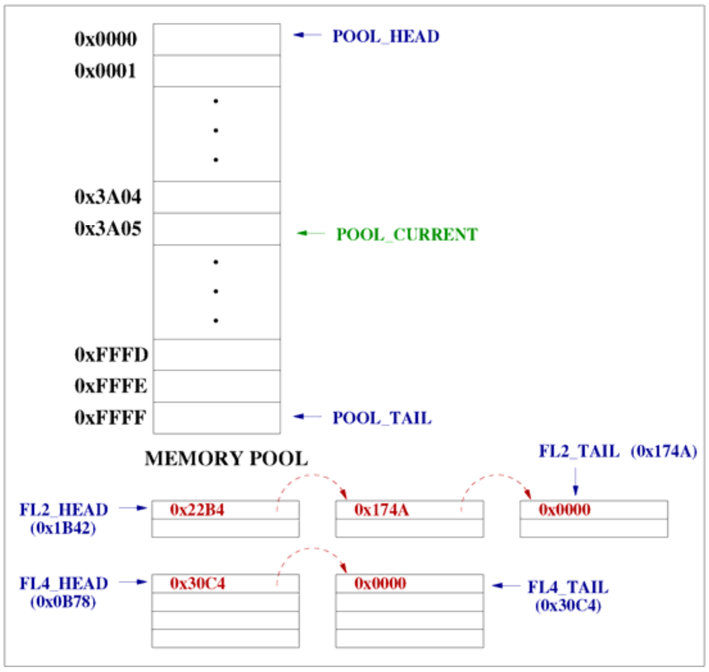


Figure 4-4 Snapshot of dynamic memory management mechanism

4.5 MEM_CTRL

The memory controller has an FSM to implement the timing described by the Cypress ZBT SRAM datasheet [33] and provides an interface to read and write the memory. Writes are performed in a single cycle but reads have two cycles latency since the data outputted from the memory need to be registered in order to be safely returned. Figure 4-5 illustrates the view of the system and highlights the read path. The memory inserts a single cycle latency and the register another cycle. The input data are registered because they come from an external memory interface and it is not safe to use this input in slow logic or long routed paths. Moreover, the valid read data are given along with an acknowledge signal that exists in the controller interface.

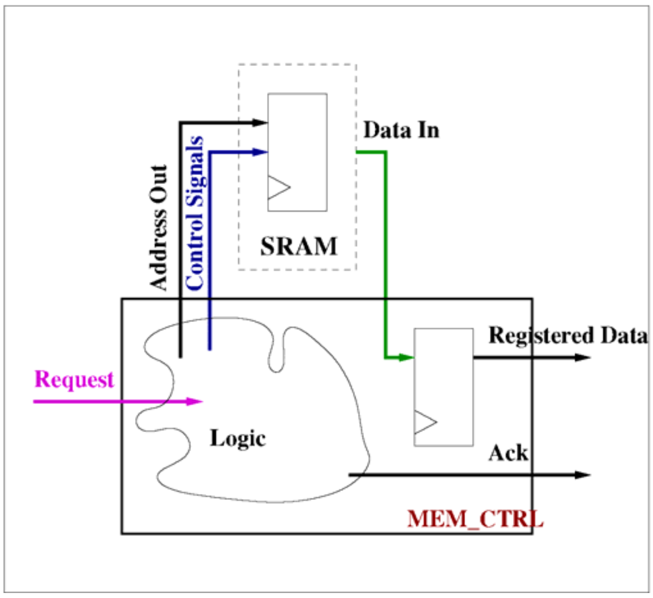


Figure 4-5 Overview of MEM_CTRL

4.6 Implementation Analysis

In this subsection we provide an analysis of the block latencies and an estimation of the implementation cost for the reference design.

4.6.1 Latency Analysis

We calculate the minimum and the maximum number of clock cycles required by each block to complete its operation. Many of the blocks have variable latencies which depend on the access patterns and the data stored in the data structures. Moreover, the blocks that access the external SSRAM for the stored data structures have to also suffer from the latency of our memory controller. In Table 4-2 we present the latency per block of HBCE.

Block Name	**Min Latency (clock cycles)**	**Max Latency (clock cycles)**
OPB_INF	1	3
MCB_CTRL	1	-
MCB_INS	3	13
MCB_LUP	2	17
MCB_DEL	3	15
MAC_VID	1	3
MAC_HSH	1	1
MEM_HDLR	0	3
MEM_CTRL	1	2

Table 4-2 B2PC Blocks Latencies

The fact that the memory controller has latency 2 cycles for a read operation in the external SSRAM significantly affects the performance of the blocks that perform sequential accesses to the memory. Insert, lookup and delete operations are high depending on the read data to decide the address of the next memory access and thus the 2 cycle latency of the memory controller is continuously introduced. Additionally, for the blocks MCB_INS, MCB_LUP and MCB_DEL we consider that we may have and support at most 15 collisions and this bounds the maximum latency. According to the number of memory accesses we calculated in subsection 3.4.3 we need for a lookup 7,3 clock cycles on average.

4.6.2 Hardware Cost Analysis

We have used VHDL to describe the design and the results presented are the reports from the synthesis tools. We have synthesized the design using the Synopsys Design Compiler [35] which is the most widely used synthesis tool. We have used UMCs 0.13μm technology library to estimate the area and the frequency of the design. Moreover, we used the XilinX ISE tool to implement and port the design in the FPGA.

The synthesis tool for the ASIC flow indicates that the maximum working frequency of our design is 500Mhz.Using the synthesis tool we calculated the number of flip-flops contained in our design and we present them per high level block in Table 4-3 and also calculate the total.

Block	**Block Description**	**Number of Flip-Flops**
HBCE_MCB	Main Control of HBCE	592
MAC_VID	Vendor ID replacement	28
MAC_HSH	Hashing the MAC address	38
MEM_HDLR	Memory Handler	184
MEM_CTRL	Memory Controller	43
OPB_INF	OPB Bus Interface	296
Total		**1181**

Table 4-3 Flip-Flop count per block

The area of the total design and the equivalent gate count is presented in Table 4-4. The equivalent gate count is calculated by considering how many 2-input NANDs can be accommodated in this area.

Block	Area (mm^2)	Equivalent NAND Gates
Combinatorial	0,044	8482
Non-Combinatorial	0,054	10362
Total	**0,098**	**18844**

Table 4-4 Area estimations of HBCE

ISE tool of the Xilinx FPGA flow shows that the maximum working frequency of our design is 100 Mhz. The tool reports the occupied resources after a full back-end FPGA flow while occupying optimizations to remove redundant logic or replicate logic to improve speed. The final results are shown in Table 4-5.

Resource	Resource count
Used 4 input LUTs	2371
Slice Flip Flops	1060

Table 4-5 FPGA resource allocation

4.6.3 HBCE Hardware Performance

Considering that we have a 100MHz clock, the external memory works on the same frequency and the average lookup time is 7,3 clock cycles then, the FPGA prototype design of HBCE supports at worst case 7 Gbps.

Chapter 5

Bitmap Oriented Strides

In this chapter we present Bitmap Oriented Strides (BOS), our algorithm for Longest Prefix Matching (LPM). We developed an algorithm for LPM, suitable for pipelined hardware implementation which can be used in an environment that prefix lookups are essential. Applications of this kind are routing lookups, forwarding and packet classification. BOS is a multi-bit trie algorithm that uses bitmaps across strides and involves complex data structures and certain optimization techniques so as to support fast and storage efficient IPv4 prefix lookups. The design of BOS is based on observations and simulations upon real IPv4 routing prefixes. We also strive after a scheme that can support incremental updates in modest time and storage.

5.1 Analysis and Description of BOS Algorithm

The BOS algorithm design and analysis is based on some very important observations that were made after extended literature study and routing tables' analysis.

5.1.1 Routing Table Analysis

We collected several routing tables from backbone routers of the Internet that are available in IPMA [36] and analyze them in statistical manner. We counted lengths of the prefixes included in those routing tables and observe the distribution shown in Figure 5-1. Table 5-1 shows values collected from the tables' analysis. It is clearly shown that more than 99% of the prefixes have lengths in the interval between 16 and 24 and more that half of the total prefixes have length equal to 24. This distribution has been found to be constant over time and stable between routing tables of various sizes, hence we can use it as a guide for our algorithm.

Prefix Length	**AADS 2000/10**		**MAE-EAST 2000/01**		**PAIX 2000/10**	
	Prefix Count	%	Prefix Count	%	Prefix Count	%
0	0	0,00	0	0,00	0	0,00
1	0	0,00	0	0,00	0	0,00
2	0	0,00	0	0,00	0	0,00
3	0	0,00	0	0,00	0	0,00
4	0	0,00	0	0,00	0	0,00
5	0	0,00	0	0,00	0	0,00
6	0	0,00	0	0,00	0	0,00
7	0	0,00	0	0,00	0	0,00
8	14	0,04	28	0,05	25	0,03
9	2	0,01	4	0,01	4	0,00
10	1	0,00	5	0,01	5	0,01
11	0	0,00	9	0,01	9	0,01
12	5	0,01	28	0,05	29	0,03
13	13	0,03	36	0,06	60	0,07
14	49	0,12	130	0,22	174	0,19
15	95	0,24	224	0,37	289	0,32
16	2726	6,84	5610	9,35	6693	7,33
17	450	1,13	625	1,04	933	1,02
18	849	2,13	1284	2,14	1889	2,07
19	2833	7,10	4195	6,99	6023	6,60
20	1670	4,19	2321	3,87	3875	4,25
21	1553	3,89	2671	4,45	3932	4,31
22	2329	5,84	3757	6,26	5900	6,46
23	2984	7,48	5175	8,62	7883	8,64
24	**19846**	**49,77**	**33691**	**56,15**	**52679**	**57,71**
25	428	1,07	28	0,05	258	0,28
26	555	1,39	54	0,09	323	0,35
27	421	1,06	9	0,01	190	0,21
28	625	1,57	13	0,02	54	0,06
29	307	0,77	12	0,02	26	0,03
30	761	1,91	84	0,14	18	0,02
31	25	0,06	0	0,00	0	0,00
32	1335	3,35	11	0,02	7	0,01
Total	**39876**		**60004**		**91278**	

Table 5-1 Routing Table Data

It is obvious that we wanted to design an algorithm that takes into consideration the form of routing tables and exploit these observations. Since most of the prefixes are in the interval between 16 and 24 we tried to optimize the data structure so as to handle these prefixes as fast and as efficient as possible. Since we would like to “make the common case fast” we concentrated our efforts on the lookups contained in this particular interval.

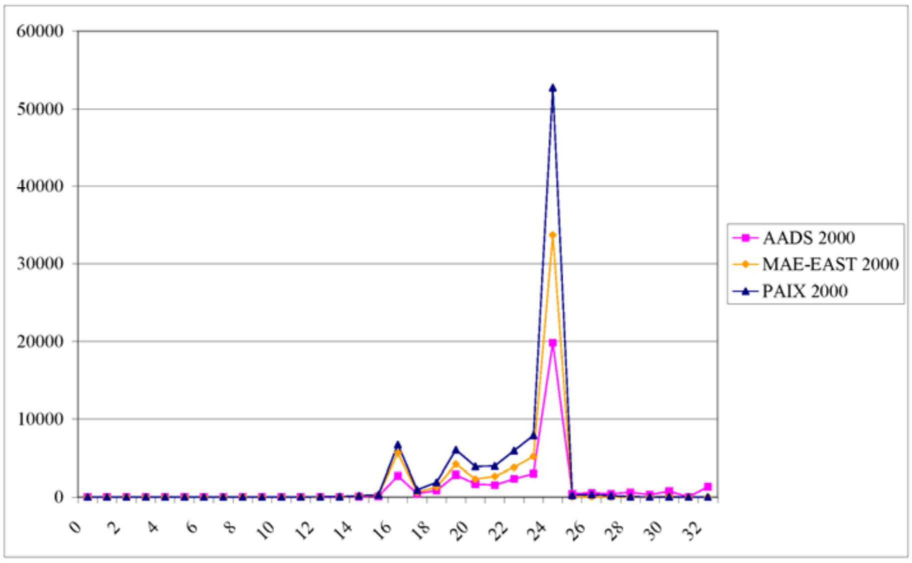


Figure 5-1 Routing Table Distribution

5.1.2 Trie-Based Solutions

Many algorithmic solutions on the LPM problem make extended use of tries and traverse tree data structures to find the matching prefix. Unibit tries check one bit at a time and follow the nodes until no matching bit is found. Schemes of this type have a worst case lookup of 32 memory accesses for IPv4 (since the IPv4 address fields are 32 bits long) and spend also lot of memory to save the pointers for the next nodes. On the other hand, multi-bit tries traverse several bits at a time and this provides faster searches. For example if we check 4 bits at a time (4-bit strides) then the worst case is 8 memory accesses. In these tries, problems arise when the prefixes are not multiples of the stride length. Solution to this problem is prefix expansion as described in [11]. CPE generates many prefixes and leads to great memory waste (especially when the stride length grows) and to non deterministic update times.

Other, LPM schemes from literature like Lulea [13] tried to solve the memory waste of CPE by using compressed bitmaps to represent strides. They use strides of 16,8 and 8-bits consecutively to represent the 32-bit IPv4 address space. The first 16bits are used as an index to a 64K table and the next 8-bit strides are represented by their own bitmap algorithm where each stride requires 32 bytes nodes even if only 1 prefix exists in the 256 space. A lookup is performed at worst case with 9 memory accesses but incremental updates to this scheme are inherently slow. Lulea is the most storage efficient scheme presented in literature so far.

5.1.3 Memory technologies and wire speed

The routing lookup operation is very important in the latest switching/routing equipments and networks. The need for wire-speed means that a routing decision should be made in time less that 40ns (worst case in 10Gbps) and of course this cannot be done efficiently in software. Moreover the routing tables' sizes require the use of big (dense) and fast memories that can provide high bandwidth.

Today's memory technology provides fast SRAMs and high-throughput and large DRAMs but a designer must make the right decision given the requirements of his system. DRAMs can be big (256Mbytes) and relatively cheap but their access time is poor (~60ns) when is to be used in routing lookup functions. SRAMs on the other side are a lot faster with access times smaller than 5ns but large capacity SRAMs cost a lot. Additionally, SDRAMs are highly suggested for sequential accesses. Under these conditions they provide high bandwidth but in the case of trie-based algorithms the use of pointers to random addresses makes this choice not practical. Contrarily SRAMs give the flexibility of fast random accesses and constant bandwidth.

5.1.4 BOS approach

In our approach to find a solution to the LPM problem we will use all the above observations to extract an algorithm that will have the following properties:

1. Easily implementable in hardware
2. Moderate algorithmic complexity
3. Fast lookups times for common case
4. Decent storage requirements and affordable for low budget designs
5. Deterministic and bounded incremental update times

In order to cope with the above requirements we ended up with BOS algorithm which:

- Uses strides and multi-bit trie nodes in order to traverse several bits at a time and produce fast lookups.
- Employs data structures with multi-bit nodes optimized to perform efficiently in the prefix interval 16 to 24.
- Its nodes are represented with bitmaps that can be processed fast in hardware and require small storage.
- The updates in the nodes are executed by well defined routines and in deterministic time.

BOS Trie Nodes

The key ingredient of BOS is a trie node that can hold prefixes of lengths from 0 to 7 bits. This trie has 8 levels and therefore the total number of possible prefixes that can be accommodated are 2^8-1=255. We can use a bitmap to represent all the possible prefixes and this needs at least 255 bits as presented in Lulea [13]. According to this representation every prefix is correlated with a specific bit position inside the bitmap. If a specific bit is set then it is denoted that the corresponding prefix exists.

Consider a trie that can accommodate prefixes with lengths from 0 to 3 bits as shown in Figure 5-2. The prefix with length 0, namely *, is assigned with number 0, the prefix with length 1 and the prefix bit set to 0, namely 0*, is assigned with number 1, the prefix with length 1 and the prefix bit set to 1, namely 1*, is assigned with number 2 and so on as Figure 5-2 presents. Moreover, the level of the trie where a specific prefix is located is equal to its length.

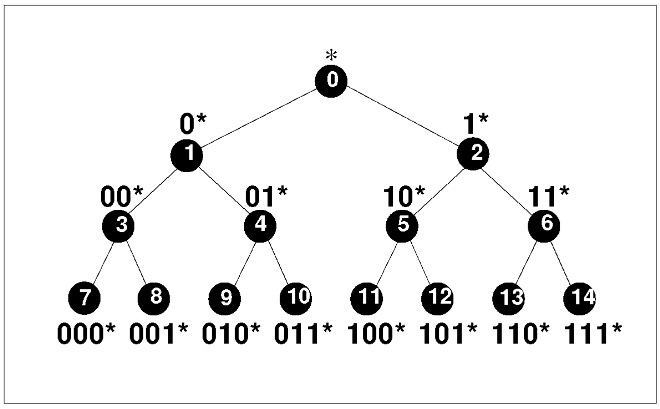


Figure 5-2 Prefix trie that supports prefixes up to length 3

We can derive a formula that correlates the length and the decimal value of a prefix with a number. Prefix with length 0 is assigned number 0 and all the other prefixes use the following formula:

$$Prefix_{NO} = PrefixValue + 2^{PrefixLength} - 1$$

The assigned prefix number can be used to indicate a specific bit position inside the bitmap. The bitmap that can accommodate all prefix lengths from 0 to 7 needs 255 bits and this means that even for a single prefix in this range, the trie node needs 32 bytes. We can prevent this memory waste and partition this trie in 17 subtries where each subtrie can support prefixes with lengths 0 to 3 as shown in Figure 5-3. We store the prefixes that have length 0 to 3 in the subtrie numbered 0 and the prefixes of greater length, namely 4 to 7, to an appropriate subtrie. The appropriate subtrie for the prefixes that have length 4 to 7 is defined by the 4 MSB of the prefix. The prefixes that have their 4MSB equal to 0000 are stored in the subtrie numbered 1, the prefixes that have their 4MSB equal to 0001 are stored in the subtrie numbered 2 and so on as Figure 5-3 presents.

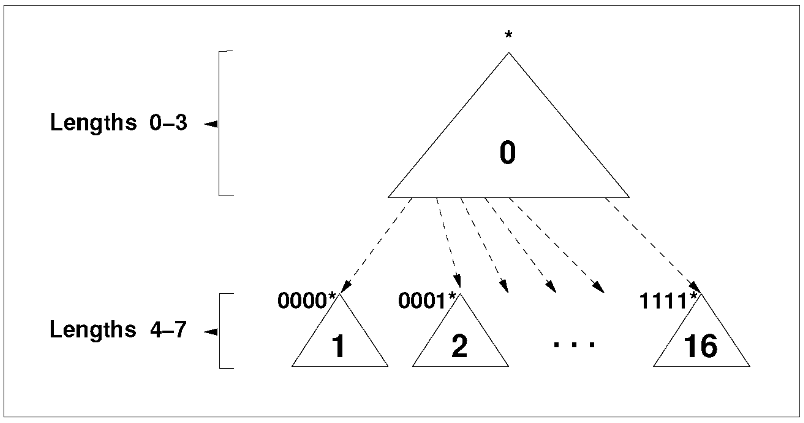


Figure 5-3 Trie partitions

We can derive a formula that correlates the length and the MSB of a prefix with a subtrie number. Prefixes with length 0 to 3 are stored in the subtrie 0 and for the prefixes of lengths from 4 to 7 we use the following formula to find the subtrie number:

$Subtrie_{NO}$ = PrefixValue[0:3] + 1

BOS, now uses the subtrie partitioning described in the last paragraph and the tries that support 0 to 3 length prefixes to represent the trie node that can support 0 to 7 length prefixes. To store efficiently the information about the subtries we define a bitmap (TrieBmp). In TrieBmp we correlate each bit with a specific subtrie according to the $Subtrie_{NO}$ formula. When a bit inside TrieBmp is set then it means that the corresponding subtrie has a least 1 prefix active. For every active subtrie we need the information about the included active prefixes, therefore we define another bitmap (PrefixBmp). In PrefixBmp we correlate each bit with a specific prefix according to the $Prefix_{NO}$ formula. When a bit inside PrefixBmp is set then it means that the corresponding prefix is active.

The partitioning of 8-bit tries into smaller 4-bit subtries gives the flexibility to save only the necessary prefix bitmaps (active) and not all of them. The trie bitmap needs 17 bits and each prefix bitmap needs 15 bits. This partitioning can be efficiently implemented by the dynamic memory management scheme discussed in subsection

3.3.5 because thevariable number of prefix bitmaps requires pointers to variable size blocks.

The associated information for each prefix is considered an N-bit quantity (the data associated with each rule), say 16-bits, and should be stored along with the prefix bitmap. Since more that one prefixes could be active we also need dynamic pointers to variable size blocks. So along with the prefix bitmap we save a pointer to the associated prefix data.

To locate the subtrie of a specific prefix in the trie bitmap we use the subtrie formula below, where T_{index} indicates the bit position of the actual subtrie number.

```
▪ If the prefix has length 0-3 then :
  $T_{index}$ = 0
▪ If the prefix has length 4-7 then :
  $T_{index}$ = prefix[0:3] + 1
```

To locate a specific prefix in the prefix bitmap we present the formula shown below, where P_{index} indicates the bit position of the actual prefix number in a specific subtrie.

```
▪ If $T_{index}$ = 0
    o If the prefix inside the trie has length 0 then :
      $P_{index}$ = 0
    o If the prefix inside the trie has length 1 then :
      $P_{index}$ = prefix[0] + 1
    o If the prefix inside the trie has length 2 then :
      $P_{index}$ = prefix[0:1] + 3
    o If the prefix inside the trie has length 3 then :
      $P_{index}$ = prefix[0:2] + 7
▪ If $T_{index}$ != 0
    o If the prefix inside the trie has length 0 then :
      $P_{index}$ = 0
    o If the prefix inside the trie has length 1 then :
      $P_{index}$ = prefix[4] + 1
    o If the prefix inside the trie has length 2 then :
      $P_{index}$ = prefix[4:5] + 3
    o If the prefix inside the trie has length 3 then :
      $P_{index}$ = prefix[4:6] + 7
```

In order to be able to efficiently search the blocks that are generated by our dynamic memory management scheme we have to have the prefix bitmaps and the associated prefix information sorted inside the blocks. The prefix bitmap for the first active subtrie should be placed first in the variable size block, the second in the second position etc. Moreover this indicates that we should know the number of set bits in the bitmap, fortunately this is a trivial operation for hardware to perform. The requirement for dynamic memory management generates an additional complexity in insertions or updates since the variable size blocks need to be resized appropriately and put sorted. This operation can be handled easily since resizing and sorting is limited to 17 nodes.

To illustrate the data structures used by BOS we introduce an example with the prefixes shown in Table 5-2. The two leftmost columns have the actual prefixes and the associated information and the two rightmost columns show the internally represented subtrie and prefix number pairs. As calculated, a general view of the data structure needed to store the prefixes of the example is shown in Figure 5-4.

Prefix [0:6]	Associated Info	Subtrie Number	Prefix Number
00001*	23	1	2
0000101*	47	1	12
0000110*	7	1	13
01*	15	0	5
100*	121	0	11
1001*	36	10	0
1100*	51	13	0
110011*	3	13	6

Table 5-2 Prefix example

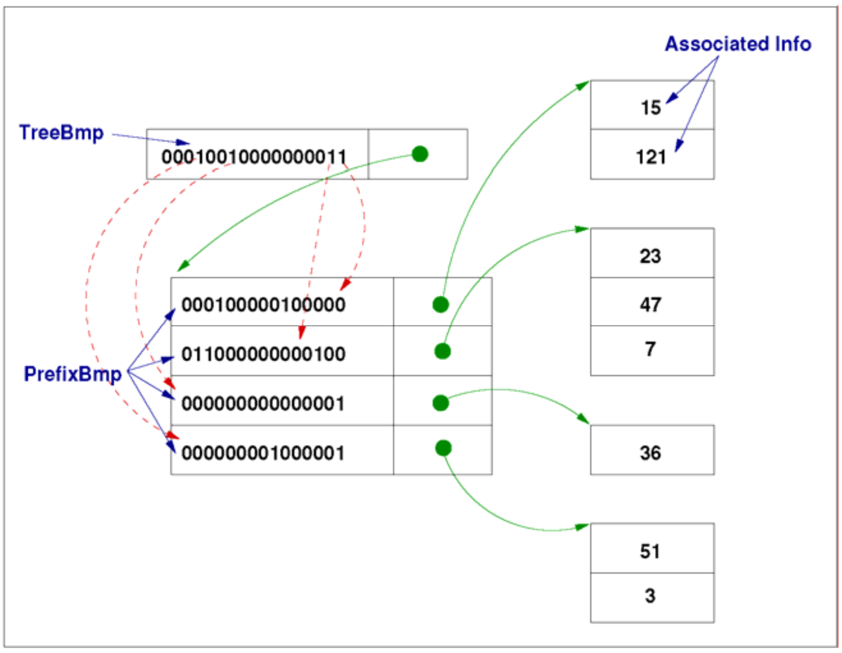


Figure 5-4 Trie data structure example

For a given 7-bit value, BOS should first find the candidate subtries that could match a certain prefix and then the candidate prefixes, inside the subtrie, that could also match. Tracking the longest one is the solution. The candidate subtries are always two:

```
▪  T1_index= 0 and
▪  One of the subtries 1-16 depending on the value
   T2_index = value[0:3] + 1.
```

Inside the 2 subtries the candidate prefixes are four:

```
▪  for T1_index :
      o  P1_index = 0
      o  P2_index = value[0] + 1
      o  P3_index = value[0:1] + 3
      o  P4_index = value[0:2] + 7
▪  for T2_index:
      o  P1_index = 0
      o  P2_index = value[4] + 1
      o  P3_index = value[4:5] + 3
      o  P4_index = value[4:6] + 7
```

We check the bit positions in TrieBmp for the 2 subtries and if both exist we give priority to the second subtrie which produces longer prefixes. Inside a matching subtrie we check all the bit positions in PrefixBmp for the 4 prefixes by giving priority to the fourth prefix which is the longest. The associated information for a matched prefix is retrieved by the node indicated by the pointer stored at the node of the matched prefix.

BOS Tables

BOS scheme uses the trie nodes for all the distinct 7-bit prefix lengths inside the 32-bit address space. BOS in its simplest form (BOS-SIMPLE) has trie nodes for the following prefix intervals:

i. 0-7,
ii. 8-15,
iii. 16-23,
iv. 24-31 and
v. 32

To hold the root nodes for the prefixes in each distinct interval, BOS-SIMPLE uses several tables as shown in Figure 5-5. For the interval 0-7 we have a single entry for root called TBL0. For interval 8-15 we have 2^8 possible roots, therefore we use a 256-entry table called TBL8 and the indexing is done with the first 8-bits of the prefix. For interval 16-23 we use a 2^{16}=65536 table called TBL16 and uses the first 16-bits of the prefix as index. For interval 24-31 we don't use 2^{24} entries because it would lead to great storage waste since no routing table could have 16777216 prefixes in this interval. Instead we use 2^{16} entries in table TBL24 and indexing is done by hashing the first 24-bits of the value. The collisions that occur due to hashing are handled with pointers to variable size blocks. For the 32 bit prefixes we use only 2^{12} entries in table TBL32, since most routing tables have few entries in this interval, and addressing is done by hashing. Collisions in this table are also handled with variable size blocks.

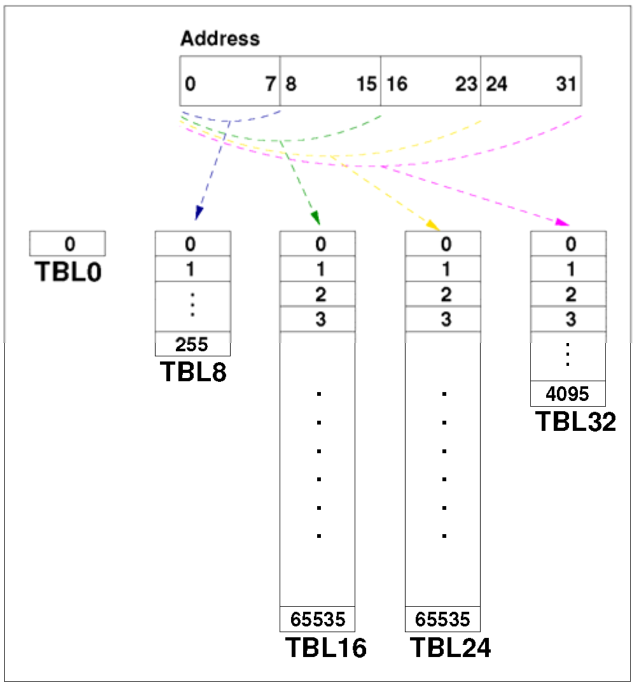


Figure 5-5 BOS Tables

Note that all distinct intervals are independent and this gives us the flexibility to start searching for a prefix from the middle of the address space. Searching sequentially would require to lookup all 5 tables but we can use a binary search type of access and limit the lookups to 3 or less. Furthermore, we can implement parallel searches in hardware if each table is stored in a separate memory.

Indexes in TBL24 are generated by the use of the XOR function in the first 24 bits of the prefix and a 16-bit address is produced as follows:

```
TBL24_index = prefix(8:23) xor (0000,prefix(0:7),0000)
```

Indexes in TBL32 are also generated by XOR function and the 12-bit address is produced as follows:

```
TBL32_index = prefix(4:15) xor prefix(20:31)
```

The decision for the hashing functions described above is presented in Section 5.3.

To handle the collisions in TBL24 we use pointers to variable size blocks as mentioned above. The collision resolving nodes save the number of prefixes that collide and a pointer to the variable size block, as described in subsection 3.3.3. To identify a prefix in the block we need to save some information to distinguish between the prefixes. Fortunately, we don't need to save all 24-bits and we take advantage of the fact that the address has been produced by the actual prefix. Therefore a prefix located address A of TBL24 can be reproduced by the 16-bits of A and the first 8-bits(value) of the prefix as follows:

```
Prefix(0:7)  =  value(0:7)
Prefix(8:23) =  A(0:15) xor (0000,value(0:7),0000)
```

It is now clear that to resolve collisions, the quantity that must be kept in the variable size block is dependent of the hashing scheme, in our case the first 8-bits of the prefix. Additionally we keep a pointer to the basic trie node starting from this root.

In TBL32 collision handling is done the same way as in TBL24 but the quantity that must be kept here is 20-bits and there is no need for a pointer to a trie node (no longer prefixes exist) but only store the associated information itself. A prefix located in address A of TBL32 can be reproduced by the 12-bits of A and the first 20-bits(value) of the prefix as follows:

```
Prefix(0:19) =  value(0:19)
Prefix(20:31) = A(0:11) xor value(4:15)
```

BOS-BASIC searches the tables in specific sequence in order to minimize the number of accesses. Since 99% of the prefixes exist in the intervals 16-23 and 24-31, it is more likely to find the longest match there by examining the associated tables. At first we look in TBL16 and if a prefix match occurs then we can search in TBL24 and TBL32 to find a matching prefix. If lookups in TBL16 or TBL24 or TBL32 cannot find a match then we proceed to search TBL8 and if there is not any match again we finally search in TBL0. The sequence of lookups is the following:

TBL16 → TBL24 → TBL32 → TBL8 → TBL0

If after TBL32 a match was produced then our lookup process does not proceed to the next tables.

5.2 BOS optimizations

BOS, as described above, has some weaknesses in terms of storage efficiency since it contains redundant information in some special cases. This section proposes some optimizations in the basic scheme and explores the trade-off between storage requirements and the number of memory accesses. Moreover we quote some modifications that could allow BOS to become the single field solution for the general decomposed N-dimensional packet classification, described in detail in Chapter 6.

5.2.1 Prefix Node Optimization

We observe an irritating feature of the trie node data structure that keeps the associated prefix node information. In case only a single prefix is active inside a subtrie, we need to store a pointer to the prefix information node and then acquire these data. This waste can be avoided by keeping the associated data in the node itself instead of the pointer to the data. This modification requires a flag to indicate that there is only a single prefix. Additionally, instead of keeping the prefix bitmap we can only keep the prefix number. The prefix number needs 4-bits and the flag 1-bit. In total, now, we use 5-bits instead of 15 required for the bitmap. By this trick we save one memory word that would keep the associated data and we also save the extra memory access to acquire these data.

5.2.2 Trie Node Optimization

In the cases where only one prefix is active inside the entire trie then there is only one subtrie active. Normally we should store a pointer to the prefix node and then lookup for the prefix number and the associated data. We can improve this case and save memory by keeping all the information in the basic trie node, similarly to 5.2.1. Instead of the 17-bit trie bitmap and the prefix node pointer, we keep the subtrie number in 5-bits, the prefix number in 4-bits and an extra flag to indicate this special case. Moreover, we don’t save a pointer to prefix node but the actual associated prefix data. This optimization saves the memory of the prefix node and the extra memory access.

5.2.3 TBL16 Optimization

BOS adopted the use of TBL16, which has 64K entries (2^{16}), in order to reach the prefix interval 16 to 23 very fast. This practice is very common in literature [13][37][38] although it could waste memory since many of the 64K entries could be empty. To reduce the storage requirements of BOS the large 64K static tables TBL16 and TBL24 should probably be shrinked. At first glance TBL16 is likely the most underutilized table since not all the entries could be roots of tries. We can save memory by reducing TBL16 table into a smaller one, say 16K entries, and use hashing for indexing. For indexing now we need 14-bits and we have to produce each index by the first 16-bits of the prefix. Hashing the address into 14-bits requires saving 2-bits in the entries to identify a root of prefixes.

In this hashing we can take advantage of the fact that CIDR [2] aggregates consecutive routing prefixes from the early class-based addressing and gives us information about the first 2-bits of each address. When aggregating subnets from the old Class A addresses then bits(0:1) have the value 00 or 01 but these prefixes would have length lower that 8, so they don't affect TBL16. Similarly for aggregation of Class B subnets, the addresses bits(0:1) have the value 10 but their prefix length is lower than 16. Aggregation of Class C subnets has value 11 in bits(0:1) of the address and the prefix lengths exceeds 16. The prefixes from Class C addresses well affect TBL16 and it is likely that most roots in TBL16 come from these addresses. However, routing protocols like BGP [39] implement, what is called route aggregations, so as to be efficient. This route aggregation is generally based on the associated prefix information (namely NEXT_HOP) and can create prefixes longer than 16 from Class A and Class B addresses. Because CIDR is widely used we decided to use bits (2:15) for indexing of the TBL16 table. Therefore we define a new index for TBL16 which is:

$TBL16_{index} = prefix(2:15)$

The performance of the above indexing scheme is discussed in Section 1.3.

In case of collisions inside TBL16 we can use the solution of sorted roots inside variable size blocks as described for TBL24 and TBL32. In TBL16 the maximum number of collisions is limited to 4 and 2-bits can identify the root of the prefixes. It is obvious that by reducing TBL16 we trade the storage for the number of memory accesses to locate a specific root for prefixes in the interval 16-23. We make TBL16 4 times smaller by sacrificing one possible extra memory access and a pointer.

Reduction of TBL16 is also helpful if BOS is to be used in decomposed N-dimensional classification. The number of rules-prefixes in classification databases is notably smaller than in routing tables. In related literature [40][20][26] the number of rules is lower than 2000 and the number of distinct prefixes is even smaller. This denotes that smaller TBL16 can produce better memory utilization. If the target application is packet classification with a small number of rules, we can even shrink TBL16 to 4K entries.

5.2.4 TBL24 and TBL32 Optimization

BOS assigns a 64K entry table for TBL24 to save the trie nodes for the roots of prefix lengths equal or longer than 24. In order to avoid underutilization of this table we propose a more fine-grained approach with dynamic memory management. We decide to link the entries of TBL16 with the entries that extend further than 24, so as to share the common 16-bit prefix, by using dynamic pointers to 256 entries' blocks (BLK256). Every entry of TBL16 has a pointer to the basic trie node for lengths 16-23 and a pointer to a 256 block that saves roots for lengths 24-31 if applicable.

For prefixes that have length 24 or more we allocate a BLK256 and link to the corresponding entry of TBL16. The first 16 bits of the prefix index TBL16 (or the 14 rightmost from that first 16 according to TBL16 optimization) and the next 8 bits index the corresponding BLK256. When a new prefix of length greater or equal to 24 is inserted we first check the corresponding entry in TBL16 and if a BLK256 is linked we insert the prefix in the specific BLK256 otherwise we allocate a new block.

There are cases where a BLK256 is underutilized because it contains much less than 256 entries. To avoid this possible underutilization we can assign the same BLK256 in more that one entries of TBL16. This means that the prefixes in this block could have the first 16-bits different, so this should be the information that we save in BLK256 to distinguish the prefix roots. Roots inside BLK256 are still indexed with the last 8-bits and every entry has a pointer to the associated trie node. The scheme assigns a BLK256 to more than one TBL16 entries and has counters to keep the utilization of the block. If a BLK256 is highly populated and reaches its limits, namely 256 roots, then the scheme assigns a new block for the forthcoming 24 length roots.

When roots with different first 16-bits are inserted in a BLK256 it is possible that we have collisions since indexing is done only by the last 8-bits. Handling these collisions is trivial with our variable size blocks but it could lead to extra memory accesses. We can trade the collision resolving accesses by setting the utilization factor of BLK256 to lower limits. We can decide not to have fully populated BLK256 but allocate new block when the number of roots inside a BLK256 is lower than 256. These limits can be 224 or 192 or 160 or 128 which means that many entries in the block can be empty. This waste can help us have fewer collisions in the blocks and therefore fewer memory accesses.

The same strategy is used for TBL32 which is transformed into multiple blocks of BLK256 linked to the corresponding BLK256 blocks containing the items with length 24. This multi-linking scheme gives us the flexibility to save the obligatory memory accesses to TBL24 and TBL32 when no prefix exists. Now lookups start at TBL16 and if a link to further roots exists then we lookup to the corresponding BLK256. Further access to 32 length nodes is done only if a link from a BLK256 that holds 24 length nodes exists. TBL8 and TBL0 are now accessed only if no entry matches in TBL16 and no block for 24 roots exists. The form of BOS after the table optimizations is shown in Figure 5-6.

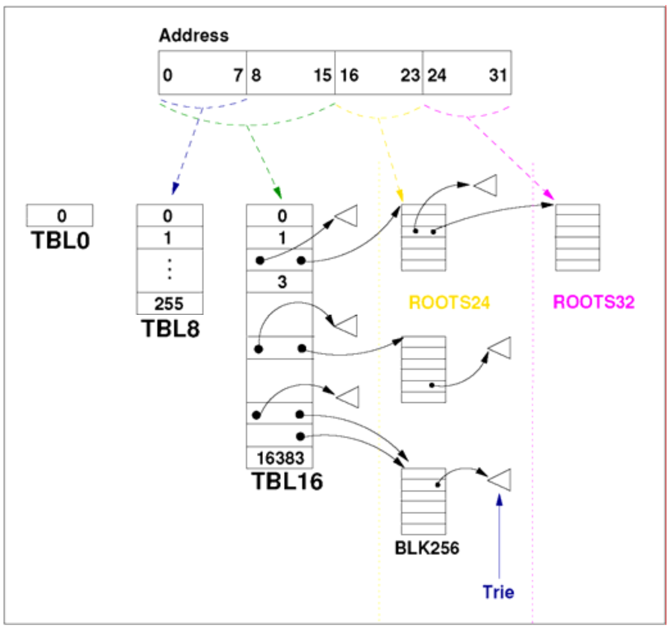


Figure 5-6 BOS with BLK256

The dynamic management of roots 24 and 32 is also helpful if BOS is to be used in decomposed N-dimensional classification. The number of rules-prefixes in classification databases is notably smaller than in routing tables as discussed above. We have the flexibility to create as many BLK256 as required by the number of 24 and 32 existing roots and we can fine-tune the thresholds, where new BLK256 are allocated, depending on our memory budget.

5.2.5 All prefix match

BOS is designed to solve the Longest Prefix Matching (LPM) problem but it can easily adapt to support All Prefix Matching (APM); BOS should return all the matching prefixes during its way to find the longest one. APM is essential for decomposed N-dimensional classification where multiple field searches can be converted into several single field searches, as described in Chapter 6.

BOS can support APM by searching in parallel or sequentially all of its tables and blocks to find all the prefixes. BOS simple should search TBL0 then TBL8 and so on until it finds all prefixes. The sequence of searching in tables for BOS-SIMPLE is:

TBL0→ TBL8 →TBL16 → TBL24 → TBL32

In every table, when an active root exists BOS searches for all the matching prefixes inside the trie node. Inside the trie node the maximum number of matching prefixes is 8. At first, BOS should lookup in all the candidate subtries; these are subtrie 0 and the subtrie indicated by the prefix value. Inside every subtrie BOS looks for matches in all four candidate prefixes. Once a match is found the associated prefix information is returned.

If BOS is implemented with the proposed TBL24 and TBL32 optimizations then the sequence of lookups is the same as BOS-SIMPLE but searches in ranges further than 23 proceed only is a link from TBL16 exists. Therefore, search in BLK256s for TBL24 is performed if a link from TBL16 exists. Similarly, searches in BLK256s for TBL32 are performed only if a block is linked to the corresponding BLK256 which contains the 24-bit roots.

5.3 Simulation Results and Performance

In this subsection we present our simulation results based on real routing tables and details about the storage and speed of the presented scheme. We first analyze the prefixes of several real world routing tables and count the number of tries nodes that should be used by our scheme. We present our results before and after the optimizations proposed in Section 5.2 and illustrate their effectiveness. BOS storage and speed performance is based on assumptions for the hardware and memory configurations.

5.3.1 Hashing functions and Indexing

TBL24 and TBL32 Hashing

Indexing in TBL24 and TBL32 is implemented by hashing the most significant bits of each prefix. The decisions for the hashing functions were taken by comparing the performance (in terms of collisions) achieved by a large number of them when applied in real routing tables. The hashing functions we created and tested are following:

- *HSH24_1$_{index}$ = prefix(0:15)*
- *HSH24_2$_{index}$ = prefix(0:15)* ***xor*** *(0000,prefix(16:23),0000)*
- *HSH24_3$_{index}$ = prefix(8:23)* ***xor*** *(0000,prefix(0:7),0000)*
- *CRC-16*

The first three functions require 8 additional bits to be saved in the collision resolving nodes so as to be able to distinguish the collided prefixes and the CRC function requires all 24-bits to be stored. The maximum and the average number of collisions produced by each hash function are shown in Table 5-3. The simulation results, presented in the table, show that the hash functions which use XOR applied in the first 24-bits of each prefix, generate more uniformly distributed values and therefore the indexes are better shuffled that just using the 16 most significant bits (HSH24_1). CRC-16 has the better results in terms of collisions and HSH24_3 is very close. Despite the fact that CRC-16 is slightly better than HSH24_3, we decide to use HSH24_3 function because it requires only 8-bits to be saved in the collision nodes instead of 24-bits that CRC-16 requires.

Routing Table (Total Prefixes)	Hash function	Max Collisions	Average Collisions
AADS 10/2000 (39876)	HSH24_1	209	34,09
	HSH24_2	12	3,81
	HSH24_3	4	**1,30**
	CRC-16	5	1,32
MAE-EAST 01/2000 (60004)	HSH24_1	199	36,98
	HSH24_2	16	6,24
	HSH24_3	5	**1,49**
	CRC-16	5	1,48
PAIX 10/2000 (91278)	HSH24_1	228	41,83
	HSH24_2	20	7,54
	HSH24_3	6	**1,77**
	CRC-16	8	1,73

Table 5-3 Hash functions performance

TBL16 Indexing

As discussed in subsection 5.2.3 (TBL16 optimization) we decided to shrink TBL16 and use the last 14-bits from the 16 leftmost as index to the table. This decision was guided by the form of the internet addressing but is also confirmed by simulation results. We test in simulation the following indexing functions:

- *IDX16_1$_{index}$ = prefix(0:13)*
- *IDX16_2$_{index}$ = prefix(1:14)*
- *IDX16_3$_{index}$ = prefix(2:15)*

All functions use a 14-bit portion from the first 16-bits of the prefix and their performance in presented in Table 5-4.

Routing Table (Total Prefixes)	Index function	Max Collisions	Average Collisions
AADS 10/2000 (39876)	IDX16_1	4	2,65
	IDX16_2	4	1,62
	IDX16_3	3	**1,26**
MAE-EAST 01/2000 (60004)	IDX16_1	4	3,12
	IDX16_2	4	1,74
	IDX16_3	3	**1,42**
PAIX 10/2000 (91278)	IDX16_1	4	3,30
	IDX16_2	4	1,87
	IDX16_3	3	**1,52**

Table 5-4 Performance of BOS indexing functions

It is clear from these results that the last 14-bits carry the most "important" information that differentiates prefixes by one other. Indexing scheme IDX16_3 is suitable to be used for TBL16 since it generates fewer collisions than the others and thus triggers less memory accesses.

5.3.2 Storage requirements

This subsection presents the storage requirement of our scheme for several routing tables. We present the requirements of our scheme before and after the proposed optimizations. To calculate the total storage for BOS we need to count the number of the active subtries for the trie nodes and the total number of stored prefixes. Additionally, we have to calculate the total amount of memory needed for the static tables. We simulated our scheme and the results are shown in Table 5-5.

On a static table entry we have to fit a 17-bit trie bitmap and a possible 16-bit pointer to the subtrie nodes; this assumes a memory word equal or larger than 33-bits. A popular memory word size for commercial off-chip SRAMs is 36-bits and can match our requirements. Hence, for every memory word on BOS we assume 36-bits (4,5 bytes).

Routing Table (Total Prefixes)	**Interval**	**Active Roots**	**Active Tries**	**Active Prefixes**
AADS 10/2000 (39876)	0-7	0	0	0
	8-15	52	132	179
	16-23	5555	10542	15394
	24-31	20705	21765	22968
	32	1335	1335	1335
	Total	**27647**	**33774**	**39876**
MAE-EAST 01/2000 (60004)	0-7	0	0	0
	8-15	85	315	464
	16-23	8587	17180	25638
	24-31	33776	33815	33891
	32	11	11	11
	Total	**42459**	**51321**	**60004**
PAIX 10/2000 (91278)	0-7	0	0	0
	8-15	91	377	595
	16-23	10248	23193	37128
	24-31	53167	53217	53548
	32	7	7	7
	Total	**63513**	**76794**	**91278**

Table 5-5 Routing Tables Properties

For each active subtrie we have to allocate dynamically a memory word to fit the 15-bits of the prefix bitmap and the 16-bit pointer to the actual prefix node; those can fit in one memory word. As a result the memory words for the subtries are equal to the total number of subtries. For the prefixes we allocate a memory word every two prefixes since we can fit two associated prefix data in a word. We have assumed 16-bit data, which is what is basically assumed in the majority of similar studies such as

[xRef]. Therefore the total number of the bytes for the trie nodes including the associated prefix information is:

$$Total_{bytes} = (no\_active\_tries + no\_prefixes/2) * 4,5$$

As far as the static tables are concerned, we calculate the total amount of memory needed in Table 5-6.

Table	Entries	Total Bytes
TBL0	1	4,5
TBL8	256	1152
TBL16	65536	294912
TBL24	65536	294912
TBL32	4096	18432
Total	**135425**	**609412,5 (595Kb)**

Table 5-6 Static tables memory requirements

In Table 5-7 we present the final storage requirements for the simple BOS scheme without any optimizations, including the collisions in TBL24 and TBL32.

Routing Table (Total Prefixes)	Static Tables (Kbytes)	Collision Nodes (Kbytes)	Trie Nodes (Kbytes)	Total (Kbytes)
AADS 10/2000 (39876)	595	24	236	**855**
MAE-EAST 01/2000 (60004)	595	59	357	**1011**
PAIX 10/2000 (91278)	595	128	538	**1264**

Table 5-7 BOS simple storage

Results show that the static tables consume nearly 50% percent of the total storage. The collision nodes required are relatively small and require few Kbytes but the trie nodes possess a respectable part of the overall storage.

Trie Node Optimizations

By applying the prefix node optimization (subsection 5.2.1) we don’t need an extra memory word for the prefix information in the case where there is only one prefix inside a subtrie. We count the number of subtries that have exactly one prefix and present them in Table 5-8.

Routing Table (Total Prefixes)	Total Active Subtries	Single Prefix Subtries	% of Single Prefix Subtries
AADS 10/2000 (39876)	33774	28994	**85,8%**
MAE-EAST 01/2000 (60004)	51321	46745	**91%**
PAIX 10/2000 (91278)	76794	69562	**90,5%**

Table 5-8 Single Prefix Subtries

The results from the routing tables show that more than 85% of the subtries existing in BOS have exactly one prefix and therefore we don't need extra memory words to save them. The number of single prefix subtries gives us the number of prefixes that we don't need to save in separate nodes. It is obvious that the prefix node optimization is significantly effective since it saves many wasted memory words. We recalculate the required storage for BOS after the prefix node optimization in Table 5-9. We observe that prefix node optimization improved the initial trie node storage approximately 28% and the total storage requirement by 7% - 12%.

Routing Table (Total Prefixes)	**Static Tables (Kbytes)**	**Collision Nodes (Kbytes)**	**Trie Nodes (Kbytes)**	**Total (Kbytes)**
AADS 10/2000 (39876)	595	24	172	**791**
MAE-EAST 01/2000 (60004)	595	59	255	**909**
PAIX 10/2000 (91278)	595	128	385	**1108**

Table 5-9 Prefix Node Optimization Storage

A further optimization discussed was when a root has just one subtrie and just one prefix (trie node optimization 5.2.2). In this case all the information for this subtrie is saved in the corresponding static table. In order to measure this optimization's effectiveness we have to calculate the roots that have exactly one subtrie and exactly one prefix; single prefix roots. Moreover we have to calculate the new number of single prefix subtries since the single prefix roots are a subset of the single prefix subtries. Our results are presented in Table 5-10.

Routing Table (Total Prefixes)	**Total Active Subtries**	**Single Prefix Roots**	**% of Single Prefix Roots**	**Single Prefix Subtries**	**% of Single Prefix Subtries**
AADS 10/2000 (39876)	33774	23398	**69,2%**	5596	**16,6%**
MAE-EAST 01/2000 (60004)	51321	39372	**76,7%**	7373	**14,3%**
PAIX 10/2000 (91278)	76794	59181	**77%**	10381	**13,5%**

Table 5-10 Single prefix roots

Calculations on the routing tables show that more than 70% of the roots existing in BOS have exactly one prefix and therefore we don't need allocation of extra memory words; we can save them in the static table. We can take advantage of the common case that trie node optimization reveals us and save memory words. We recalculate the required storage for BOS after the trie node and prefix node optimization in Table 5-11.

Routing Table (Total Prefixes)	Static Tables (Kbytes)	Collision Nodes (Kbytes)	Trie Nodes (Kbytes)	Total (Kbytes)
AADS 10/2000 (39876)	595	24	69	**688**
MAE-EAST 01/2000 (60004)	595	59	82	**736**
PAIX 10/2000 (91278)	595	128	125	**848**

Table 5-11 Trie and Prefix Node Optimization Storage

Trie node and prefix node optimization together, improved the initial trie node storage 70% – 77% and the total storage requirement 19% - 33%.

Static Table Optimizations

Moving to the optimization of the static tables, as described in subsection 5.2.3, we decided to shrink TBL16 to 16K entries instead of 64K at the cost of an extra memory access and some extra memory space in the case of the additional collisions. When we analyze the routing tables in Table 5-5 we see that the active roots of the interval 16-23 are significantly less than 64K on all the examined routing tables. We find that in the worst case we have 10248 active roots and a 16K table is enough for this interval. These results confirm our arguments about few active roots inside TBL16. We calculate the collisions and the required storage for TBL16 in this case and present our results in Table 5-12.

Routing Table (Total Prefixes)	Original TBL16 (Kbytes)	Optimized TBL16 (Kbytes)	Collision Nodes (Kbytes)	Total (Kbytes)
AADS 10/2000 (39876)	288	72	6	**78**
MAE-EAST 01/2000 (60004)	288	72	15	**87**
PAIX 10/2000 (91278)	288	72	21	**93**

Table 5-12 TBL16 Storage Optimization

The decision for smaller TBL16 gives us 67% - 73% better storage requirements for TBL16 by the cost of an extra memory access.

In subsection 5.2.4 we have also proposed the replacement of TBL24 and TBL32. According to this optimization we replace these tables with dynamic memory blocks of 256 entries and link them to TBL16. This means that for the calculation of storage we have to count the number of BLK256 and the collision nodes inside them. We simulate our approach and present the results in Table 5-13. We have used a number of different threshold values for the utilization of BLK256s to illustrate their effectiveness in terms of collisions. The thresholds clearly show that underutilized blocks can provide better average access performance at the cost of additional storage.

Routing Table (Total Prefixes)	Threshold	Interval	Number of Blocks	Maximum Collisions	Average Collisions
AADS 10/2000 (39876)	256	24-31	90	7	**2,01**
		32	6	11	**2,45**
	192	24-31	120	6	**1,75**
		32	7	8	**2,15**
	128	24-31	179	5	**1,50**
		32	11	7	**1,77**
MAE-EAST 01/2000 (60004)	256	24-31	133	7	**2,00**
		32	1	1	**1,00**
	192	24-31	177	6	**1,74**
		32	1	1	**1,00**
	128	24-31	265	6	**1,50**
		32	1	1	**1,00**
PAIX 10/2000 (91278)	256	24-31	210	8	**2,00**
		32	1	2	**1,28**
	192	24-31	279	6	**1,74**
		32	1	2	**1,28**
	128	24-31	419	6	**1,49**
		32	1	2	**1,28**

Table 5-13 Dynamic BLK256 for TBL24 and TBL32

After the static tables optimizations we recalculate the required storage for BOS in Table 5-14.

Routing Table (Total Prefixes)	Thres.	Trie Nodes (Kb)	Modified Static Tables (Kb)	TBL16 Collision Nodes (Kb)	Dyn. Blocks (Kb)	Dyn. Collision Nodes (Kb)	Total (Kb)
AADS 10/2000 (39876)	256	69	73	6	107	64	**319**
	192	69	73	6	143	53	**344**
	128	69	73	6	214	40	**402**
MAE-EAST 01/2000 (60004)	256	82	73	15	150	96	**416**
	192	82	73	15	200	78	**448**
	128	82	73	15	299	59	**528**
PAIX 10/2000 (91278)	256	125	73	21	237	148	**604**
	192	125	73	21	315	124	**658**
	128	125	73	21	472	92	**783**

Table 5-14 Fully optimized BOS storage

We can calculate the overall efficiency of BOS by computing a metric that indicates the average storage space in terms of bytes per prefix. The absolutely essential information includes 32 bits for the prefix itself, 6-bits for the length of the prefix and 16-bits for the associated prefix information. So we need at most 54 bits or 6,75 bytes per prefix. Our scheme requires the values shown in Table 5-15.

Routing Table (Total Prefixes)	**Thres.**	**Total (Kb)**	**Bytes/Prefix**
AADS 10/2000 (39876)	256	319	**8,19**
	192	344	**8,83**
	128	402	**10,32**
MAE-EAST 01/2000 (60004)	256	416	**7,09**
	192	448	**7,64**
	128	528	**9,01**
PAIX 10/2000 (91278)	256	604	**6,77**
	192	658	**7,38**
	128	783	**8,78**

Table 5-15 BOS bytes per prefix

We can see that as the routing tables grow, BOS is more efficient in terms of storage and provides lower average bytes per prefix that approximate the “perfect” reference solution. The overhead of BOS comes from the data structure that we use, contrarily the perfect approach does not imply any data structure or organization, neither assumes any lookup mechanisms.

5.3.3 Lookup Performance

In this subsection we analyze the lookup performance of BOS scheme in terms of memory accesses. BOS lookup performance differs before and after the optimizations that were proposed. For every interval of the address space we can compute the worst and average latency of lookups inside a trie node.

BOS Simple

BOS needs one memory access to acquire the data stored in the root node, then it follows the pointer to the candidate subtrie node and then the pointer to the prefix node. The normal case requires 3 memory accesses for each interval. The worst case is when we don’t find a prefix match inside the 1st candidate subtrie, so we seek in the 2nd subtrie and then to the prefix node, this case requires 4 memory accesses. In case neither prefixes inside the subtries match, we spend 3 memory accesses and we don’t follow any pointer to the prefix node. BOS-SIMPLE has 5 intervals and 5 distinct tables for each prefix interval, therefore in the worst case we need to search all of them so as to determine if a match exists. We need 3 memory accesses for the first 4 tables and 4 for the last table. This sums to 16 memory accesses in the worst case. If the tables are in separate memories we can search all of them in parallel and then the worst case is 4 memory accesses.

BOS with optimized trie nodes

If we use prefix and trie node optimizations we can achieve better average number of memory accesses but the worst case will remain the same. According to the results presented in Table 5-10, nearly 70% of the roots have a single prefix and nearly 14% have a single subtrie. On single prefix roots we need 1 memory access and on single subtrie roots we need 2 memory accesses. Hence, the average number of memory accesses to locate a prefix in an interval is calculated to be 1,62. We see that the average case after the node optimizations is 60% better. In case we use sequential accesses to tables we need in total 8,1 memory accesses and in case we use parallel searches we need on average only 1,62 memory accesses.

BOS with optimized TBL16 and dynamic blocks.

Optimizations in TBL16 are rather helpful in terms of storage but significantly increase the number of memory accesses to locate a prefix inside an interval. For every colliding root inside TBL16 we need an extra memory access and due to that the average lookup latency in TBL16 is measured to be 2,62 memory accesses and the worst case 5 memory accesses.

By adding the dynamic memory blocks in our scheme may have saved storage but the average and maximum number of memory accesses is increased. For roots of prefix lengths 24 or more, we need to locate the block by accessing TBL16, then locate the specific root between the roots that collide, and then lookup inside the trie node. This sequence of accesses sums to 4,62 memory accesses on average and 11 on the worst case.

The summary of lookup performance for BOS with and without optimizations is presented in Table 5-16 and shows the number of memory accesses per lookup in every case. These results are an average of all the simulated routing tables.

Scheme	Average	Worst Case	Parallel
BOS-SIMPLE No opt.	**10,1**	16	4
BOS Opt. Nodes	**4,86**	16	1,62
BOS Opt. Tables	**12,1**	24	4,62

Table 5-16 Memory access performance of BOS

Lookup Perfomance and Link Speeds

According to our lookup performance we can calculate the efficiency of BOS as a forwarding engine in a high speed router. To calculate the network performance we assumed a certain speed of the memory and a pipelined hardware implementation that can provide one memory access per cycle. The results we present assume 2 possible memory configurations:

- 200Mhz off-chip synchronous SRAM
- 400Mhz on-chip synchronous SRAM

Table 5-17 presents the network performance of BOS counted in millions of packets per second (Mpps).

Scheme	**Off-chip SRAM 200Mhz**			**On-Chip SRAM 400Mhz**		
	Average (Mpps)	Worst Case (Mpps)	Parallel (Mpps)	Average (Mpps)	Worst Case (Mpps)	Parallel (Mpps)
BOS-SIMPLE No opt.	**20**	12,5	50	**40**	25	100
BOS Opt. Nodes	**41,2**	12,5	123,5	**82,3**	25	246
BOS Opt. Tables	**16,5**	8,3	43,3	**33**	16,6	86,6

Table 5-17 Network Performance of BOS in Mpps

If we assume the worst case of taking routing decisions for minimum sized IP packets (40 bytes) then the supported link speeds are shown in Table 5-18.

Scheme	**Off-chip SRAM 200Mhz**			**On-Chip SRAM 400Mhz**		
	Average (Gbps)	Worst Case (Gbps)	Parallel (Gbps)	Average (Gbps)	Worst Case (Gbps)	Parallel (Gbps)
BOS-SIMPLE No opt.	**6,4**	4	16	**12,8**	8	32
BOS Opt. Nodes	**13,2**	4	39,5	**26,4**	8	79
BOS Opt. Tables	**5,3**	2,67	13,8	**10,6**	5,3	27,7

Table 5-18 Network Performance of BOS in Gbps

Chapter 6

Bloom Filter Based Packet Classification

In this chapter we present Bloom Based Packet Classification (B2PC), our scheme for efficient packet classification. We developed a scheme suitable for pipelined hardware implementation which can be used as a classification engine for network streams. B2PC comprises of a 5-field search algorithm and decomposes multi-field classification rules into internal single field rules which are then organized in Bloom filter sets. The design of B2PC is optimized for the common case based on analysis of real world filter sets and uses the BOS single field technique which was described in Chapter 5.

6.1 Real Filter Sets

Researchers' attempts to discover better classification techniques are mainly focused in analysis of real world sets of classification rules. Many research groups have studied real classification data from commercial ISPs and access lists (ACLs) from enterprise networks to exploit the specific characteristics of these sets. The results from these surveys provide statistical characteristics of the filter sets and are valuable as a guide for the classification algorithms' designers.

The standard packet classifiers are 5-dimensional and their fields come from the Network Layer (L3) and the Transport Layer (L4) network packet fields. These fields are the following:

- Source IP address in 32-bits (L3)
- Destination IP address in 32-bits (L3)
- Source Port in 16-bits (L4)
- Destination Port in 16-bits (L4)
- Protocol in 8-bits (L4)

A filter in a classifier may specify all the fields with prefixes, ranges, exact values or wildcards[4].

There exist several studies of the specific characteristics of the real world classification rules. Primarily Gupta and McKeown published a number of observations regarding the characteristics of real filters sets [26], while others have performed analyses on real filter sets and published their observations [40][41]. The following key observations are a review of these studies:

I. Current filter sets' size are small, ranging from tens of filters to less than 5000 filters. However, it is not clear if the size limitation is "natural" or a result of the limited performance of packet classification solutions.

II. The protocol field is restricted to small set of values. TCP, UDP and wildcarded are the most common specifications.

III. Filters specify a limited number of unique transport port ranges. The specifications for port ranges vary and have definitions like 'greater than 1023' or '20 to 23'.

IV. The number of unique address prefixes matching a given address is typically five or less.

V. The number of filters matching a given packet is typically five or less.

VI. Different filters often share a number of the same field values.

VII. The number of unique field values is significantly less than the number of filters.

To evaluate the performance of classification schemes and algorithms it is important to test it with representative filter sets. The properties of the filter sets and the query patterns are essential to benchmark classification schemes and thus realistic filters and test patterns should both be used. D. Taylor has created ClassBench [42] to address this problem. ClassBench is a suite of tools for performance evaluation of classification algorithms and is publicly available. ClassBench involves a filter set generator that uses seeds from real filter sets to provide synthetic filter sets that accurately model real filters. Moreover, it includes a packet header generator that produces a sequence of packet headers to exercise a given filter set. This generator uses the Pareto Distribution[43] that is widely used to model the Internet traffic.

[4] Wildcards are used when we don't specify a value and want to represent all the possible values. The symbol used for wildcards is *.

6.2 B2PC Design and Description

B2PC design is driven by the observations presented in the last section. Our approach for packet classification lays on the idea of decomposition where multiple field searches are divided into many single field searches. The results of single fields are then combined to produce the final rule/filter match. We strive to design a packet classifier that supports 5-dimensional rules and provides the associated FlowID of a matching rule/filter for a given packet.

The fields we use are the standard supported by all 5D classifiers, namely two 32-bits IP addresses, two 16-bit ports and an 8-bit protocol. We allow the database to have at most 4096 of such rules, which seems enough according to the referenced observations. Consequently, each rule/filter of the database can be identified by a 12-bit FlowID value. An example filter set is shown in Table 6-1.

No	Src IP	Dest IP	Src Port	Dest Port	Protocol	Flow ID
1	139.91.70.*	147.52.16.*	*	*	TCP	10
2	139.91.*.*	147.102.*.*	*	21	TCP	14
3	139.91.*.*	147.27.*.*	< 1024	*	*	17
4	*.*.*.*	139.91.*.*	*	80	UDP	26
5	139.91.70.33	147.52.16.33	135	< 1024	TCP	31
6	139.91.70.36	147.27.*.*	< 1024	21	*	45
7	*.*.*.*	147.52.*.*	*	23	*	47
8	139.91.*.*	147.52.*.*	135	135	TCP	50
9	139.*.*.*	147.*.*.*	*	80	TCP	54
10	139.91.*.*	147.52.*.*	*	135	TCP	55

Table 6-1 Filter Set Example

6.2.1 Single Field Operations

Given the fact that we have followed the decomposition path a very efficient single field engine supporting both exact and prefix matches is essential. Hence, we decided to use the BOS scheme described in Chapter 5 as our single field engine. Each field of the rule can be inserted in a single field engine and be identified by the rule's FlowID. It should be noted that this single field lookup should not only report the longest prefix match but, instead all the prefixes that match; as discussed in subsection 2.2.3. Fortunately BOS has the capability to work as APM engine as described in subsection 5.2.5.

The rules regarding the IP address fields are specified as prefixes and this makes BOS an excellent solution. The Port fields are usually specified as ranges but can be transformed into prefixes with well known formulas [17]. Additionally, the

BOS engine that keeps the Port Fields should be finetuned since BOS provides all prefix match (APM) for 32-bit values and we intend to store only 16-bit values. Protocol field is assigned exact values and since it is 8-bit we can map it into a 256 entry table (PRO_TBL).

6.2.2 Internally Represented Filters

The observation that many rules may share same field values gives us the opportunity to save storage for rules that have common values. However, a problem arises due to this value sharing and the fact that BOS and many other APM solutions support only one Flow ID to be stored and returned during single field searches. To solve this problem we decide to keep internally represented filters where each field is assigned an internal ID during insertion. The internal ID of each field is the originally given Flow ID value. In case the value of a field was previously inserted then its internal ID is set to be equal to the existing Flow ID value, which is the first inserted. Table 6-2 illustrates how the rules presented in Table 6-1 are kept internally in B2PC.

No	Src IP ID	Dest IP ID	Src Port ID	Dest Port ID	Protocol ID	Flow ID
1	10	10	10	10	10	10
2	14	14	10	14	10	14
3	14	17	17	10	17	17
4	26	26	10	26	26	26
5	31	31	31	31	10	31
6	45	17	17	14	17	45
7	26	47	10	47	17	47
8	14	47	31	50	10	50
9	54	54	10	26	10	54
10	14	47	10	50	10	55

Table 6-2 B2PC internally represented filter set

This internal representation of filters requires storing the five 12-bit internal IDs that belong to a rule so as to be able to identify it. Keeping 4096 rules with their 5- field internal IDs can be stored in a table with 4096 entries (RULES_TBL) where indexing is done by the 12-bit Flow ID value.

A side-effect of this ID sharing is that a value of a single field cannot be deleted since many rules may depend on this internal ID. In order to cope with this problem we keep a reference count for each internal ID of each field. Inherently, we can have up to 4096 Flow IDs and therefore the same number of distinct internal IDs. Each internal ID may be referenced from at most 4096 rules and therefore we need

4096 12-bit counters for each field. In total we need 5 x 4096 12-bit counters to support incremental updates in our scheme. Accordingly, when a rule upon insertion references an internal ID, we increment the appropriate counter and when a rule is deleted we decrement the counter. The original single field value is only deleted when the related counter reaches to zero.

6.2.3 Combining Results

Given the 5 fields of a packet, B2PC has to find which of the existing rules best matches them. Single field engines provide a number of matching prefixes and the associated IDs. The IP address fields, namely Source IP and Destination IP, are prefix based and may provide at most 33 matches each; 32 possible matches for the 32 possible prefix lengths and 1 for the zero length wildcard. The port fields are also prefix based and may provide at most 17 matches; 16 possible matches for the 16 possible prefix lengths and 1 for the zero length wildcard. The protocol field is an exact value so it may provide a match on either the value itself or the wildcard; therefore we have at most 2 matches.

The internal IDs and the lengths of each matching field are gathered in certain collection points, one for every field, and they are forwarded to the mechanism that combines all the single field results. The collection points are taken the matched prefixes from the BOS modules and keep them in decreasing length order. Each collection point gives the longest prefix match first and proceeds with the less specific matches.

The results from every single field should be combined to cover all the possible permutations and then determine which of these permutations are actually valid, namely determine if such a multi field rule exists. Although the possible number of permutations could be large, the published observations indicate that the maximum number of matches in the fields is typically less than 5 and the rules that match are usually less than five. The best matching rule is the rule that has the most specific value. To accomplish this, we first check if the combination of the internal IDs that come from longest single field matches, as collection points provide it, is indeed valid and continue on checking the less specific matches. B2PC assigns priorities to the fields so as to guide the generation of permutations. The permutations are generated by keeping the current matched value of the most significant field and producing the combinations of the values coming from the least significant fields. The significance of fields in decreasing order is: Source IP, Destination IP, Source Port, Destination Port and Protocol.

Note that when all the collection points provide the same internal ID, then we surely know that this permutation belongs to our set. The same value for all the internal IDs in a permutation denotes that the values in all fields are the initially inserted ones for this specific FlowID (since that this is the way we keep internally the rules). The only thing we have to look, in this case, is whether this rule has been deleted and the values found have only been kept due to references from other rules.

The following example illustrates how the permutations are generated. Assume an incoming packet with the field values shown in Table 6-3 and the rules of Table 6-1.

Src IP	Dest IP	Src Port	Dest Port	Protocol
139.91.62.39	147.52.17.25	5000	80	TCP

Table 6-3 B2PC incoming packet example

The matching results in every collection point are stored in order from the most specific to the less specific and are shown in Table 6-4.

Src IP ID	Dest IP ID	Src Port ID	Dest Port ID	Protocol ID
14	47	10	26	10
54	54	-	31	17
26	-	-	10	-

Table 6-4 Collection points contents

The total number of possible permutations is equal to the overall product of the number of matches in every field.

$\text{Total}_{\text{perm}}$ = #Src IP IDs * #Dest IP IDs * #Src Port IDs * #Dest Prt IDs * #Proto IDs.

Hence for the matches shown in Table 6-4 the total number of permutations is:

$\text{Total}_{\text{perm}} = 3 * 2 * 1 * 3 * 2 = 36$

These 36 generated permutations are shown in Table 6-5 and the permutation that corresponds to an existing ruleset entry is shown in bold.

Perm No	Src IP ID	Dest IP ID	Src Port ID	Dest Port ID	Protocol ID
1	14	47	10	26	10
2	14	47	10	26	17
3	14	47	10	31	10
4	14	47	10	31	17
5	14	47	10	10	10
6	14	47	10	10	17
7	14	54	10	26	10
8	14	54	10	26	17
9	14	54	10	31	10
10	14	54	10	31	17
11	14	54	10	10	10
12	14	54	10	10	17
13	54	47	10	26	10
14	54	47	10	26	17
15	54	47	10	31	10
16	54	47	10	31	17
17	54	47	10	10	10
18	54	47	10	10	17
19	**54**	**54**	**10**	**26**	**10**
20	54	54	10	26	17
21	54	54	10	31	10
22	54	54	10	31	17
23	54	54	10	10	10
24	54	54	10	10	17
25	26	47	10	26	10
26	26	47	10	26	17
27	26	47	10	31	10
28	26	47	10	31	17
29	26	47	10	10	10
30	26	47	10	10	17
31	26	54	10	26	10
32	26	54	10	26	17
33	26	54	10	31	10
34	26	54	10	31	17
35	26	54	10	10	10
36	26	54	10	10	17

Table 6-5 Total possible permutations

6.2.4 Set Membership Queries with Bloom Filters

We have well studied how the rules can be decomposed into fields, inserted in the rule database, assigned an internal ID as well as how permutations are generated. The challenge we have also faced now is how to identify that a permutation belongs to our set of rules. Sequential accesses to the rule table are very slow since we may need to access them all. We need a data structure that can efficiently represent our ruleset and support quick set membership queries. Hash tables and B-Trees are widely used for this type of queries but there are also Bloom Filters [6] that have received renewed attention for network applications according to [7][44]. A Bloom filter is an efficient data structure that supports set membership queries and has tunable false positive errors as described in subsection 2.1.1.3.

We represent our rule database with a Bloom Filter that can hold 4096 rules and we have to tune the parameters of the filter so as to produce tolerable false positive rate. We have to find the optimal number of bits for the bloom filter bit-vector and the number of hash functions that set these bits. We choose the size of the bit vector to be 2^{14} bits wide and then according to the theory presented, the optimal number of hash functions is #Hash = $(2^{14} / 2^{12}) * \ln2 = 2{,}76$. So by using the optimal number of 3 hash functions we can expect false positive probability $0{,}5^3 = 0{,}125$. We decide to use 4 hash functions and further reduce the false positive probability to $0{,}5^4 = 0{,}062$, namely 6,2 %.

The bit-vector of the Bloom filter is relatively large to be kept in registers/flip-flops, and therefore we need a memory array to hold these bits. Moreover, having 4 hash functions means that we have to set (program) 4 bit positions in the bit vector and always test 4 bits. Due to the fact that the bit-vector is stored in a memory array we may require up to 4 memory accesses to locate each bit. Thus, to avoid sequential accesses and since the array is quite small and can be kept on-chip, we can increase parallelization and split this bit-vector into 4 equal sub-vectors of 4096 bits each and assign each hash function to set and test a sub-vector. This allows us to implement the accesses in parallel and decide in a single parallel memory access if the current permutation belongs to our set. Additionally, this splitting prevents the hash functions from setting the same bit.

The bits of the Bloom filter may be shared by many rules in the ruleset and thus we cannot delete a bit if other rules depend on this. The solution to this problem is given by [8] which proposes to keep counters for every bit of the Bloom filter. Hence, for the 16384 bit-vector of our bloom filter we need the same number of counters. Each counter is at most 12-bits since this is the maximum number of filters supported. Accordingly, a bit from the vector is deleted only when its counter reaches zero.

The results of the hash functions have to point to only one bit of the 4096 possible in the sub-vector and thus generate a 12-bit value. Moreover these hash functions have to use all the ID information so as to be efficient and provide discrete values for each permutation. Inherently, the IDs we use are the actual Source IP (SIP), Destination IP(DIP), Source Port(SPO), Destination Port (DPO) and Protocol (PRO). We have defined the hash functions by the use of XOR, SHIFT (>>,<<) and the reverse (REV) function according to the following formulas:

- *BLH1 = (SIP>>4)* ***xor*** *REV(DIP>>2)* ***xor*** *(SPO<<4)* ***xor*** *(DPO>>3)* ***xor*** *(PRO<<3)*
- *BLH2 = SIP* ***xor*** *(DIP<<6)* ***xor*** *(SPO>>2)* ***xor*** *REV(DPO)* ***xor*** *PRO*
- *BLH3 = (SIP<<3)* ***xor*** *REV(DIP)* ***xor*** *REV(SPO)* ***xor*** *DPO* ***xor*** *(PRO<<6)*
- *BLH4 = REV(SIP)* ***xor*** *(DIP<<3)* ***xor*** *(SPO>>3)* ***xor*** *(DPO<<1)* ***xor*** *(PRO>>2)*

The performance of these hash functions is studied and analyzed in subsection 6.3.2.

6.2.5 Flow ID Resolving

Once we have a match in a set membership query we have to determine whether it is a false positive match or in case it is not, to return the corresponding FlowID. To locate the FlowID we use a hash table of 16K entries (HSH_TBL) that shall give us the matched FlowID. Once we have the FlowID we visit RULES_TBL (subsection 6.2.2) and compare the stored IDs with the IDs of the current permutation. In case the IDs match we have found the final result, otherwise this match is a false positive and we continue with the generation and testing of the permutations.

Indexing the HSH_TBL requires a hash function and obviously this hash function may produce collisions. Resolving these collisions is trivial by using variable size blocks that hold the colliding FlowIDs. If more than one FlowIDs are stored in a specific HSH_TBL entry then we have to check the currently matched IDs with the corresponding IDs of each FlowID. The hash function uses the already hashed values of the BLH1, BLH2, BLH3 and BLH4 to indicate an entry in HSH_TBL. Its 14-bit value is defined as follows:

HSH_TBL_{index} *= (BLH1,00)* ***xor*** *(00,BLH2>>4)* ***xor*** *(00,BLH3)* ***xor*** *(00,REV(BLH4))*

The performance of this hash function is studied and analyzed in subsection 6.3.2.

6.2.6 Improving the Efficiency of Set Membership Queries

According to the generation of permutations we have to query every permutation in the Bloom filter despite the fact that a pair of source-destination prefixes or a pair of source-destination ports may not be part of the ruleset. To avoid these useless queries we can represent these pairs with additional Bloom Filters and split the membership queries problem into two sub-problems. This proposed splitting is compatible with the guidelines that were proposed in [40] and indicate that the IP address pair characterizes the actual network paths and the Port pairs characterize the network applications.

Now, we have to query the additional Bloom filters with the IP pair permutations and the Port pair permutations and if both match then we query the Bloom filter that holds the actual rules. For the Bloom filters of each pair we define two smaller bit-vectors of size 8192 with two hash functions for each one. We also split each bit-vector into two equal sub-vectors and store them is separate tables to exploit parallelism since they can also be placed on-chip. Moreover, accessing the Bloom filters of the IP pair and Port pair can be done in parallel and simultaneously perform accesses in the Rule Bloom Filter.

We have defined the hash functions for the IP and Port pairs by the use of XOR and the reverse (REV) function according to the following formulas:

- *IP_BLH1 = { SIP(6:11)* ***xor*** *DIP(0:5) , SIP(0:5)* ***xor*** *DIP(6:11) }*
- *IP_BLH2 = { SIP(0:5)* ***xor*** *DIP(6:11) , SIP(6:11)* ***xor*** *DIP(0:5) }*
- *PR_BLH1 = SPO* ***xor*** *(DPO<<2)*
- *PR_BLH2 = (SPO<<2)* ***xor*** *REV(DPO)*

The performance of these hash functions is studied and analyzed in subsection 6.3.2.

The number of generated permutations for IP and Port pairs now is significantly smaller compared to the total number of permutations and can be checked in a parallel fashion. When both queries for pairs are successful then, these pairs along with the 2 possible Protocol matches are queried in the Bloom filter that handles the actual rules. Using the example of Table 6-3 and the data shown in Table 6-4 we illustrate in Table 6-6 which queries are performed in parallel in the three Bloom filters. Queries in both IP and Port pair Bloom Filters are started together. Once matches in both pairs occur then queries in the Rule Bloom filter start, if a pair matches and the other has not yet found a match then it pauses. For the matches of pairs and rules we should consult Table 6-2. We continue by keeping the IP pair stable we test all the Port pairs given by the corresponding collection points until they finish. The queries in bold indicate the paused and stable condition of the matched permutations. The bold underlined is the matched query.

Query Number	**IP Pair Perm.**		**Port Pair Perm.**		**Rule Permutation**				
	Src IP ID	**Dest IP ID**	**Src Port ID**	**Dest Port ID**	**Src IP ID**	**Dest IP ID**	**Src Port ID**	**Dest Port ID**	**Proto ID**
1	14	47	10	26	-	-	-	-	-
2	**14**	**47**	10	31	14	47	10	26	10
3	**14**	**47**	10	10	14	47	10	26	17
4	14	54	10	26	14	47	10	10	10
5	54	47	**10**	**26**	14	47	10	10	17
6	54	54	**10**	**26**	-	-	-	-	-
7	**54**	**54**	10	31	***54***	***54***	***10***	***26***	***10***
8	**54**	**54**	10	10	54	54	10	26	17
9	26	47	10	26	54	54	10	10	10
10	**26**	**47**	**10**	**26**	54	54	10	10	17
11	**26**	**47**	10	31	26	47	10	26	10
12	**26**	**47**	10	10	26	47	10	26	17
13	26	54	10	26	26	47	10	10	10
14	-	-	-	-	26	47	10	10	17

Table 6-6 Parallel Bloom filter Queries

Breaking the problem into two gives us the opportunity to better handle the required membership tests. The IP pair first determines existing network paths in the ruleset and then the port pair determines existing network configurations. The final rule membership query then tests if those pairs match together in a rule. Searching these pairs independently distributes the queries in a more efficient manner and provides faster matches.

A general overview of the final B2PC form is presented in Figure 6-1 where all the components of the scheme are shown.

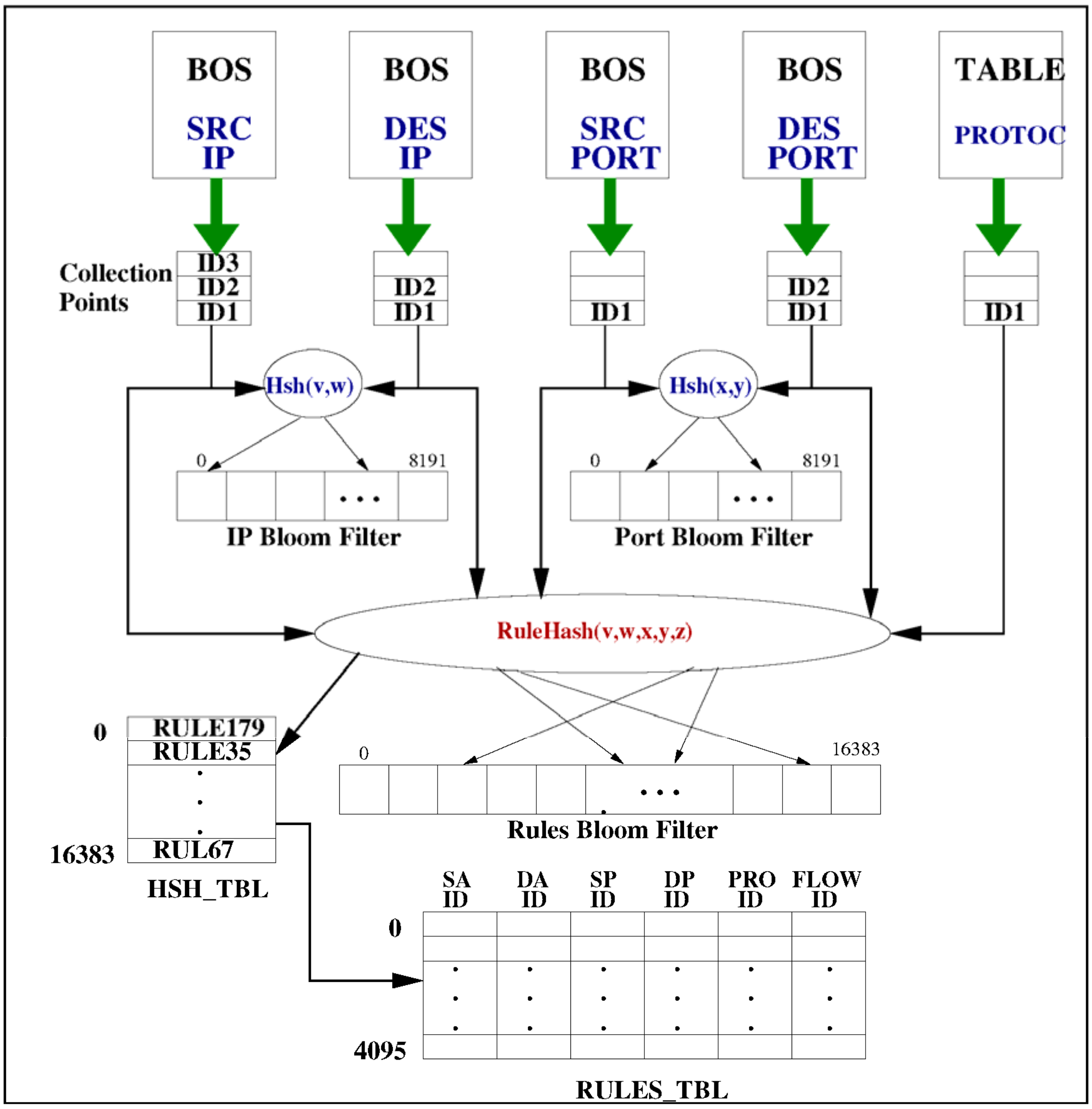


Figure 6-1 Overall view of B2PC components

6.3 Simulation Results and Performance

In this subsection we discuss simulation results based on synthetic filter sets and present our results on storage and speed. We generate 12 synthetic filter sets of various sizes with the ClassBench [42] tool and corresponding packet filter headers to test the efficiency of B2PC. We also analyze the properties of the generated filter sets and compare them with the observations found in literature. Moreover, we illustrate the efficiency of the hashing functions used by the Bloom filters and perform analysis on the observed false positives.

6.3.1 Analysis of Generated Filter Sets

We use the ClassBench tool and the seeds from real filter sets that are provided by this tool to generate sets that represent the most common filter formats: Access Control Lists (ACL), Firewall (FW) and IP Chain (IPC). We use all the real filter seeds and generate 12 synthetic filter sets of various sizes and formats. The generated filter sets and an analysis on the unique number of field values produced is shown in Table 6-7. We present the unique Source IP Addresses (SA), Destination IP Addresses (DA), Source Ports (SP), Destination Ports (DP) and the Protocols (PRO).

Filter Set Name	Set Size	Unique SA	Unique DA	Unique SP	Unique DP	Unique PRO
ACL1	712	25	316	1	96	4
ACL2	615	172	378	1	24	5
ACL3	2348	403	188	2	154	4
ACL4	2974	271	329	1	204	6
ACL5	3343	297	502	1	39	4
FW1	282	50	74	12	32	5
FW2	68	34	26	7	1	5
FW3	178	36	43	8	33	4
FW4	263	33	56	25	39	7
FW5	156	39	55	9	28	4
IPC1	1687	123	607	29	50	7
IPC2	169	24	19	3	3	4

Table 6-7 Unique field values for the generated filter sets

The filter set sizes that were generated by the tool range from 68 to 3343 and this fact is in line with observation I (section 6.1) , namely that the filter set sizes are smaller than 5000. Moreover, observation VII is also confirmed by the results of Table 6-7 since we can see that the number of unique field values found is relatively small compared to the database size. The fact that we have few unique values means that many filters share the same data as stated by observation VI. Additionally, the number of unique protocol values is ranging from 4 to 7 and is in line with the restriction professed by observation II. The port range specifications are also limited something also stated in observation III.

ClassBench provides us with a very useful packet header generator which we used to take several measurements. We generated large packet header traces and simulated B2PC in order to count the number of matches for every field. These results are shown in Table 6-8.

Set Name	Set Size	Max SA	Avg SA	Max DA	Avg DA	Max SP	Avg SP	Max DP	Avg DP	Max PRO	Avg PRO
ACL1	712	4	3,98	4	3,87	1	1	5	3,05	2	1,91
ACL2	615	5	4,92	7	5,20	1	1	4	2,36	2	1,63
ACL3	2348	6	5,92	5	4,00	2	1,00	5	2,56	2	1,95
ACL4	2974	7	6,93	7	5,30	2	1,00	6	3,02	2	1,98
ACL5	3343	3	2,99	3	1,99	1	1	4	2,01	1	1
FW1	282	4	3,75	5	4,08	3	1,63	3	1,90	2	1,91
FW2	68	3	2,76	2	1,93	2	1,75	1	1	2	1,76
FW3	178	4	3,81	4	3,00	3	1,79	3	1,96	2	1,99
FW4	263	3	2,88	4	3,90	4	2,94	3	2,61	2	1,90
FW5	156	5	4,18	4	3,82	3	1,71	3	2,04	2	1,98
IPC1	1687	4	3,99	7	5,85	4	1,20	5	2,05	2	1,89
IPC2	169	2	1,86	2	2	2	1,14	2	1,14	2	1,46

Table 6-8 Number of matched values per field

A careful look in the results reveals us that observation IV is also valid. The maximum number of either SA or DA matching a given packet ranges from 2 to 7 while the average is smaller and ranges from 1,86 to 6,93. As far as the port fields are concerned we can also see that observation IV is also valid. The maximum number of protocol field matches is bound by 2 which naturally come from the fact that we can only have an exact value or the wildcard.

6.3.2 Hashing Functions and False Positives

We incorporate many hash functions in B2PC in order to index specific bits inside the Bloom filters and to resolve the final FlowID. The most important property of a hash function used to index the Bloom filter bits is to produce several distinct values and minimize the number of different rules referencing the same bit. These hash functions are described in subsections 6.2.4 and 6.2.6. We provide an analysis of the number of bits set in each of the Bloom filters and the number of rules that reference these bits. The results are shown in Table 6-9.

Filter Set	**Set Size**	**IP Bloom Filter (8192 bits)**			**Port Bloom Filter (8192 bits)**			**Rule Bloom Filter (16384 bits)**		
		# set bits	Max Refs	Avg refs	# set Bits	Max Refs	Avg Refs	# set bits	Max refs	Avg refs
ACL1	712	911	21	1,56	192	189	7,41	2242	9	**1,27**
ACL2	615	1071	4	1,14	48	406	25,62	2153	5	**1,14**
ACL3	2348	2651	29	1,77	305	321	15,39	6566	16	**1,43**
ACL4	2974	2912	32	2,04	396	336	15,02	7847	10	**1,51**
ACL5	3343	2985	40	2,23	78	708	85,71	8468	9	**1,57**
FW1	282	418	8	1,34	107	47	5,27	1023	4	**1,10**
FW2	68	126	3	1,07	14	19	9,71	251	3	**1,08**
FW3	178	233	7	1,52	88	16	4,04	629	7	**1,13**
FW4	263	355	13	1,48	219	29	2,40	958	4	**1,09**
FW5	156	228	7	1,36	78	29	4,00	568	3	**1,09**
IPC1	1687	2406	10	1,40	164	650	20,5	5251	5	**1,28**
IPC2	169	202	8	1,67	18	111	18,77	503	7	**1,34**

Table 6-9 Number of references in Bloom Filters

The results show that our hashing functions behave quite efficiently and set many different distinct bits in the Bloom filters. As far as the Port Bloom filter is concerned, the high rate of references comes from the fact that we have a limited number of common specifications as we have observed in Table 6-7 . The Rule Bloom Filter has many bits set with a small average number of references to each bit due to the diversity of the used ID values in each rule. However, the average number of references in IP Bloom filter is a little higher than in Rule Bloom filter as an effect of the small number of unique field values compared to the size of the set (observation VII). This means that many rules in the same set share the same Source and Destination IP address specifications.

The hash functions used to index the Bloom filters and are also responsible for the number of false positives that occur. Additionally the bit-vector size of the Bloom filters influences that false positive rate. We simulate B2PC with the generated filter sets and the corresponding packet headers and we counted the false positives. The rate of observed false positives for every Bloom filter is shown in Table 6-10.

Filter Set Name	Set Size	IP Bloom False Positives (%)	Port Bloom False Positives (%)	Rule Bloom False Positives (%)
ACL1	712	0	0	0,02
ACL2	615	7,7	0	0,01
ACL3	2348	3,2	0	5,1
ACL4	2974	8,4	0	8,3
ACL5	3343	0,005	0	0,01
FW1	282	3,7	0	0
FW2	68	0	0	0
FW3	178	0	0	0
FW4	263	1,5	0	0
FW5	156	2,0	0,7	0,2
IPC1	1687	0,3	0	0,5
IPC2	169	0,1	0	0

Table 6-10 Observed false positives rate in B2PC

The observed false positives rate in B2PC is close to the theoretical 6,2% for 4096 active rules and it is very low for small filter sets. The high rate of false positives in IP and Rule Bloom filters for ACL3 and ACL4 filter sets can be justified by the fact that our hashing functions have produced higher maximum and average reference counts as shown in Table 6-9. Moreover these filter sets have an increased number of matched values per field as shown Table 6-8 and thus produce more permutations that are probed in the Bloom filters. On the other hand, ACL5, which is the largest database we generated, has a very low rate of false positives despite the fact that we observe the highest maximum and average reference counts. However, this is due to the fact that we have a small number of matched values in the all the fields as shown in Table 6-8 and therefore fewer permutations are generated and probed in the Bloom filters.

B2PC also uses a hash function to resolve the final FlowID of the matching permutation as described in subsection 6.2.5. We illustrate the collisions produced by this hash function in Table 6-11. We see that this hash function produces very few collisions and is certainly satisfactory for our scheme.

Filter Set Name	Set Size	Max Collisions	Average Collisions
ACL1	712	2	1,09
ACL2	615	2	1,03
ACL3	2348	3	1,17
ACL4	2974	3	1,19
ACL5	3343	3	1,21
FW1	282	2	1,02

FW2	68	1	1
FW3	178	1	1
FW4	263	2	1,07
FW5	156	1	1
IPC1	1687	2	1,10
IPC2	169	3	1,29

Table 6-11 B2PC hash table collisions

6.3.3 Storage Requirements

This subsection presents the storage requirements of B2PC for all the generated filter sets. To calculate the total storage for B2PC we need the storage requirements of the B2PC tables and the storage of every included BOS engine.

During simulations we find that each BOS engine has very few unique values as shown before in Table 6-7 for all the generated databases and additionally to the included static tables the memory requirements for the dynamic part of the algorithm are between 2 and 5 Kbytes. Therefore every BOS engine needs 73Kbytes for its static tables as discussed in subsection 5.3.2 and along with the included trie nodes and dynamic blocks it needs at most 78Kbytes.

For the storage requirements of B2PC we have to calculate the size of the Bloom filters, the associated counters, the counters for the IDs of each BOS engine, the protocol table (PRO_TBL), the hash table (HSH_TBL) and the rules table (RULES_TBL). For our calculations we keep the same memory configuration as in BOS, namely 36-bit wide memory words. We also assume that two counters can fit in a 36-bit word and each rule entry needs 2 memory words. Accordingly, the storage requirements for the B2PC components which are independent of the size of database are calculated in Table 6-12.

Component	Memory Words	Total Bytes
BOS ID counters	10240	46080
Bloom Filters counters	16384	73728
HSH_TBL	16384	73728
RULES_TBL	8192	36864
PRO_TBL	256	1152
Total	**51456**	**231552 (226Kb)**

Table 6-12 B2PC components memory requirements

In total we need 4 BOS engines and therefore 78 x 4 = 312 Kbytes and 226 Kbytes for B2PC components, so we finally require 538 Kbytes. These requirements are approximately the same for the entire generated filter sets since all the BOS tables are underutilized.

6.3.4 Lookup Performance

In this subsection we analyze the lookup performance of the B2PC scheme in terms of memory accesses. B2PC lookup performance is highly dependant on the APM lookup time of each BOS and on the set membership queries in the Bloom filters.

BOS supporting APM

The BOS scheme was introduced and analyzed in Chapter 5 and here we only discuss how it is used to provide matches for many prefixes so as to be used in B2PC. BOS needs several memory accesses to provide all the matches in an interval of the 32-bit address space. In every interval we check the two candidate subtries and in every matching subtrie we check all the four possible prefixes. Therefore at worst case we require one memory access to acquire the node, then another access for every subtrie node and one more memory access for every prefix. This worst case sums to 11 memory accesses and provides the FlowIDs for 8 prefixes. For the cases of single prefix subtries and single prefix roots according to the optimizations of BOS the required memory accesses are 3 and 1 respectively.

When all BOS intervals are accessed in parallel then we have the final results when lookups in the most populated interval finish, thus the number of memory accesses of the slowest trie lookup. If lookups are performed sequentially in every interval then the total number of memory accesses is equal to the sum of accesses. Note that for BOS engines that are used for the port specification we have only 3 intervals and for the IP specifications we have 5.

We simulate BOS with the generated filter sets and the packets headers and count the average and the worst case of memory accesses in every interval. We have found that the number of matching prefixes inside an interval is typically 1 and the average number of memory accessed needed to obtain the FlowID is 2,2 while the worst case observed is 6 memory accesses despite the theoretical number of 11 accesses. Therefore if we lookup the intervals in parallel we have a complete match operation every 2,2 memory accesses on average and every 6 memory accesses on the worst case. When we lookup the intervals sequentially we need 9,2 accesses on average and 25 on the worst case.

B2PC Bloom Filter Probes

The other essential factor of performance for B2PC besides the BOS matches is the number of sequential probes in the Bloom filters. We query the IP and Port pair Bloom filters in parallel and simultaneously probe the rule Bloom filter for the matched IP and Port pairs. We simulate each filter set with the corresponding packets headers and calculate the average and worst case of the sequential Bloom filter probes. The results are shown in Table 6-13.

Filter Set Name	Set Size	Max Probes	Average Probes
ACL1	712	10	2,21
ACL2	615	21	2,91
ACL3	2348	17	2,68
ACL4	2974	29	4,03
ACL5	3343	6	2,01
FW1	282	22	4,74
FW2	68	5	2,64
FW3	178	14	3,63
FW4	263	18	3,18
FW5	156	34	5,34
IPC1	1687	16	2,16
IPC2	169	4	2,07

Table 6-13 Sequential Bloom Filter probes

In the number of sequential accesses we have to add the average number of accesses in the hash table (HSH_TBL) that are equal to the collisions presented in Table 6-11 and two memory accesses to acquire the final rule from RULES_TBL. Now, the total number of memory accesses is presented in Table 6-14.

Filter Set Name	Set Size	Hash Table Accesses	Bloom Filter Accesses	Total Accesses
ACL1	712	1,09	2,21	**5,30**
ACL2	615	1,03	2,91	**5,94**
ACL3	2348	1,17	2,68	**5,85**
ACL4	2974	1,19	4,03	**7,22**
ACL5	3343	1,21	2,01	**5,22**
FW1	282	1,02	4,74	**7,76**
FW2	68	1	2,64	**5,64**
FW3	178	1	3,63	**6,63**
FW4	263	1,07	3,18	**6,25**
FW5	156	1	5,34	**8,34**

IPC1	1687	1,10	2,16	**5,26**
IPC2	169	1,29	2,07	**5,36**

Table 6-14 Average number of memory accesses for B2PC data structures

To calculate the total average number of memory accesses for B2PC we have to include the BOS lookup times. We perform parallel accesses in all the BOS engines and collect simultaneously all the results in the collection points. Each BOS engine may perform parallel or sequential accesses in its intevals. In Table 6-15 we present the final number of memory accesses needed for B2PC to produce a result.

Filter Set Name	Set Size	BOS Parallel	BOS Sequential	B2PC Accesses	B2PC with Seq. BOS	B2PC with Par. BOS
ACL1	712	2,20	9,20	5,30	**14,50**	**7,50**
ACL2	615	2,20	9,20	5,94	**15,14**	**8,14**
ACL3	2348	2,20	9,20	5,85	**15,05**	**8,05**
ACL4	2974	2,20	9,20	7,22	**16,42**	**9,42**
ACL5	3343	2,20	9,20	5,22	**14,42**	**7,42**
FW1	282	2,20	9,20	7,76	**16,96**	**9,96**
FW2	68	2,20	9,20	5,64	**14,84**	**7,84**
FW3	178	2,20	9,20	6,63	**15,83**	**8,83**
FW4	263	2,20	9,20	6,25	**15,45**	**8,45**
FW5	156	2,20	9,20	8,34	**17,54**	**10,54**
IPC1	1687	2,20	9,20	5,26	**14,46**	**7,46**
IPC2	169	2,20	9,20	5,36	**14,56**	**7,56**

Table 6-15 Final number of average memory accesses for B2PC

Lookup Perfomance and Link Speeds

According to our lookup performance we can calculate the efficiency of B2PC as a classification engine in a high speed router. To calculate the network performance we have to assumed a certain speed of the memory and a pipelined hardware implementation that can provide one memory access per cycle. The results we present assume 2 possible memory configurations:

- 200Mhz off-chip synchronous SRAM
- 400Mhz on-chip synchronous SRAM

Table 6-16 presents the network performance of B2PC counted in millions of packets per second (Mpps).

Filter Set Name	Set Size	Off-chip SRAM 200Mhz		On-Chip SRAM 400Mhz	
		B2PC with Seq. BOS (Mpps)	B2PC with Par. BOS (Mpps)	B2PC with Seq. BOS (Mpps)	B2PC with Par. BOS (Mpps)
ACL1	712	13,79	26,66	27,58	53,33
ACL2	615	13,21	24,57	26,42	49,14
ACL3	2348	13,28	24,84	26,57	49,68
ACL4	2974	12,18	21,23	24,36	42,46
ACL5	3343	13,86	26,95	27,73	53,90
FW1	282	11,79	20,08	23,58	40,16
FW2	68	13,47	25,51	26,95	51,02
FW3	178	12,63	22,65	25,26	45,30
FW4	263	12,94	23,66	25,88	47,33
FW5	156	11,40	18,97	22,80	37,95
IPC1	1687	13,81	26,80	27,66	53,61
IPC2	169	13,73	26,45	27,47	52,91

Table 6-16 Network performance of B2PC in Mpps

If we assume the worst case of classifying minimum sized IP packets (40 bytes) then the supported link speeds are shown in Table 6-17.

Filter Set Name	Set Size	Off-chip SRAM 200Mhz		On-Chip SRAM 400Mhz	
		B2PC with Seq. BOS (Gbps)	B2PC with Par. BOS (Gbps)	B2PC with Seq. BOS (Gbps)	B2PC with Par. BOS (Gbps)
ACL1	712	4,41	8,53	8,83	17,07
ACL2	615	4,23	7,86	8,45	15,72
ACL3	2348	4,25	7,95	8,50	15,90
ACL4	2974	3,90	6,79	7,80	13,59
ACL5	3343	4,44	8,63	8,88	17,25
FW1	282	3,77	6,43	7,55	12,85
FW2	68	4,31	8,16	8,63	16,33
FW3	178	4,04	7,25	8,09	14,50
FW4	263	4,14	7,57	8,28	15,15
FW5	156	3,65	6,07	7,30	12,14
IPC1	1687	4,43	8,58	8,85	17,16
IPC2	169	4,40	8,47	8,79	16,93

Table 6-17 Network performance of B2PC in Gbps

We can compare the performance of B2PC with other similar classification schemes presented in literature in terms of supported rules, storage requirements, throughput and working frequency. Our comparison is based on the results presented in the corresponding papers, where hardware implementations without TCAMs are described, and are shown in Table 6-18.

Scheme	**Working Frequency (MHz)**	**Number of Rules**	**Storage Requirements (Number of memories)**	**Throughput (Mpps)**
BV [24]	33	512	640Kb (5)	1
RFC [25]	125	1700	976 Kb (2) + 15,6 Mb (2)	30
B2PC	200	3300	540 Kb (4)	4,5

Table 6-18 Summary of Classification Schemes

Further, we introduce the metric of Mpps per Mbyte to illustrate the efficiency of classification schemes. This metric has been calculated for all the schemes of Table 6-18 by considering that all schemes work in 200MHz and extrapolating the throughput. The values of this metric for every scheme are shown in Table 6-19. We see that BV seems to be the most efficient but it only supports 512 rules. Despite RFC has the best throughput, its performance is based on greedy memory consumption as our metric shows, moreover it supports at most 1700 rules. Our scheme is very close to BV and supports more than 3300 rules with dandy efficiency.

Scheme	**Efficiency (Mpps/Mbytes)**
BV [24]	9,6
RFC [26]	2,9
B2PC	**8,65**

Table 6-19 Schemes efficiency in Mpps per Mbyte

Chapter 7

Hardware Implementation of B2PC

In this chapter we present a reference hardware implementation of the B2PC classification scheme that was described in Chapter 6. We provide a detailed description of all the internal blocks of the system and the hardware resources utilized. We also present the speed and silicon area estimations of the final design. We decided to implement the final design in an FPGA platform so as to prove the feasibility and scalability of the architecture, even when limited hardware resources are available. The FPGA platform we use is a Xilinx Virtex II Pro [32] with external Cypress ZBT SSRAMs [33].

7.1 B2PC Organization

B2PC consists of many internal blocks which are shown in Figure 7-1. The operation of the system is coordinated by the B2PC_CTRL block which receives commands and data from an external command interface (CMD_INF). Upon a reception of a command, B2PC_CTRL orders all the BOS blocks and PRO_CTRL to start in parallel their operation and feeds them with the appropriate values. The BOS blocks are responsible to work on the prefix based values and PRO_CTRL to control the protocol related table. The data structures handled by each BOS are stored in an external SSRAM and each BOS communicates with the memory handler (MEM_HDLR) and the memory controller (MEM_CTRL). The MEM_HDLR implements the dynamic memory management scheme described in section 4.4 by employing several free-lists and the MEM_CTRL is the actual low level memory interface. PRO_CTRL works on the protocol field of the network packets and stores its data on PRO_TBL which is a Single Port Block RAM (SPBRAM) of size 256x12 which is kept inside the FPGA. The results of all the BOS blocks and PRO_CTRL are given and kept to the collection points (CLPT). When all BOS operations finish then

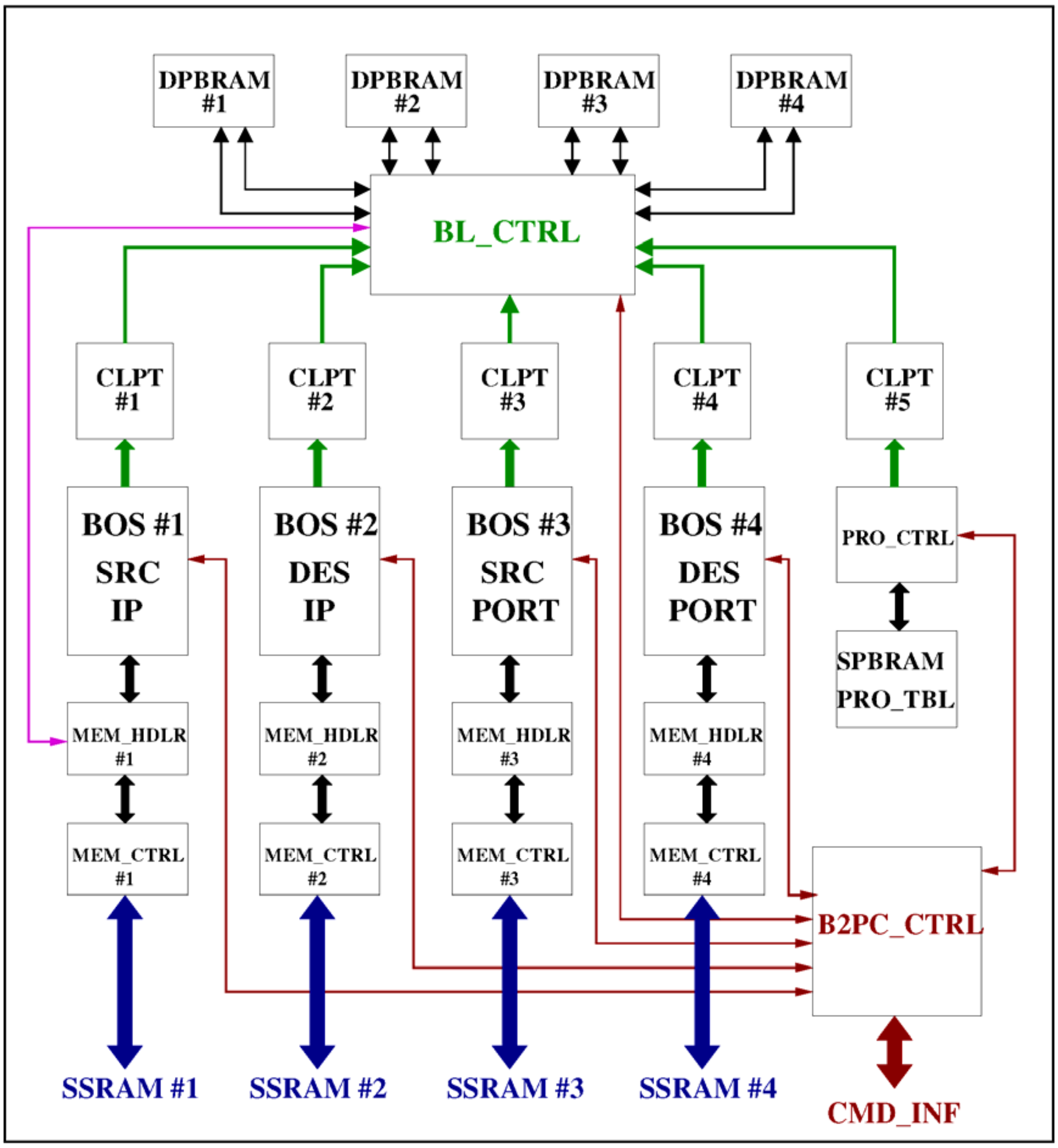


Figure 7-1 B2PC organization and block diagram

the Bloom filter control block (BL_CTRL) is instructed by B2PC_CTRL to handle all the intermediate results provided by the CLPTs. BL_CTRL generates the permutations and operates on the Bloom Filters which are stored in four on-chip Dual Port Block RAMs (DPBRAM) of size 256x32. When BL_CTRL completes the specific operation then the final result is forwarded to B2PC_CTRL and is fed to the CMD_INF. More detailed descriptions of the internal blocks are provided in the next sections.

7.2 B2PC_CTRL Block

B2PC_CTRL is the coordinator and controls all the operations that are requested by the command interface CMD_INF. This interface provides B2PC with the incoming data and also the exact operation that must be executed. The signals of the interface and their descriptions are shown in Table 7-1. This is a rather simple interface that provides all the incoming data fields together so as to be given in a single cycle to all the functional blocks in parallel.

Signal	Length	In/Out	Description
i_req	1	I	Request signal
i_opcode	2	I	Opcode for insert, lookup and delete
i_pfx1_data	32	I	Data for 1st 32-bit prefix
i_pfx1_len	5	I	Length of the 1st 32-bit prefix
i_pfx2_data	32	I	Data for 2nd 32-bit prefix
i_pfx2_len	5	I	Length of the 2nd 32-bit prefix
i_pfx3_data	16	I	Data for 1st 16-bit prefix
i_pfx3_len	4	I	Length of the 1st 16-bit prefix
i_pfx4_data	16	I	Data for 2nd 16-bit prefix
i_pfx4_len	4	I	Length of the 2nd 16-bit prefix
i_pfx5_data	8	I	Data for the protocol exact value
i_pfx5_wc	1	I	Protocol wildcarded or not
o_ack	1	O	Acknowledgement
o_flow_id	12	O	The returned flow ID

Table 7-1 Command Interface Signals

B2PC_CTRL involves a finite state machine (FSM) to handle all the possible cases and generates request signals to all the other blocks. The block receives a request for a command defined by i_opcode and latches all the incoming data to registers. The following opcodes are defined:

- *2'b00 : Lookup*
- *2'b01 : Insert*
- *2'b10 : Delete*
- *2'b11 : Reserved*

Request signals are generated to all BOS and PRO_CTRL blocks with the incoming opcode and the appropriate data and lengths. When all these sub-blocks finish then it sends a request to BL_CTRL so as to start the generation of permutations and operate on the Bloom filters. When BL_CTRL finishes it returns the concluding FlowID which is then returned to CMD_INF along with the o_ack signal.

7.3 BOS Block

BOS needs several internal blocks so as to handle the operations of the BOS scheme as described in Chapter 5. The internal organization of BOS is depicted in Figure 7-2. The BOS block receives commands from B2PC_CTRL and informs it when it completes an operation. It also provides the CLPT with the matched prefixes information, namely the FlowID and the length of each matched prefix. Moreover the required memory communication is done over the MEM_HDLR block where requests for read, write, memory allocation and deallocation are given.

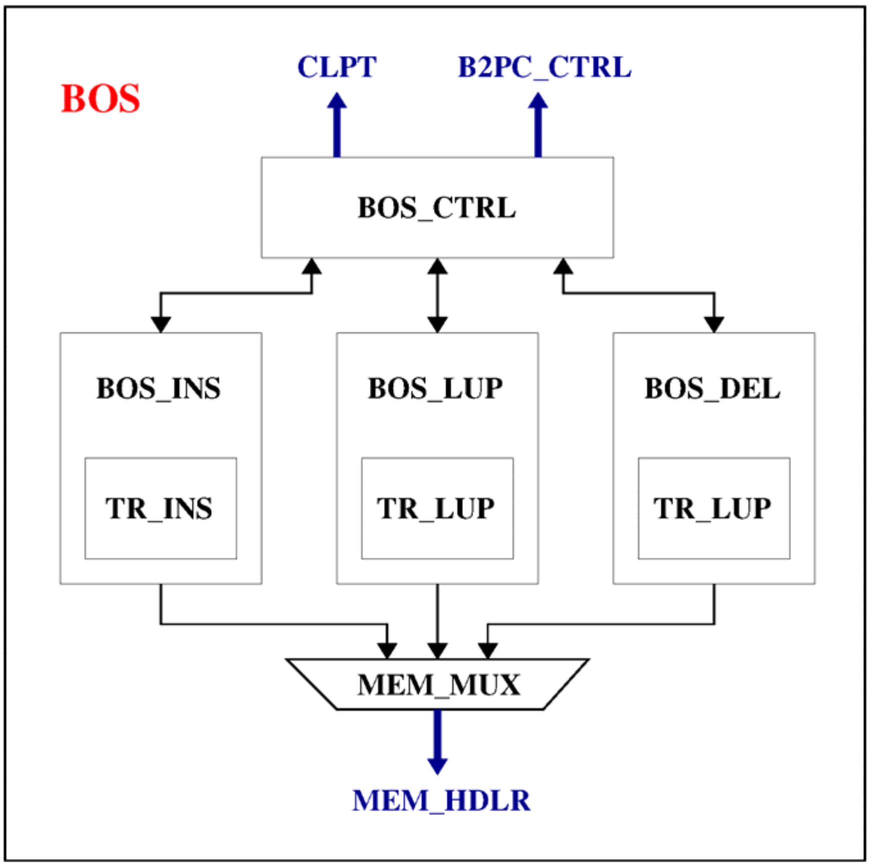


Figure 7-2 BOS internal organization

Memory Organization and Nodes

The current BOS implementation is based on sequential accesses to the BOS tables TBL0, TBL8 and TBL16 because we prefer for cost purposes not to have separate memories and all the tables of the same BOS engine are stored in the same SSRAM. The memory word we have is 36-bits and we use at most 32K words which are sufficient as presented in subsection 6.3.3. The organization of the tables in the memory and the pool of free addresses for the dynamic memory management scheme is shown in Figure 7-3. The first 16K words are used for TBL16, the next 256 words are for TBL8 and a single memory word for TBL0. The remaining 16127 memory words are used by the memory handler (MEM_HDLR) to provide dynamic allocation and deallocation of the required memory blocks.

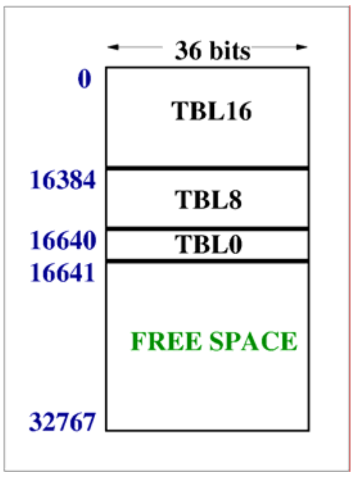


Figure 7-3 BOS Memory Organization

BOS internally defines some data structures for the nodes that are used, namely the basic nodes, the root nodes, the trie nodes and the prefix nodes. The formats of the nodes we defined for BOS are shown in Figure 7-4.

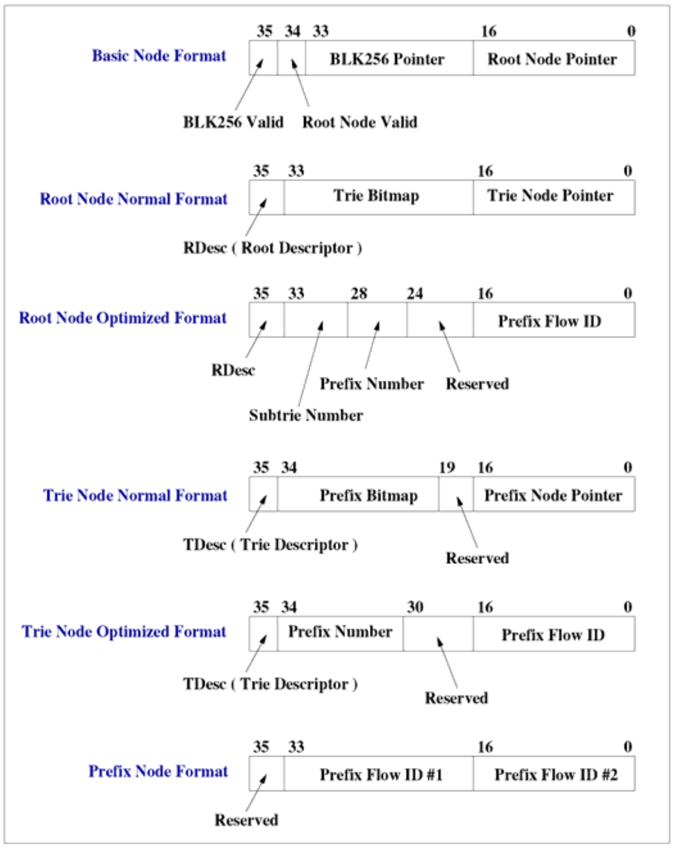


Figure 7-4 BOS nodes format

The basic nodes are stored in TBL16 or within BLK256s and provide the linking from 16 length nodes to 24 and 32 length nodes according to the TBL24 and TBL32 optimization discussed in subsection 5.2.4. The fields of the basic nodes are the following:

- ***BLK256 Valid*** *: Indicates whether there is a link in the next stride and if the data contained in the* ***BLK256 Pointer*** *field are valid (1-bit).*
- ***Root Node Valid*** *: Indicates whether the data contained in the* ***Root Node Pointer*** *field are valid (1-bit).*
- ***BLK256 Pointer*** *: The address of the linked BLK256 (17-bits).*
- ***Root Node Pointer*** *: The address of the root node (17-bits).*

The root nodes are stored in TBL0 and TBL8 and are linked in the Root Node Pointer fields of the basic nodes existing in TBL16 and BLK256s. There are 2 types of root nodes, the normal and the optimized node that implements the trie node optimization discussed in subsection 5.2.2. The fields of the normal root nodes are the following:

- ***RDesc*** *: Root descriptor that indicates if this is a normal or an optimized node. The value 1 indicates optimized node. Two values are reserved for future use (2-bits).*
- ***Trie Bitmap*** *: The bitmap that indicates which subtries are active (17-bits).*
- ***Trie Node Pointer*** *: The address of the trie node (17-bits).*

The fields of the optimized root nodes are the following:

- ***RDesc*** *: Root descriptor that indicates if this is a normal or an optimized node. The value 0 indicates an empty node, value 1 indicates an optimized node and value 2 indicates a normal node (2-bits).*
- ***Subtrie Number*** *: The number of the single active subtrie (5-bits).*
- ***Prefix Number*** *: The number of the active prefix (4-bits).*
- ***Prefix Flow ID*** *: The corresponding FlowID of the prefix (17-bits).*

The trie nodes are used to keep the prefix bitmap and the pointer to the associated data. There are two formats for the trie nodes, the normal and the optimized that exploits the prefix node optimization discussed in subsection 5.2.1. The fields of the normal trie nodes are the following:

- ***TDesc*** *: Trie descriptor that indicates if this is a normal or an optimized node. The value 1 indicates optimized node (1-bit).*
- ***Prefix Bitmap*** *: The bitmap that indicates which prefixes are active (15-bits).*
- ***Prefix Node Pointer*** *: The address of the prefix node (17-bits).*

The fields of the optimized trie nodes are the following:

- ***TDesc*** *: Trie descriptor that indicates if this is a normal or an optimized node. The value 1 indicates optimized node (1-bit).*
- ***Prefix Number*** *: The number of the active prefix (4-bits).*
- ***Prefix Flow ID*** *: The corresponding FlowID of the prefix (17-bits).*

Note that a single root may have many trie nodes depending on the number of active subtries. When there is more than one subtrie then these trie nodes are kept in blocks of adjacent memory words in sorted order. Sorting is performed by putting a given subtrie node in the position of the block which is equal to the number of set bits in the Trie Bitmap before the correlated subtrie bit, namely by counting the number of active subtries that have number smaller than the current subtrie.

The prefix nodes are used to keep the associated prefix data for subtrie nodes that have more than one active prefixes. They have two fields that keep the Flow IDs in sorted order. The fields of the prefix nodes are the following :

- ***Prefix Flow ID #1*** *: The FlowID of the 1st saved prefix (17-bits).*
- ***Prefix Flow ID #2*** *: The FlowID of the 2nd saved prefix (17-bits).*

Also, note that a single trie may have many prefix nodes depending on the number of active prefixes. When there is more than one prefix then these prefix nodes are kept in blocks of adjacent memory words in sorted order. Sorting is performed the same way as in subtrie nodes but now the Prefix Bitmap is used.

7.3.1 BOS_CTRL

BOS_CTRL is responsible for managing the operations of the block and involves an FSM to handle the requests for the insert, lookup and delete operations. For each operation there is a sub-block responsible to complete it. BOS_INS is responsible for the inserts, BOS_LUP for the lookups and BOS_DEL for the deletes. Upon a reception of a command BOS_CTRL generates a request to the appropriate block and sets the appropriate select value in MEM_MUX so as to output a specific block's requests to the memory handler.

7.3.2 BOS_INS

BOS_INS sub-block handles all the prefix insertions in the appropriate table or BLK256 together with TR_INS. This block also provides the final internal ID for B2PC by checking if the prefix already exists. The functional aim of this sub-block inside BOS is to provide the suitable root node pointer to TR_INS which implements the final insertion in the trie data structures. Depending on the incoming prefix length BOS_INS accesses the tables and reads the root node pointer. The insert procedure has an FSM to handle the following series of actions:

- *If the incoming prefix length is shorter that 8 then it provides TR_INS with the address of the single memory word* ***TBL0****.*
- *If the incoming prefix length is shorter that 16 then it provides TR_INS with the address of the memory word inside* ***TBL8*** *that is defined by the first 8-bits of the prefix.*
- *If the incoming prefix length is shorter that 24 then it accesses the memory word of* ***TBL16*** *defined by the first 16-bits of the prefix and the checks the* ***Root Node Valid*** *flag (Basic Node Format).*
  - *If this flag is set then gives the* ***Root Node Pointer*** *address to TR_INS*
  - *Otherwise, it requests allocation of a single word from MEM_HDLR and sets the* ***Root Node Valid*** *bit and the* ***Root Node Pointer*** *with the allocated address. Moreover, the allocated address in forwarded to TR_INS.*
- *If the incoming prefix length is shorter that 32 then it accesses the memory word of* ***TBL16*** *defined by the first 16-bits of the prefix and the checks the* ***BLK256 Valid*** *flag (Basic Node Format).*
  - *If this flag is set then it accesses the address shown by* ***BLK256 Pointer*** *in the offset defined by the active 8 LSB of the prefix and checks the* ***Root Node Valid*** *flag.*
    - *If this flag is set then gives the* ***Root Node Pointer*** *address to TR_INS*
    - *Otherwise, it requests allocation of a single word from MEM_HDLR and sets the* ***Root Node Valid*** *bit and the* ***Root Node Pointer*** *with the allocated address. Moreover, the allocated address in forwarded to TR_INS.*
  - *If the* ***BLK256 Valid*** *flag is not set then it requests allocation of a 256 word block and a single memory word. The* ***BLK256 Valid*** *flag and the* ***BLK256 Pointer*** *are set in the TBL16 entry and the R****oot Node Valid*** *flag and* ***Root Node Pointer*** *are set inside the newly allocated block in the address defined by the active 8 LSB of the prefix. The address of the single memory word is given to TR_INS.*
- *If the incoming prefix has length 32 then it accesses the memory word of* ***TBL16*** *defined by the first 16-bits of the prefix and the checks the* ***BLK256 Valid*** *flag (Basic Node Format).*
  - *If this flag is set then it accesses the address shown by* ***BLK256 Pointer*** *in the offset defined by the bits 16-23 of the prefix and checks the new* ***BLK256 Valid*** *flag. If this flag is set then it accesses the address shown by* ***BLK256 Pointer*** *in the offset defined by the active 8 LSB of the prefix and checks the* ***Root Node Valid*** *flag.*

- *If this flag is set then we have found the FlowID of the prefix and get it from the root node pointer field.*
- *If this is not set then we set it and put the incoming FlowID in the **Root Node Pointer** field.*

o *If the BLK256 valid flag is not set then it requests allocation of two 256 word blocks. **The BLK256 Valid** flag and the **BLK256 Pointer** are set in the **TBL16** entry and next the **BLK256 Valid** flag and **BLK256 Pointer** are set inside the newly allocated block in the address defined by the bits 16-23 of the prefix. Inside the second BLK256 in the offset defined by the 8 LSB of the prefix it sets the **Root Node Valid** flag and puts the FlowID in the **Root Node Pointer** field.*

7.3.3 TR_INS

TR_INS sub-block handles the actual prefix insertions in the appropriate root nodes. This sub-block also provides the final internal ID for B2PC by checking if the prefix already exists. The aim of this sub-block is to generate or update the existing root nodes in order to incorporate the incoming prefix. TR_INS works on root nodes, trie nodes and prefix nodes.

For the prefix to be inserted, TR_INS has first to find the subtrie number and the prefix number in order to work on the bitmaps. The formulas for generating these numbers have been discussed in subsection 5.1.4. Once these numbers have been generated then an FSM examines the contents of the root node and follows the steps shown below:

- *If the **RDesc** field is 0 we have an empty node and we proceed to create and node with optimized format. Hence, we set **RDesc** to 1, set the **Subtrie Number** and the **Prefix number** with the generated values and put the incoming FlowID in the Prefix Flow ID field.*
- *If the **RDesc** field is 1 we have an optimized node and have to proceed to generate further nodes.*

  o *If the existing **Subtrie number** matches with the generated one then we allocate memory for a trie node and a prefix node. We generate a **Trie Bitmap** with the appropriate bit set, link the trie node in the **Trie Node Pointer** field and write it as a new root node, then we generate a **Prefix Bitmap** and link the prefix node in the **Prefix Node Pointer** of the trie node and finally put the **Prefix Flow ID**s inside the prefix node in ascending order.*

  o *If the existing **Subtrie Number** does not match with the generated one then we allocate memory for two trie nodes of optimized format. We generate a **Trie Bitmap** with the appropriate bits set, link the trie nodes in the **Trie node Pointer** field and write it as* *a new root node, then we write in each trie node **TDesc** with 1 and fill the **Prefix Number** and **Prefix Flow ID** fields.*

- *If the **RDesc** field is 2 we have a normal node and have to examine if the bit number indicated by the generated subtrie number is set in the existing **Trie Bitmap.***
    - *If this specific bit is set then we follow the **Trie Node Pointer.** If it has normal format, namely **TDesc** is 0, then we check the **Prefix Bitmap.***
        - *If the bit indicated by prefix number is set the we read the **Prefix Node Pointer** in the appropriate offset and return the new internal Flow ID.*
        - *Otherwise, we set the specific bit in the **Prefix Bitmap**, allocate space for the new prefix node and write the final **Prefix Flow ID** in the proper position.*
    - *If **TDesc** is 1 then we have an optimized prefix node and we allocate memory for the prefix node, generate the **Prefix Bitmap**, link the prefix node and write the **Prefix Flow ID**s sorted in the node.*
    - *If the bit is not set in the **Subtrie Bitmap** then we set it and allocate space for the new trie node that has optimized format. We place the trie node in the proper position so as the trie nodes to be sorted and write the appropriate data. We write **TDesc** with 1, write the **Prefix Number** and the **Prefix Flow ID**.*

7.3.4 BOS_LUP

BOS_LUP sub-block handles the prefix lookups and implements the All Prefix Match (APM) algorithm. For cost purposes we have implemented BOS with sequential accesses in the tables and this sub-block visits all the tables one by one and follows the links to BLK256 to find valid root pointers. Once BOS_LUP finds an active root node then it provides the address of the root node to TR_LUP which makes the actual lookup inside the trie nodes. The insert procedure has an FSM to handle the following steps:

- *Provide TR_LUP with the address of **TBL0**.*
- *Provides TR_INS with the address of the memory word inside **TBL8** that is defined by the first 8-bits of the prefix.*
- *Access **TBL16** and*
    - *if the **Root Valid** flag if set then it gives the **Root Node Pointer** to TR_LUP.*
    - *If the **BLK256 Valid** flag is set we follow the **BLK256 Pointer** and access in the offset specified by the bits 16-23 of the prefix. If **Root Valid** is set there we provide TR_LUP with the existing **Root Node Pointer**. If **BLK256 Valid** flag is also set then we follow the*

__BLK256 Pointer__ and access in the offset specified by the 8 LSB of the prefix. If __Root Valid__ is set there then we return the Flow ID stored in the __Root Node Pointer__.

7.3.5 TR_LUP

TR_LUP sub-block handles the actual prefix lookups in the appropriate root nodes. This sub-block provides the matching IDs for B2PC by checking if the prefixes match. TR_LUP works on root nodes, trie nodes and prefix nodes. For a given value to be matched, TR_INS has first to find the candidate subtries and the prefixes numbers in order to work on the bitmaps. The formulas for producing these numbers have been discussed in subsection 5.1.4. Once these numbers have been produced then an FSM examines the contents of the root node and follows the steps shown below:

- *If the __RDesc__ field is 0 we have an empty node and therefore no matches.*
- *If the __RDesc__ field is 1 we have an optimized root node and have to check if the __Subtrie Number__ matches with one of the generated.*
    - *If it matches then we check the __Prefix Number__ with the four candidate generated prefixes.*
        - *If it matches then we return the __Prefix Flow ID__.*
        - *Otherwise, we have no match.*
    - *Otherwise, we have no match.*
- *If the __RDesc__ field is 2 we have a normal root node and have to examine the __Subtrie Bitmap__ for the two specific bits set.*
    - *For those subtries that the bit in the __Subtrie Bitmap__ is set we follow the __Trie Node Pointer.__*
        - *If __TDesc__ is 1 the we check the __Prefix Number__ with the four candidate and if one matches then we return the __Prefix Flow ID.__*
            - *If __TDesc__ is 0 then we examine the __Prefix Bitmap__ for the four specific bits indicated by the candidate prefix numbers.*
            - *For every specific bit that is set the we follow the __Prefix Node Pointer__ in the appropriate offset and return the __Prefix Flow ID.__*
    - *If neither bits are set we have no match.*

7.3.6 BOS_DEL

BOS_DEL sub-block handles all the prefix deletions in the appropriate tables or BLK256 together with TR_DEL. In terms of functionallity this sub-block provides the suitable root node pointer to TR_DEL which handles the final deletion in the trie data structures. Depending on the incoming prefix length BOS_DEL accesses the tables and reads the root node pointer. The delete procedure has an FSM to handle the following series of actions:

- *If the incoming prefix length is shorter that 8 then it provides TR_DEL with the address of the single memory word **TBL0**.*
- *If the incoming prefix length is shorter that 16 then it provides TR_DEL with the address of the memory word inside **TBL8** that is defined by the first 8-bits of the prefix.*
- *If the incoming prefix length is shorter that 24 then it accesses the memory word of **TBL16** defined by the first 16-bits of the prefix and the checks the **Root Node Valid** flag (Basic Node Format).*
    - *If this flag is set then gives the **Root Node Pointer** address to TR_DEL.*
    - *Otherwise, delete fails.*
- *If the incoming prefix length is shorter that 32 then it accesses the memory word of **TBL16** defined by the first 16-bits of the prefix and the checks the **BLK256 Valid** flag (Basic Node Format).*
    - *If this flag is set then it accesses the address shown by **BLK256 Pointer** in the offset defined by the active 8 LSB of the prefix and checks the **Root Node Valid** flag.*
        - *If this flag is set then it gives the **Root Node Pointer** address to TR_DEL*
        - *If the flag is zero, delete fails.*
- *If the incoming prefix has length 32 then it accesses the memory word of **TBL16** defined by the first 16-bits of the prefix and the checks the **BLK256 Valid** flag (Basic Node Format).*
    - *If this flag is set then it accesses the address shown by **BLK256 Pointer** in the offset defined by the bits 16-23 of the prefix and checks the new **BLK256 Valid** flag.*
        - *If this flag is set then it accesses the address shown by **BLK256 Pointer** in the offset defined by the active 8 LSB of the prefix and checks the **Root Node Valid** flag. If this flag is set then we have to reset the **Root Node Valid** flag and set the contents of the **Root Node Pointer** field to zero. Otherwise, delete fails.*
        - *Otherwise, delete fails.*
    - *If the flag is zero, delete fails.*

7.3.7 TR_DEL

TR_DEL sub-block handles the actual prefix deletions in the appropriate root nodes. The aim of this sub-block is to deallocate or update the existing root nodes in order to remove the given prefix. TR_DEL works on root nodes, trie nodes and prefix nodes.

For the prefix to be deleted, TR_DEL has first to find the subtrie number and the prefix number in order to work on the bitmaps. The formulas for generating these numbers have been discussed in subsection 5.1.4. Once these numbers have been generated then an FSM examines the contents of the root node and follows the steps shown below:

- *If the **RDesc** field is 0 we have an empty node and delete fails.*
- *If the **RDesc** field is 1 we have an optimized node and have to examine the fields.*
  - *If both the existing **Subtrie number** and the **Prefix Number** match then we put zeros in the word.*
  - *Otherwise, delete fails.*
- *If the **RDesc** field is 2 we have a normal node and have to examine if the bit number indicated by the generated subtrie number is set in the existing **Trie Bitmap.***
  - *If this specific bit is set then we follow the **Trie Node Pointer.** If it has normal format, namely **TDesc** is 0, then we check the **Prefix Bitmap.***
    - *If the bit indicated by prefix number is set, we reset it and deallocate its space for the memory of the **Prefix Node Pointer** by keeping the sorted order.*
    - *If **TDesc** is 1 and **Prefix Number** matches then we deallocate the trie node, rearrange the trie nodes and reset the bit in the **Trie Bitmap** of the root node.*
    - *Otherwise, delete fails.*
  - *Otherwise, delete fails.*

7.4 PRO_CTL

PRO_CTL block is responsible to insert, delete and find the two possible matches of the protocol field. When instructed, its FSM accesses PRO_TBL in the memory address 0 to find a match for the wildcard specification and the memory address defined by the incoming 8-bit protocol field value. The matches are sent to the appropriate collection point (CLPT) and a finish signal is asserted to B2PC_CTL.

7.5 CLPT

CLPT is the implementation of the collection point and gathers the results from every single field search. CLPT keeps at most 33 FlowIDs (subsection 6.2.3) internally in a memory, sorted in descending order and provides them to BL_CTRL. It provides the matching IDs in a show-ahead fashion and signals BL_CTRL when all of them have been read. CLPT does not dequeue the IDs upon reading and when all of them have been read, it starts providing them from the beginning. Basically, CLPT is a show-ahead circular buffer.

7.6 BL_CTRL

BL_CTRL handles all the Bloom Filters, generates the permutations to be tested and provides the final filter match. It performs the set membership queries and resolves the FlowID by visiting the hash and rule tables. BL_CTRL involves many sub-blocks to accomplish several operations and its internal organization is depicted in Figure 7-5. It reads the matched IDs from the 5 collections points with the proper sequence in order to generate the permutations based on the descriptions of subsections 6.2.3 and 6.2.6.

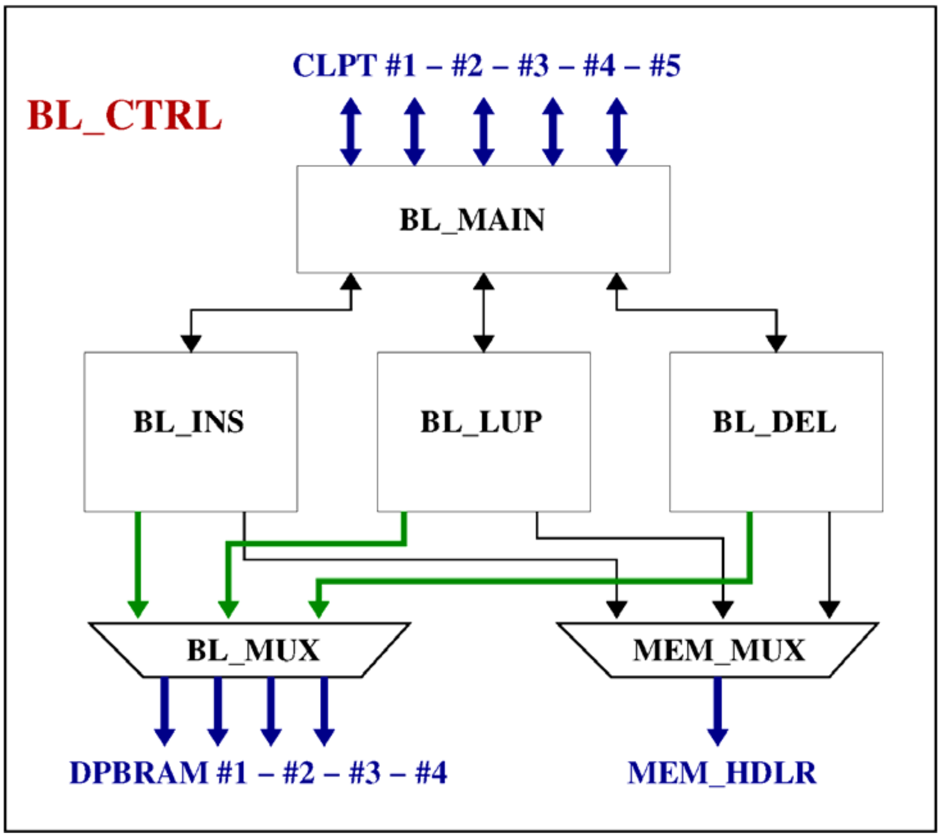


Figure 7-5 BL_CTRL Internal organization

Memory organization of Bloom Filters and Static Tables

The implementation of B2PC requires the Bloom Filters of the IP pair, the Port pair and the Rules to be stored in memories inside the FPGA. According to the design of the Bloom filters we partition the bit-vector of Rule Bloom filter into four equal parts to access them in parallel. Moreover, the bit-vectors of IP and Port Bloom filters are also split into two equal parts. Additionally all the existing bit-subvectors need to be accessed in parallel. To achieve this parallelization and store these Bloom Filters we keep on-chip four Dual Port Block RAMs (DPBRAM) of size 256x32. Figure 7-6 illustrates how the bit-vectors of the Bloom filters are organized inside the memories.

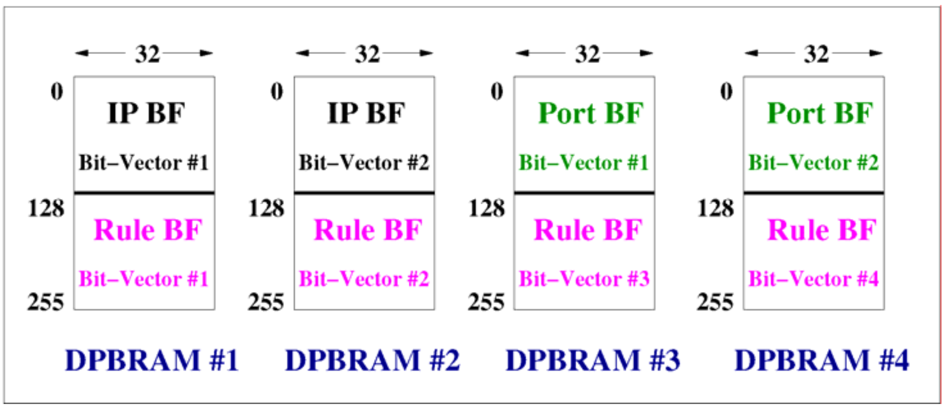


Figure 7-6 Bloom filter memory partitioning

All the memories are split into two parts and thus we have 128x32 = 4096 bits for every bit-vector of the Bloom filters as it is required. The first part of each memory is accessed by the first port and the second by the second port so as to have parallel accesses. The hash functions provide a 12-bit value to indicate a specific bit inside the bit-vector. The first 7-bits of the value can be used to identify one of the 128 memory words of each bit-vector and the last 5-bits define a specific bit from the 32 of each word.

B2PC has also some static tables to hold the internally represented rules (RULES_TBL) , a hash table (HSH_TBL) to resolve the matched permutations and several tables with counters to keep the reference counts of the BOS IDs and the Bloom filters bit references. This kind of information is only known to BL_CTRL block which knows the final internal IDs, the Bloom filter bits and calculates their reference counts. This information is stored in the first external SSRAM and BL_CTRL has an interface with the related memory handler. The first 32K memory words of the first SSRAM are allocated for BOS #1 and therefore BL_CTRL is assigned addresses below 32K. The additional memory words used by B2PC have been discussed in subsections 6.2.2 and 6.2.4 and their organization inside the SSRAM is shown in Figure 7-7.

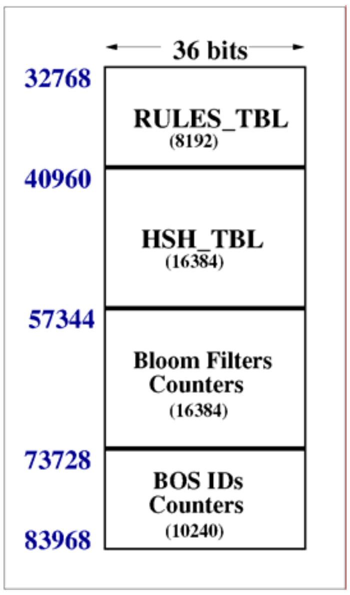


Figure 7-7 Organization of Static Tables

7.6.1 BL_MAIN

BL_MAIN sub-block responsible for the central operation of BL_CTRL and involves an FSM to handle the requests for the insert, lookup and delete operations defined by the same opcodes as in B2PC. For each operation there is a sub-block responsible to complete it. BL_INS is responsible for the inserts, BL_LUP for the lookups and BL_DEL for the deletes. Upon a reception of a command BL_MAIN generates a request to the appropriate block and sets the BL_MUX and MEM_MUX to output a specific block's requests to the Bloom filters' memory and the memory handler.

7.6.2 BL_INS

BL_INS sub-block has an FSM to handle all the programming of the Bloom filters, save the internal rule representation in the RULES_TBL, set HSH_TBL values and update the reference counters. When all single field values are inserted in BOS engines and protocol table then BL_INS calculates the hashing functions on the IDs and the following steps are performed:

- *the IDs of each field of the rule are then inserted in the RULES_TBL,*
- *the appropriate ID counters are incremented,*
- *HSH_TBL values are set,*
- *Bloom Filters' bits are set,*
- *Bloom Filters' counters are incremented.*

7.6.3 BL_LUP

BL_LUP sub-block performs all the set membership queries in the Bloom Filters and generates all the permutations. BL_LUP involves four FSM's to achieve the parallel accesses in the Bloom Filters. We have one FSM for the IP pair queries, one for the Port pair queries, one for the Rule queries and the main FSM that resolves the final FlowID.

The first FSM performs the queries in the IP pair Bloom filter. The permutations are generated by reading the Source IP ID from the appropriate collection point and while keeping this value steady, we read sequentially all the values from the Destination IP collection point. When these finish then we read the next value from the Source IP collection point and start from the beginning of Destination IP IDs. For every permutation the IP pair Bloom filter is probed and when a match occurs then the couple of IDs is sent to the FSM that handles the Rule Bloom filter queries. The Port pair Bloom filter queries and permutations are performed in the same way as the IP pair but now the Source and Destination Port collection points are read.

The FSM that handles the Rule Bloom Filter queries waits for matches from both IP and Port pair FSM's and when both provide values then these values along with the two possible IDs of the Protocol field are probed in the Rule Bloom Filter. If a query is successful then the 5 IDs are sent to the main FSM to resolve the final FlowID or indicate a false positive.

The main FSM uses the 5 ID values that matched and visits the hash table to get the possible Rule FlowID. The values that are found in the specific memory word of HSH_TBL indicate the possible FlowIDs. For the found FlowIDs then this FSM visits RULES_TBL and checks if the IDs values located there match with the 5 provided IDs. If all match then we have finally found a match and return this FlowID, otherwise if we have no match then it is a false positive and the FSMs that probe the Bloom Filters continue looking for matches.

7.6.4 BL_DEL

BL_DEL sub-block has an FSM to handle the updates of the Bloom filters, to remove rules from RULES_TBL, to reset HSH_TBL values and to update the reference counters. When a FlowID is to be deleted, all the IDs from RULES_TBL are read, the Bloom hashing functions are calculated and the following steps are performed:

- *all the IDs counters are decremented and if a counter is decremented to zero then a delete command is sent to the appropriate BOS engine to be removed.*
- *all the Bloom Filters' bit counters are decremented and if a counter is decremented to zero then the corresponding Bloom bit is cleared.*
- *the related HSH_TBL value is cleared.*
- *the entry in RULES_TBL is cleared.*

7.7 MEM_HDLR and MEM_CTRL

The MEM_HDLR sub-block provides the dynamic memory management in our system and supports the variable size blocks. MEM_HDLR is the intermediate layer between the sub-blocks and the memory controller (MEM_CTRL) and supports requests for allocation and deallocation of variable size blocks. Requests for reads or writes in the memory are immediately forwarded to the memory controller MEM_CTRL.

The design of MEM_HDLR is already described in section 4.4 and here we have the same configuration but we support many different sizes of memory word blocks. We support blocks of 1,2,4,8 and 18 words. Moreover we provide the ability to allocate the big 256-word memory blocks. The design of MEM_CTRL is already described in subsection 4.5 and here we use the same design.

7.8 Implementation Analysis

In this subsection we provide an analysis of the block latencies and an estimation of the implementation cost for the reference design.

7.8.1 Latency Analysis

We calculate the minimum and the maximum number of clock cycles required by each block to complete its operation. Many of the blocks have variable latencies which depend on the access patterns and the data stored in the data structures. Moreover, the blocks that access the external SSRAMs for the stored data structures have to also suffer the latency of our memory controller. In Table 4-2 we present the latency per block of B2PC.

Block Name	**Min Latency (clock cycles)**	**Max Latency (clock cycles)**
B2PC_CTRL	1	-
BOS_CTRL	1	-
BOS_INS	1	7
BOS_LUP	4	8
BOS_DEL	1	7
TR_INS	3	23
TR_LUP	2	22
TR_DEL	3	22
PRO_CTL	2	2
CLPT	0	0
BL_MAIN	1	-
BL_INS	38	38
BL_LUP	8	-
BL_DEL	35	35
MEM_HDLR	0	3
MEM_CTRL	1	2

Table 7-2 B2PC Blocks Latencies

The fact that the memory controller has latency 2 cycles (section 4.5) for a read operation in the external SSRAM significantly affects the performance of the blocks that perform sequential accesses to the memory. Insert, lookup and delete operations are high depending on the read data to decide the address of the next memory access and thus the 2 cycle latency of the memory controller is continuously introduced. Additionally, some blocks like BL_LUP have unspecified maximum latency since they perform iterative operations on the collected data and depend on every case specifically. Note also that BOS_LUP occupies TR_LUP at most 4 times therefore the total latency of a BOS lookup requires at minimum 6 clock cycles and at maximum 92 clock cycles. According to the number of memory accesses we calculated in subsection 6.3.4 we need for a BOS lookup 18,4 clock cycles on average and for a Bloom lookup (BL_LUP) 9,4 clock cycles. In total the average lookup time is approximately 28 clock cycles.

7.8.2 Hardware Cost Analysis

We have used VHDL to describe the design and the results presented are the reports from the synthesis tools. We have synthesized the design using the Synopsys Design Compiler[35] which is the most widely used synthesis tool. We have used UMCs 0.13μm technology library to estimate the area and the frequency of the design. Moreover, we used the XilinX ISE tool to implement and port the design in the FPGA.

The synthesis tool for the ASIC flow indicates that the maximum working frequency of our design is 200Mhz.Using the synthesis tool we calculated the number of flip-flops contained in our design and we present them per high level block in Table 7-3. Since the final design has many instances of the same blocks, we also calculate the total number of flip-flops.

Block	Block Description	Number of Flip-Flops
BOS	BOS engine	624
PRO_CTL	Control of Protocol Table	14
CLPT	Collection Point	19
BL_CTRL	Bloom Control	191
MEM_HDLR	Memory Handler	662
MEM_CTRL	Memory Controller	43
B2PC_CTRL	Control Block of B2PC	219
Total		**5835**

Table 7-3 Flip-Flop count per block

The area of the total design and the equivalent gate count is presented in Table 7-4. The equivalent gate count is calculated by considering how many 2-input NANDs can be accommodated in this area.

Components	Area (mm^2)	Equivalent NAND Gates
Combinatorial	0,595	115K
Non-Combinatorial	0,250	48K
Memories	0,456	88K
Total		**251K**

Table 7-4 B2PC area and gate count

The ISE tool of the Xilinx FPGA flow shows that the maximum working frequency of our design is 75Mhz. The tool reports the occupied resources after a full back-end FPGA flow while occupying optimizations to remove redundant logic or replicate logic to improve speed. The final results are shown in Table 7-5.

Resource	**Resource count**
Used 4 input LUTs	30867
Slice Flip Flops	5390

Table 7-5 FPGA resource allocation

7.8.3 B2PC Hardware Performance

Considering that we have a 75MHz clock, the external memories work on the same frequency and the average lookup time is 28 clock cycles then, the FPGA prototype design of B2PC supports 2,7 Mpps.

Chapter 8

Contributions and Future Work

8.1 Summary of Contributions

We have extensively studied packet classification, the longest prefix matching problem and the related literature and worked on several issues of them. We designed, simulated and proposed classification solutions that exploit the most important information existing in the packet headers. We have designed and implemented hardware schemes that can support high speed packet classification based on the packet's headers of network layers 2, 3 and 4.

In Chapter 3 we propose a classification solution for the MAC layer of the Ethernet networks. We used a hashing scheme and an internal replacement of MAC Vendor IDs to compact the MAC address tables and support high speed decisions. The proposed hardware scheme, *Hash Based Classification Engine* (HBCE), uses modest amount of memory and a single memory to store and retrieve its data structures. When HBCE is implemented with on-chip memories it can support aggregate speeds of more than 50 Gbps. In Chapter 4 we fully describe a reference hardware implementation of HBCE that can be implemented in FPGAs.

Chapter 5 presents our solution for the Longest Prefix Matching (LPM) problem that mainly applies in route lookups. We developed an innovative data structure that uses bitmaps to compact the prefixes and retrieve them in relatively high speed. When the proposed solution, *Bitmap Oriented Strides* (BOS) is implemented on-chip with parallel memory arrays it can support destination route lookups of more than 240 Million packets per second, translated into 80Gbps.

This thesis also proposes a novel packet classification scheme for the IP 5-tuple in Chapter 6. The proposed solution, *Bloom Based Packet Classification* (B2PC), approaches the packet classification problem in a decomposed manner, where single field matches of each packet field are combined to identify the matching rule. B2PC uses the BOS solution for LPM to provide efficient single field independent matches of 5D classification rules. Moreover, it represents internally the 5D classification rules and stores them in Bloom filter data structures so as to provide fast and efficient set membership queries. On-chip implementation of B2PC with parallel BOS engines provides classification of packets at rates greater than 8Gbps for more than 4000 rules. In Chapter 7 we fully describe a reference hardware implementation of B2PC that can be implemented in FPGAs.

8.2 Future work

Our solution to the Longest Prefix Matching, BOS, is strictly restricted to find prefix matches for IPv4 addresses. It will be rather useful to examine whether the data structures involved in BOS could be used to support IPv6 [3] routing lookups. The IPv4 addresses are 32-bits long and the IPv6 addresses are defined to be 128-bit. These 128-bit addresses could be possibly split into 32-bit segments and follow a decomposition solution similar to that proposed for B2PC. Hence, we may use 4 parallel 32-bit BOS engines to examine each segment independently and combine all the intermediate results. Moreover, another interesting point is how the BOS hardware implementation can scale in respect of state-of-the-art deep submicron chip technologies.

On the other hand, our packet classification solution (B2PC), was designed to support a few thousand rules and this restricts its scalability. However, the arrival of new network protocols for dynamic resource reservation, like RSVP [45], can increase the number of rules to hundreds of thousands. Hosting such a large number of rules demands altering many parameters of the scheme, like the Bloom filters' sizes and the associated memory sizes and this should be extensively studied. Moreover, we can study how B2PC could support additional packet header fields beyond the standard IP 5-tuple. Adding more fields in B2PC requires more parallel single field searches which would naturally increase the number of intermediate results. Handling and combining an increased number of intermediate results can become a serious threat to the performance our scheme. The possible number of single field permutations could be a serious bottleneck and may require more sophisticated combination techniques.

References

[1] *"Internet Protocol"*, RFC 791, September 1981.

[2] S. Fuller, T. Li, J. Yu, and K. Varadhan, *"Classless inter-domain routing (CIDR): an address assignment and aggregation strategy",* RFC 1519, September 1993.

[3] S. Deering and R. Hinden, "*Internet Protocol, Version 6 (IPv6) Specification"*, RFC 2460, December 1998.

[4] IEEE 802.1q Standard, *"Virtual Bridged Local Area Networks"*, http://standards.ieee.org/getieee802/download/802.1Q-2003.pdf

[5] D. E. Knuth, *"Sorting and Searching, vol. 3 of The Art of Computer Programming"*, Addison- Wesley, 1973.

[6] B. H. Bloom, *"Space/Time Trade-offs in Hash Coding with Allowable Error"*, Communications of the ACM, vol. 13, pp. 422–426, July 1970.

[7] Broder and M. Mitzenmacher, *"Network applications of bloom filters: A survey"*, in Proceedings of 40th Annual Allerton Conference, October 2002.

[8] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, *"Summary cache: A scalable wide-area web cache sharing protocol"*, IEEE/ACM Transactions on Networking, vol. 8, pp. 281–293, June 2000.

[9] J. McAulay and P. Francis, *"Fast Routing Table Lookup Using CAMs"*, in IEEE Infocom, 1993.

[10] K. Sklower, *"A tree-based routing table for Berkeley Unix"*, Tech. Rep., University of California, Berkeley, 1993.

[11] V. Srinivasan and G. Varghese, *"Faster IP Lookups using Controlled Prefix Expansion"*, in IEEE Sigmetrics, 1998.

[12] P. Gupta, S. Lin, and N. McKeown, "*Routing Lookups in Hardware at Memory Access Speeds"*, in IEEE Infocom, 1998.

[13] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink, "*Small Forwarding Tables for Fast Routing Lookups",* in ACM SIGCOMM, 1997.

[14] W. N. Eatherton, *"Hardware-Based Internet Protocol Prefix Lookups"*, MSc thesis, Washington University in St. Louis, 1998.

[15] B. Lampson, V. Srinivasan, and G. Varghese, *"IP Lookups Using Multiway and Multicolumn Search"*, IEEE/ACM Transactions on Networking, vol. 7, no. 3, pp. 324–334, 1999.

[16] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, *"Scalable high speed IP routing table lookups"*, in Proceedings of ACM SIGCOMM '97, pp. 25–36, September 1997

[17] Feldmann and S. Muthukrishnan, *"Tradeoffs for Packet Classification"*, in IEEE Infocom, March 2000.

[18] J. van Lunteren and T. Engbersen, *"Fast and scalable packet classification"*, IEEE Journal on Selected Areas in Communications, vol. 21, pp. 560–571, May 2003.

[19] T. Y. C. Woo, *"A Modular Approach to Packet Classification: Algorithms and Results"*, in IEEE Infocom, March 2000.

[20] P. Gupta and N. McKeown, *"Packet Classification using Hierarchical Intelligent Cuttings"*, in Hot Interconnects VII, August 1999.

[21] S. Singh, F. Baboescu, G. Varghese, and J. Wang, *"Packet Classification Using Multidimensional Cutting"*, in Proceedings of ACM SIGCOMM'03, August 2003. Karlsruhe, Germany.

[22] V. Srinivasan, S. Suri, G. Varghese, and M.Waldvogel, *"Fast and Scalable Layer Four Switching"*, in ACM SIGCOMM, June 1998.

[23] D. Decasper, G. Parulkar, Z. Dittia, and B. Plattner, *"Router Plugins: A Software Architecture for Next Generation Routers"*, in Proceedings of ACM SIGCOMM, September 1998.

[24] T. V. Lakshman and D. Stiliadis, *"High-Speed Policy-based Packet Forwarding Using Efficient Multi-dimensional Range Matching"*, in ACM SIGCOMM, September 1998.

[25] F. Baboescu and G. Varghese, *"Scalable Packet Classification"*, in ACM SIGCOMM, August 2001.

[26] P. Gupta and N. McKeown, *"Packet Classification on Multiple Fields"*, in ACM SIGCOMM, August 1999.

[27] IEEE 802.1p Standard, *"LAN Layer 2 QoS/CoS Protocol for Traffic Prioritization"*.

[28] http://www.ncasia.com/rfq/24port_0303.cfm?rfq=Enterprise_24-port_rack-mount_switch

[29] N. McKeown, B. Prabhakar, *"Lectures on Packet Switch Architectures II – Address Lookup and Classification",* http://www.stanford.edu/class/ee384y/Handouts/EE384y_lookups_1.pdf

[30] R. Jain, *"A Comparison of Hashing Schemes for Address Lookup in Computer Networks"*, IEEE Transactions on Communications, Vol. 40, No. 3, October 1992, pp. 1570-1573

[31] IEEE OUI and Company_id Assignments, http://standards.ieee.org/regauth/oui/index.shtml

[32] XilinxVirtex II Pro FPGA Platform, http://www.xilinx.com/xlnx/xil_prodcat_landingpage.jsp?title=Virtex-II+Pro+FPGAs

[33] Cypress CY7C1371C, *"512K x 36 Flow-Through SRAM with NoBL™ Architecture".*

[34] Xilinx Tutorial, *"Designing Custom OPB Slave Peripherals for MicroBlaze".*

[35] Synopsys Corporation, *"Design Compiler"*, http://www.synopsys.com/products/logic/design_compiler.html

[36] Internet Performance Measurement and Analysis (IPMA) project, http://www.merit.edu/~ipma/

[37] Nen-Fu Huang, Shi-Ming Zhao, *"A novel IP-routing lookup scheme and hardware architecture for multigigabit switching routers."* IEEE Journal on Selected Areas in Communications June 1999: 1093 -1104

[38] J. van Lunteren - IBM Zurich , *"Searching very large routing tables in fast SRAM"*, in Proceedings of 10th International Conference on Computer Communications and Networks, 2001 : 4-11

[39] Y. Rekhter, T. Li, *"A Border Gateway Protocol 4 (BGP-4)"* , RFC1771, March 1995

[40] M. Kounavis, A. Kumar, H. Vin, R. Yavatkar and A.Campbell. *"Directions in Packet Classification for Network Processors".* 9th International Symposium on High-Performance Computer Architecture, February 2003.

[41] F. Baboescu, S. Singh, and G. Varghese, *"Packet classification for core routers: Is there an alternative to CAMS?"* in INFOCOM, 2003.

[42] David Taylor and Jonathan Turner , *“ClassBench: A Packet Classification Benchmark”*, *Proceedings of Infocom*, March 2005.

[43] Pareto Distribution , http://en.wikipedia.org/wiki/Pareto_distribution

[44] Sarang Dharmapurikar, Praveen Krishnamurthy, Todd S. Sproull, John W. Lockwood, *“Deep Packet Inspection using Parallel Bloom Filters”*, IEEE Micro, January 2004

[45] Lixia Zhang, Stephen Deering, and Deborah Estrin, *“RSVP: A New Resource ReSerVation Protocol”*, IEEE network, 7, September 1993.