Technical University of Crete

Electronic Engineer and Computer Engineer Department

Microprocessor & Hardware Laboratory (MHL)

Design and Implementation of a 5d Classification Engine

Master's Thesis

Antonios Nikitakis

Committee: Assist. Professor I. Papaefstathiou (Thesis Advisor)
Professor A. Dollas
Assoc. Professor D. Pnevmatikatos

Chania, Crete 2008

Acknowledgments

I have somehow managed to spend some years, working towards my M.Sc. Through all these years I have been fortunate to interact with many people who have influenced me greatly. One of the pleasures of finally finishing is this opportunity to thank them.

First of all I would like to thank my supervisors, Prof. Yiannis Papaefstathiou and Prof. Dionisios Pnevmatikatos who gave me the opportunity to work on the subject of packet classification, and whose expertise, understanding, and patience, added considerably to my graduate experience. I would like to thank Manos Touloupis, for the assistance he provided at the level of testing and verification. I would like to give a special thanks to Vassilios Papaefstahiou whose great previous work on packet classification inspired me and helped me to compose this thesis. Finally I would also like to thank all the members and colleagues from the MHL laboratory (www.mhl.tuc.gr). Special thanks to: Spyros Ninos, Evripidis Sotiriadis, Kyprianos Papademitriou, Ioannis Ermis, Iosif Koidis, Grigoris Chrysos, Dimitris Meintanis.

I would also like to thank my family for the support they provided me through all these years of studies.

Contents

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.

Due to the rapid growth of traffic in the Internet, backbone links of several gigabits per second are commonly deployed. To handle gigabit-per-second traffic rates, the backbone routers must be able to forward millions of packets per second on each of their ports. After the introduction of Classless Inter-Domain Routing (CIDR) in 1993, IP address look-up has become a much more difficult task. That is because now, a router must not only find an exact prefix match or to choose the Longest Prefix Matching (LPM) according to the IP destination address but may also have to deal with multiple fields. 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 applied techniques utilizes determines the performance of the router, and hence the entire Internet.

1.1 Internet and Networking

The **Internet** is a "network of networks" that consists of millions of smaller domestic, academic, business, and government networks, which together carry various information and services, Each of these networks consisting of heterogeneous hosts, links, and routers. Hosts send and receive packets, or datagrams, which contain chunks of data - a part of a file, digitized voice samples, etc.

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.

1.1.1 Ethernet Networks

Ethernet is a family of frame-based computer networking technologies for local area networks (LANs). 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.

1.1.2 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) [4] 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**.

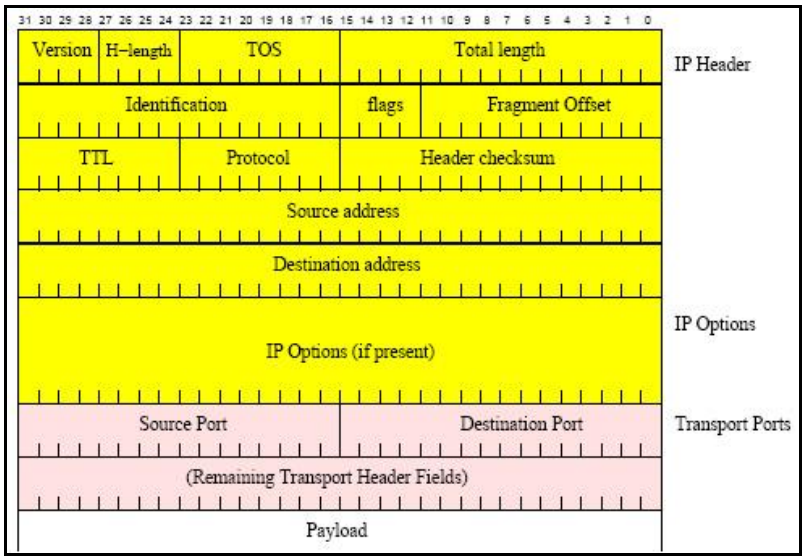


Figure 1.1: IP header format

1.1.3 Internet Addressing

IPv4 is the dominant network layer protocol on the Internet and apart from IPv6 it is the only standard internetwork-layer protocol used on the Internet. 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 [5]. 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) [6].

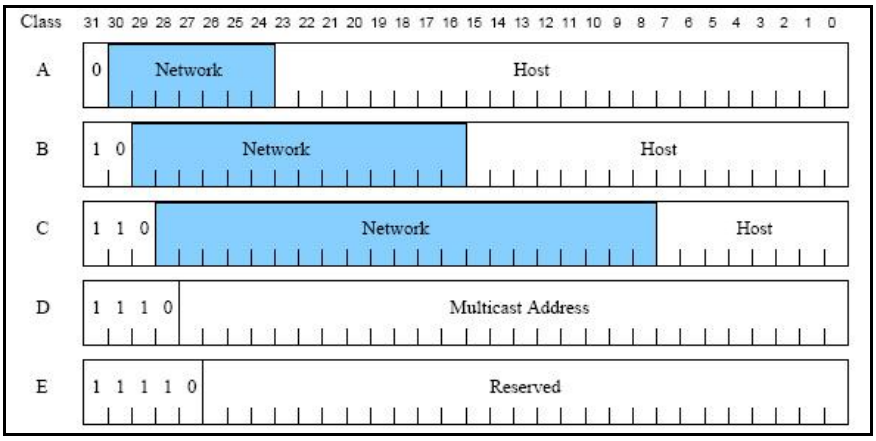


Figure 1.2: Class Based Internet Addressing

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 [7]. 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 [1]

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.

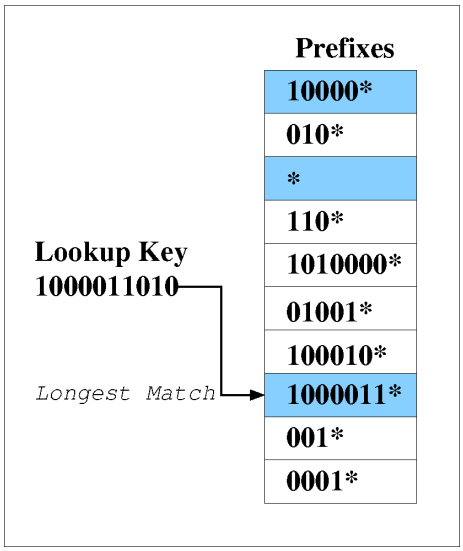


Figure 1.3: Longest Prefix Match Example

1.4 The Packet Classification Problem

Packet classification enables a number of additional, non-best-effort network services other than the provisioning of differentiated qualities of service. One of the well-known applications of packet classification is a firewall. Other network services that require packet classification include policy-based routing, traffic rate-limiting and policing, traffic shaping and billing. In each case, it is necessary to determine which flow an arriving packet belongs to so as to determine — for example — whether to forward or filter it, where to forward it to, what type of service it should receive, or how much should be charged for transporting it. [3]

Typically, the packet classification problem is referred as 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. 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 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:

[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 *.

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.

1.4.1 Packet Classification Challenges

Computational complexity is not the only challenge 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 a hardware solution to meet the delay sensitive searching tasks required by the network infrastructures. We proposed a classification engine based on Bloom Filters (2sBFCE). This engine decomposes multiple-field packet classification, into single-field and combines them in an efficient way. The innovation of this design is the performance and simplicity achieved by using Bloom Filters data structures in all stages of the classification procedure.

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 our Bloom Filter Based Classification scheme (2sBFCE). In Chapter 4 we present in detail the hardware implementation of 2sBFCE. Finally, in Chapter 5 we provide a summary of the contributions an a discussion of future work of this thesis.

Chapter 2

Related Work

In this chapter we present the most important 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 major 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.

2.1.2 B-Trees

B-Trees were originally designed to limit the number of accesses to direct access storage units such as disks [8]. In B-trees, internal nodes can have a variable number of child nodes within some pre-defined range. 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:

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

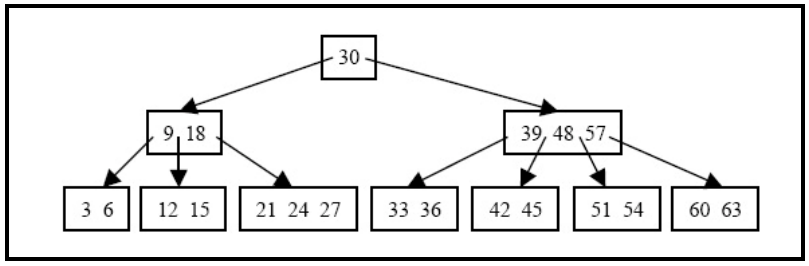


Figure 2.1: Example B-Tree data structure

2.1.3 Hashing

A **hash function** is any well-defined procedure or mathematical function for turning some kind of data into a relatively small integer, that may serve as an index into an array. **Hashing** is a technique using hash functions 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 [8].

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 [8].

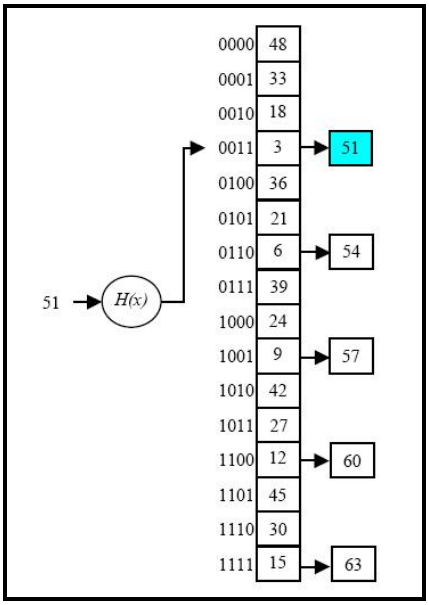


Figure 2.2: Hash function example

2.1.4 Bloom Filters

A Bloom Filters is a space-efficient probabilistic data structure that is used to test whether an element (key) is a member of a set. 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 conceived by Burton H. Bloom in 1970 [9], and has received renewed attention in the research community for various applications such as web caching, intrusion detection, and content based routing [10].

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. An empty Bloom filter is a bit array of m bits, all set to 0. 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.

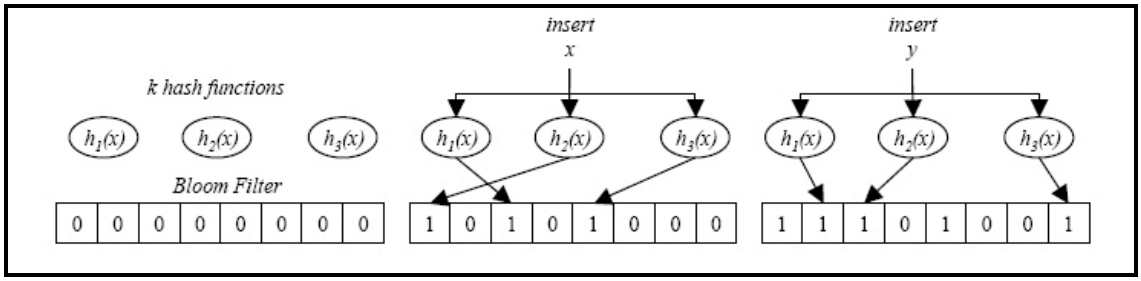


Figure 2.3: Bloom Filter Example

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. [11]. 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.

2.1.5 Longest Prefix Match

Because each entry in a routing table may specify a network, one destination address may match more than one routing table entry. The most specific table entry — the one with the highest subnet mask — is called the

Longest Prefix Match (LPM). It is called this because it is also the entry where the largest number of leading address bits in the table entry match those of the destination address. 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) [5]. 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.

2.1.6 Linear Search

Linear Search is one of the most fundamental and simple searching methods. 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.

2.1.7 Content Addressable Memory (CAM)

Content-addressable memory (CAM) is a special type of computer memory used in certain very high speed searching applications. It is also known as associative memory, associative storage. Many commercial router designers have chosen to use 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 [12], 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.

2.1.8 Trie Based Schemes

A trie, or prefix tree, is an ordered tree data structure that is used to store an associative array where the keys are usually strings. Unlike a binary search tree, no node in the tree stores the key associated with that node; instead, its position in the tree shows what key it is associated with.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.

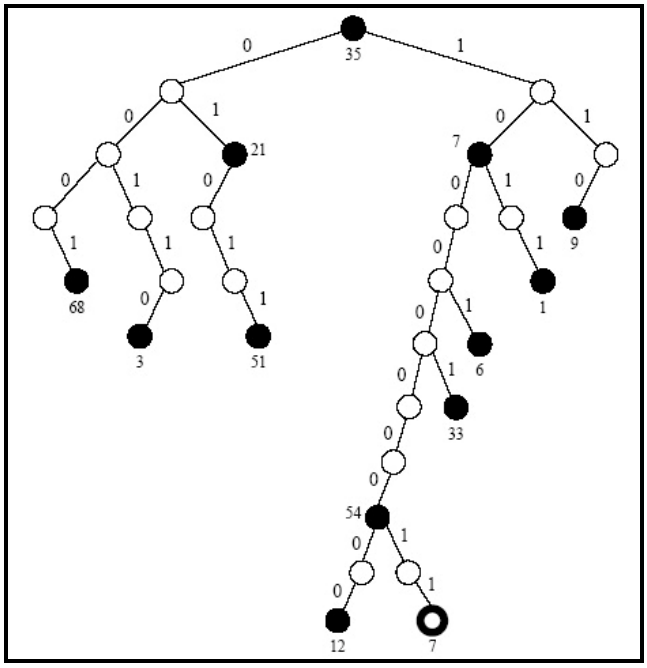


Figure 2.4: Binary Trie example

One of the first IP lookup techniques to employ *tries*[2] is Sklower's implementation of a **Patricia trie** in the BSD kernel [13]. 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* [14]. *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, 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 [15]. 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.

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

Other techniques such as *Lulea* [16] and Eatherton and Dittia's *Tree Bitmap* [17] 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.

2.1.9 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 [18]. 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.

2.1.10 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.[19]. 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.

2.1.11 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.

2.1.12 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 [20] or convert each closed interval $[i,j]$ into a set of prefixes and then employ one of the Longest Prefix Matching (LPM) algorithms.

2.1.13 Interval Tree

An **Interval Tree**, also called a segment tree or segtree, is an ordered tree data structure to hold intervals. Specifically, it allows one to efficiently find all intervals that overlap with any given interval or point. It stores a set of closed intervals X using a balanced binary tree as the underlying data structure [8]. Each node in the Interval Tree stores an interval $x \in X$. 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**.

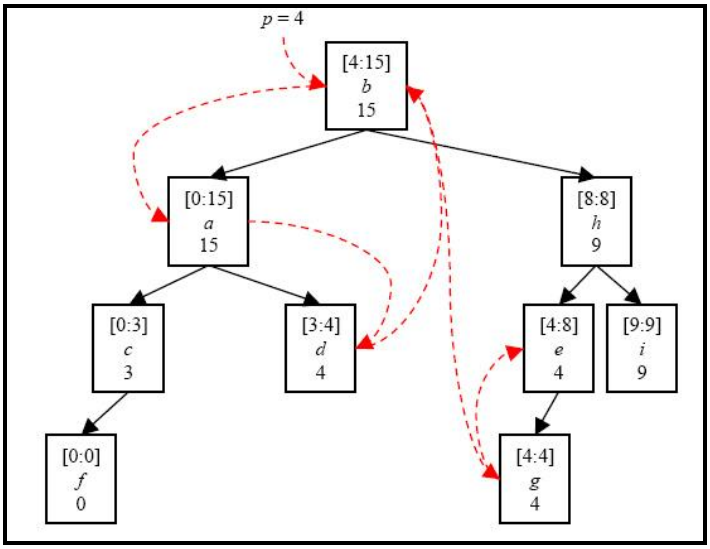


Figure 2.5: Interval Tree example

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.

2.1.14 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 × (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 [21], [22]. Finally, we note that Feldman and Muthukrishnan [20] 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.

2.2.2 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. The simplicity of linear search make it as suitable solution for the final stage of a lookup when the set of possible matching filters has been drastically reduced [22][23][24].

2.2.3 Ternary Content Addressable Memory (TCAM)

Ternary Content Addressable Memory (TCAM) is a memory technology which can do a very wide data search in a very short, fixed time period (~20ns). Alike fully-associative cache memories, 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.

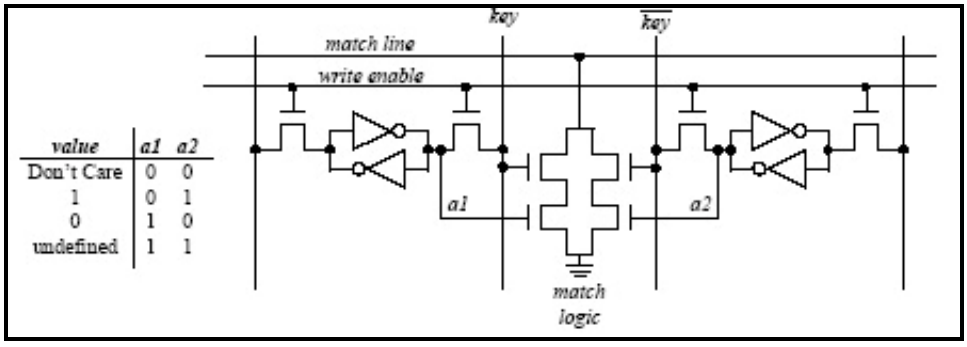


Figure 2.6: A typical TCAM cell

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.

2.2.4 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.

2.2.5 Grid of Tries

Srinivasan, Varghese, Suri, and Waldvogel introduced the original *Grid-of-Tries* algorithm for packet classification [25]. *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 [26]. 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.

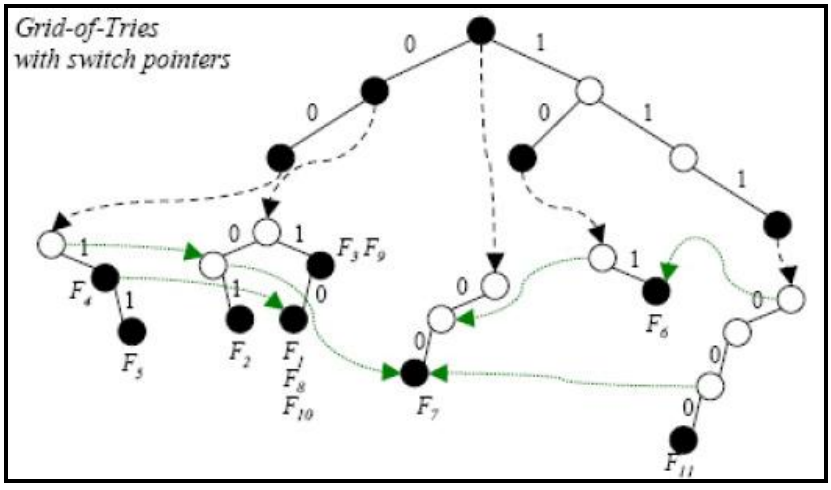


Figure 2.7: Grid of Tries data structure

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.

2.2.6 Hierarchical Intelligent Cuttings (HiCuts)

Gupta and McKeown introduced an innovative technique called *Hierarchical Intelligent Cuttings* (*Hi-Cuts*) [23]. 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.

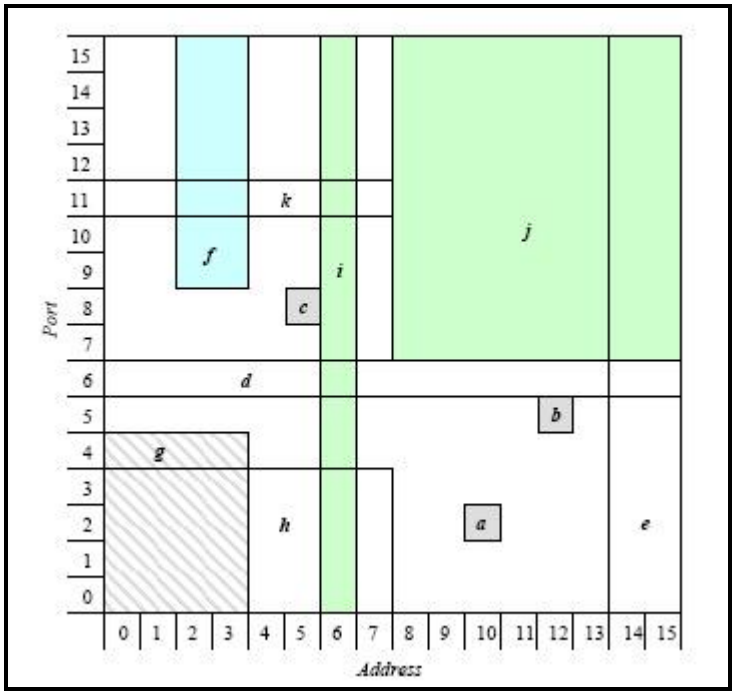


Figure 2.8: HiCuts geometric representation

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.

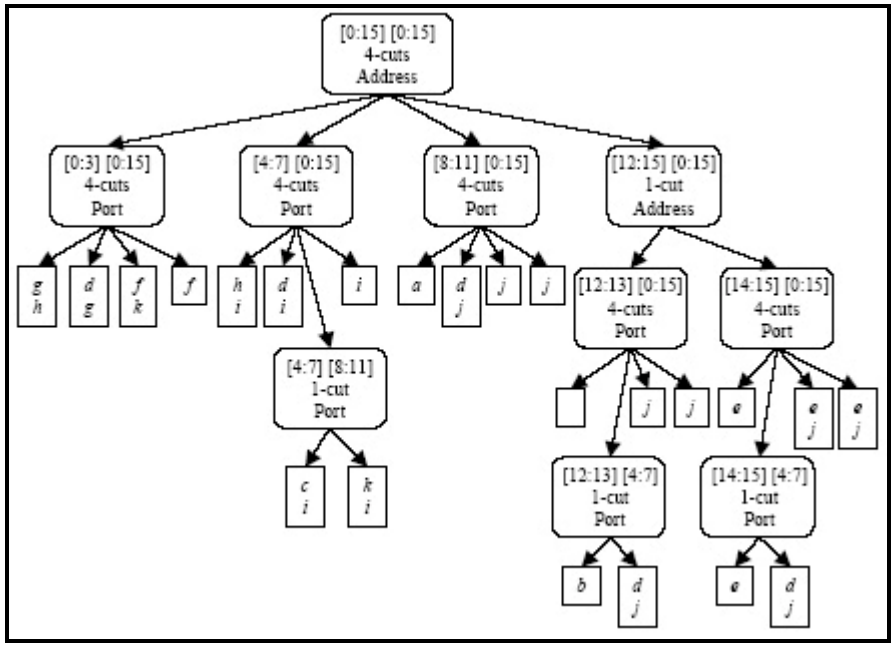


Figure 2.9: HiCuts Data Structure

2.2.7 Fat Inverted Segment (FIS) Trees

Feldman and Muthukrishnan introduced a scheme for packet classification using independent field searches on Fat Inverted Segment (FIS) Trees [20]. 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.

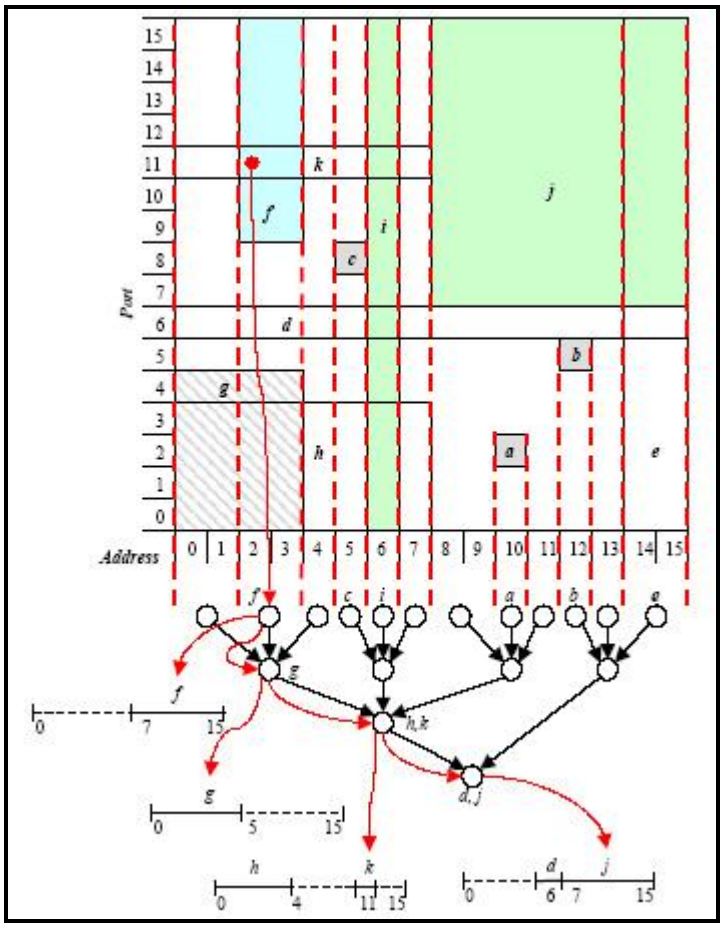


Figure 2.10: FIS example

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.

2.2.8 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.

2.2.9 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*) [26]. 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.

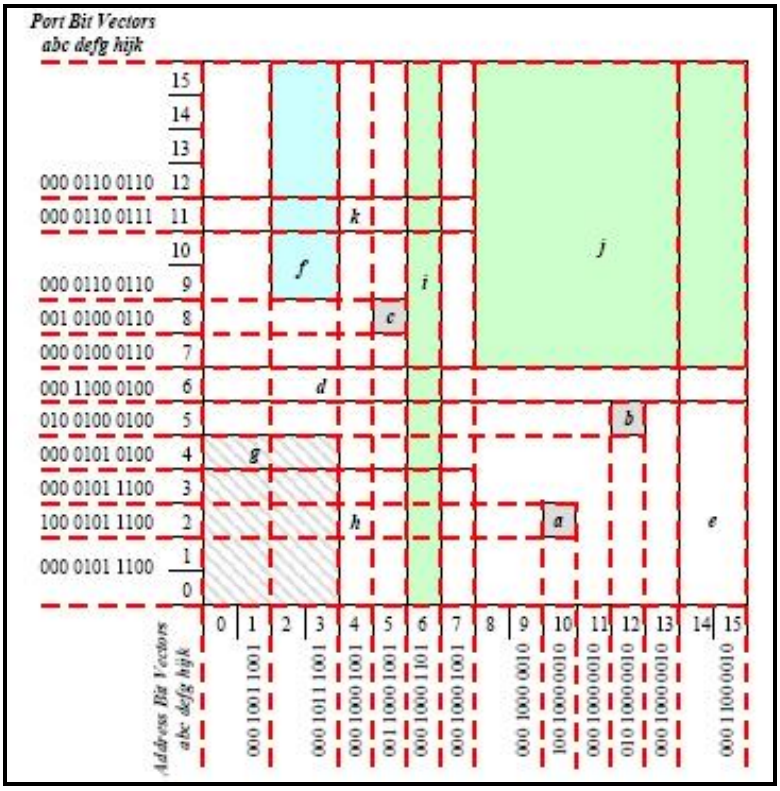


Figure 2.11: Parallel Bit Vectors example

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(logN)** 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.

2.2.10 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 [28]. 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.

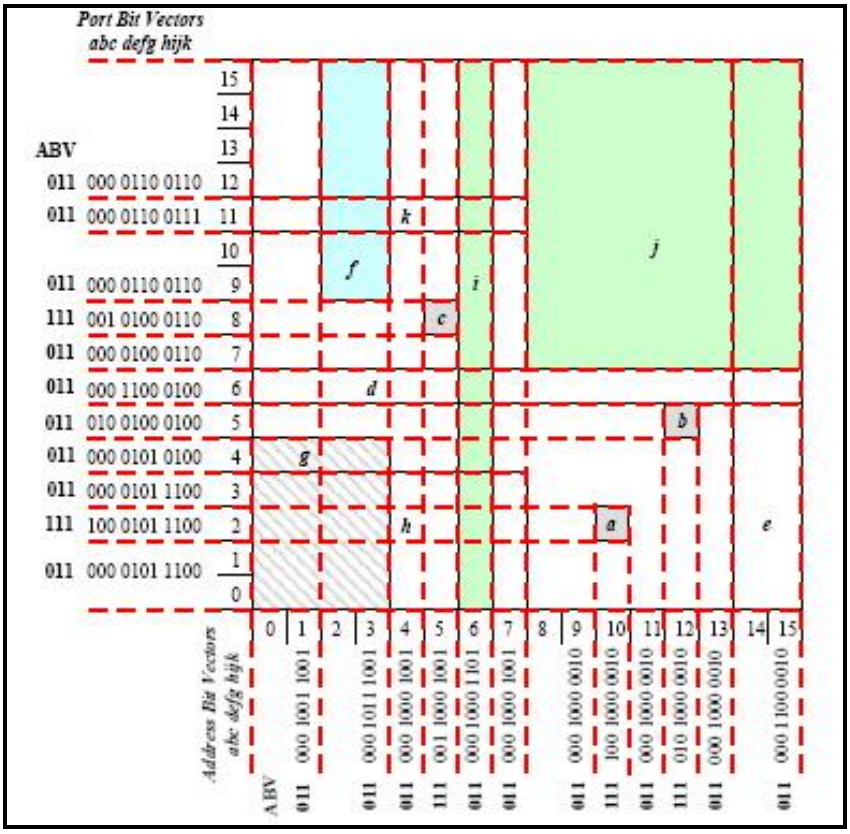


Figure 2.12: Aggregated Bit Vector example

2.2.11 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 [29]. 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.

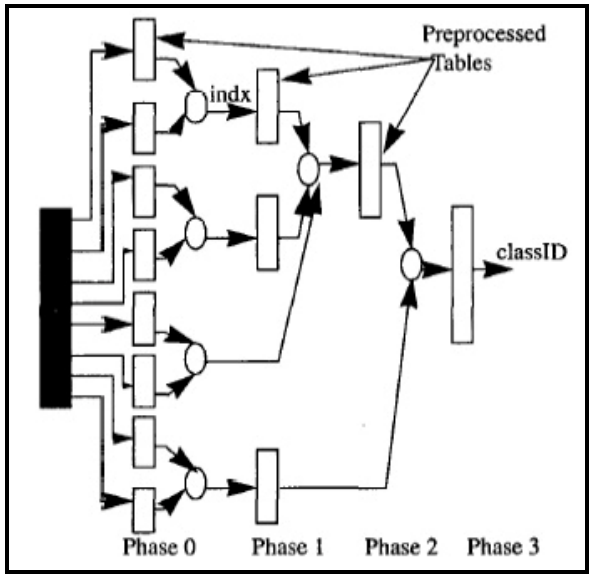


Figure 2.13: RFC aggregation scheme

Chapter 3

Bloom Filter Based Packet Classification

In this chapter we present a Bloom Filter Based Packet Classification scheme. We call it ***2 stage Bloom Filter Classification Engine*** **(2sBFCE) a**nd it is suitable for pipelined hardware implementation. 2sBFCE 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 2sBFCE is optimized for the common case based on analysis of real world filter sets.

3.1 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[3].

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 [29], while others have performed analyses on real filter sets and published their observations [43][44]. 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 single field values is significantly less than the number of overall 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 [45] 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.

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

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[46] that is widely used to model the Internet traffic.

3.2 2sBFCE Design and Description

2sBFCE 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. For our implementation we dedicate to the protocol field 2-bits as the common case includes: TCP, UDP and wildcard. We do this in order to improve the performance of our system. 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 3.1**.

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 3.1 Filter Set Example

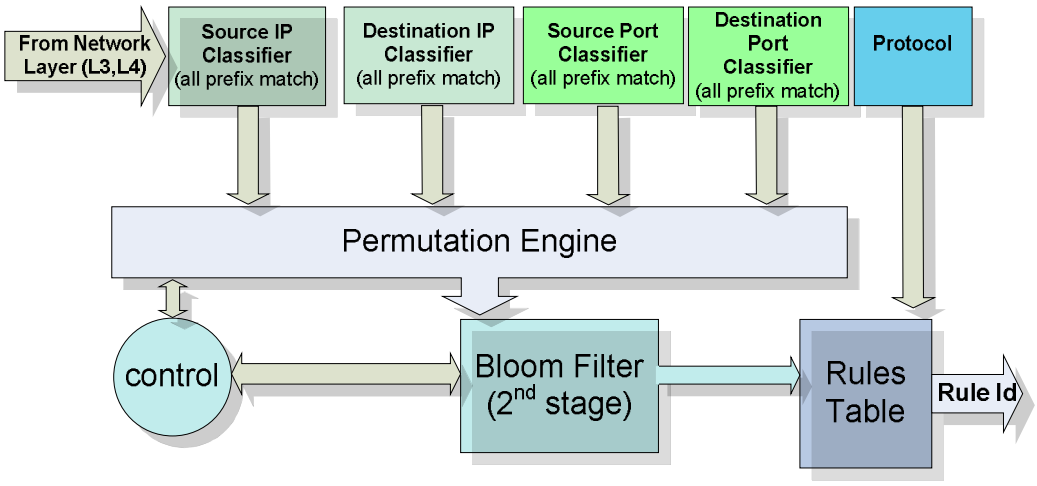
3.2.3 Single Field Operations

Given that 2sBFCE follows the decomposition approach, it is essential to employ a very efficient single-field scheme supporting both exact and prefix matches at very high speeds, while utilizing small amounts of memory. Those

requirements are fulfilled by a bloom filter scheme described in the following section. Our single-field lookup mechanism not only report the longest prefix match but, instead, all the prefixes that match, and for each match the associated match length which is used later on the process.

Based on the observations described in the last subsection the proposed scheme supports up to 4K 5-tuple rules, therefore, each filter can be identified by a 12-bit *FlowID*. The number of supported 5-tuple rules could be easily increased without affecting either the architecture or the overall performance of the design. A general overview of the 2sBFCE scheme is presented in **Figure 3.1** where all the discrete components are shown. The idea of 5-tuple classification is based on single field classification. Therefore, for each field we perform all prefix match lookup and then using a permutation process we combine these results.

Figure 3.1 : Overall Architecture of 2sBFCE

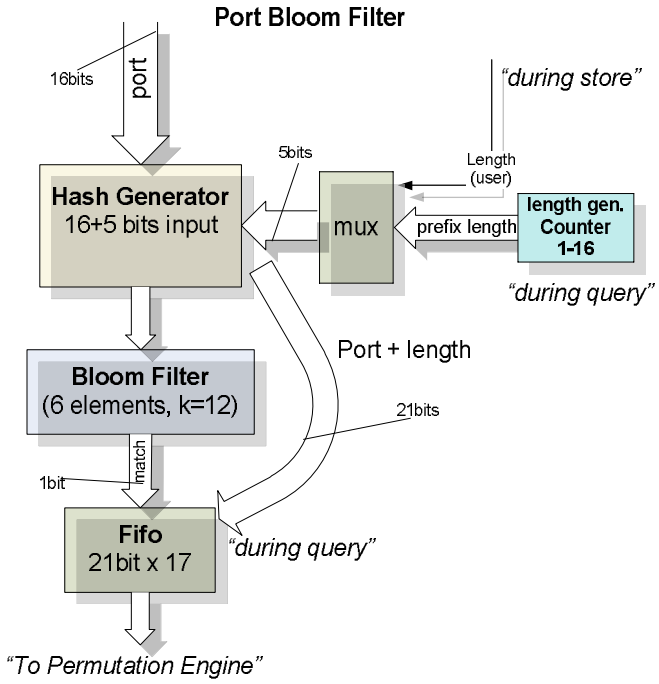


3.2.4 Internally Represented Filters

The internal representation of prefix variations in each field is based on the combination of each IP filed with the corresponding prefix length. No other internal Id is assigned. This scheme produces a representation of 32+6 bits for the SourceIP and DestIP prefixes, 16+5 bits for the Source-Port and Dest-Port prefixes and last 2 bits for the Protocol (no prefix length used). Each of these representations is applied to an individual Bloom Filter-based classification engine (sub-engine). **Figure 3.2** shows the organization of a Bloom Filter

engine used in Source and Destination Port sub-engines. Similar Bloom Filters are also used in the SourceIP and Dest.IP sub-engines. Since Protocol comprises of just two bits, in our implementation it is treated in a different way as it will be explained later.

Figure 3.2: Organization of a Port Sub-Engine



During the store procedure each single field prefix is stored at the corresponding sub-engine and the matching Bloom Filter (BF) element is set. In parallel, all these single field representations are combined to a single 120-bit "rule vector" using the Permutation Engine. This vector is applied to the second BF Stage in order to store the information of this specific combination of each single field which is needed for the later query procedure. This 120-bit "rule vector" is also combined with a unique 12-bit flow ID and kept in the 4K entry RulesTable. The RulesTable is indexed by a hash function using the 120-bit vector information as input. This hash indexing scheme introduces a collision probability during store procedure which could lead to an unwanted rule overwrite. This problem is described in Section 3.3. Table 3.2. shows an example of a real-like filter set.

3.2.5 Combining Results

Given the 5 fields of a packet, the 2sBFCE has to find which of the existing rules best matches all of them. In the first stage, the five single-field engines provide a number of matching prefixes. 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. Similarly, the port fields may provide at most 17 matches. In the protocol field we have only exact-value and "match-all" searches, resulting in at most 2 matches. For that reason we don't dedicate a separate sub-engine for the protocol filed, we just deal with it at the last stage of the procedure. The results from every single-field engine should be combined, so as to cover all the possible permutations, and then it should be determined which of these permutations are actually valid (i.e. whether such a multi-field rule exists). The above task is performed during query procedure in the Permutation Engine module. The Permutation Engine combines all the matches of each single field into a number of possible matching 120-bit "rule vectors". Each of these vectors is examined in the second stage Bloom Filter scheme as explained in Section E. The total number of possible permutations is equal to the overall product of the number of matches in every field :

$$\mathbf{Total_{perm} = \#Src\ IP\ * \#Dest\ IP\ * \#Src\ Port\ * \#Dest\ Prt}$$

Protocol is not included because we don't deal with it at this moment, leaving this check in the end of the process. We are doing this to reduce in half the number of permutations as it is meaningless a 4-field matching combination to have two different entries varying only in protocol field. Each such combination should have either a specific value or a wildcard in protocol field.

Table 3.2: Example Filter Set [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

Although the possible number of permutations could be large, in real-world databases, as it was described in Subsection 3.2, the maximum number of matches in each field is typically less than five and the rules that match a certain incoming packet are usually less than five, as well.

3.2.6 Set Membership Queries with Bloom Filters

As mentioned before the combined results from every single-field engine should be examined so as to find out which of these are actually valid. One of the most important challenges of 2sBFCE, is how to identify that a permutation belongs to the given set of rules. Sequential access to the rule table is prohibitively slow since we may need to access every single entry of it. Therefore, a data structure that can efficiently represent a given rule set and support quick set membership queries, is needed. Hash tables and B-Trees are widely used for this type of queries but we use Bloom Filters. The main advantage of those filters, when compared to the other data structures, is that they can easily be implemented in hardware while supporting set-membership queries at extremely high rates. The disadvantage of Bloom filters is that they sometimes reports that a certain item is part of the set, even though it does not actually belong to this set (i.e. false-positive error). This happens due to the hash-based structure they use. As a result, for every combination which is a positive match in the second BF stage, we consider this combination as a possible multi-field matching rule. We still say “possible” because there is a false positive probability as mentioned before. We deal with this false positive scenario in the last stage of RulesTable. In the Table 3.3 we see a number of possible permutations but only permutation No19 is a valid one. Note that ID

Numbers represents different prefixes values in order to be readable and they don't have any use in the real procedure mechanism.

Table 3.3 : Example of total possible permutations [1]

Perm No	Src IP ID	Dest IP ID	Src Port ID	Dest Port ID	Protocol ID
1	12	47	10	26	10
2	12	47	10	26	17
3	12	47	10	31	10
●	●	●	●	●	●
●	●	●	●	●	●
18	54	47	10	10	17
19	**54**	**54**	**10**	**26**	**10**
20	54	54	10	26	17
●	●	●	●	●	●
●	●	●	●	●	●
35	23	62	10	10	10
36	23	62	10	10	17

A similar scenario could also happen during the single field query procedure. These false positive results don't affect the final result because they are automatically resolved during the second BF stage. However, they do affect the overall performance of the lookup as described in subsection F.

3.2.7 Bloom Filter Tuning

In the 2sBFCE, in order to efficient support classification databases with up to 4K rules, we employ a suitable Bloom Filter. A very important characteristic of the Bloom Filter is that its false positive rate can be tuned, as discussed in [47]. In order to keep this rate low, we have carefully chosen the size of the Bloom filter bit-vector and then calculated the corresponding optimal number of hash functions that set the filter's individual bits. Based on an analysis presented in detail in [1] we ended-up with a bit vector which is $3*2^{14}$ bits wide (we used 3 identical BF Elements -BFE). Based on this same analysis the optimal number of hashing functions that set the bits on this vector is 2. Given

those parameters, the produced BF has a theoretical false positive probability of 2.36 %.

The bit-vector of the BF is relatively large to be kept in registers/flip-flops, and therefore it is stored in a memory array. Having 2 hash functions means that we have to set 2 bit positions in the bit vector and 2 bits at each access. Due to the fact that the bit-vector is to be stored in a memory array we may require up to two memory accesses to locate each of the 2 bits. Thus, in order to avoid sequential accesses, and since the array is quite small and can easily be kept on-chip. For that reason we use the dual-port Memory modules of the Xilinx Virtex-4 FPGA. These modules have appropriate configured to have 1bit of width and to each port we assign a different hash function. 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 to the bit vector.

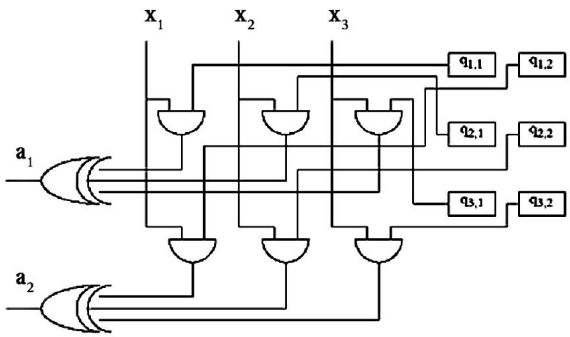
After careful analysis of the classification databases and the Bloom Filter properties, we have defined a hash function based on H_3 Class of Universal Hash Functions [1]. As it is supported, this class of hash functions, is suitable for hardware applications because of its computational simplicity and its high level of parallelism. The following formula describes the function we used:

hq(x):A→B (A: key space, B: address space)

$$h_q(x) = x(1) \bullet q(1) \oplus x(2) \bullet q(2) \oplus \ldots \oplus x(i) \bullet q(i)$$

where $\bullet$ denotes the biwise AND operation and $\oplus$ the bitwise XOR operation. The q(i) derives from a randomized vector. Figure 3.3 shows an example of producing a 2-bit hash value for a 3-bit Key and demonstrates how such a hash value is generated in hardware [1].

Figure 3.3: A hash function producing a 2-bit hash value for a 3-bit Key [1]



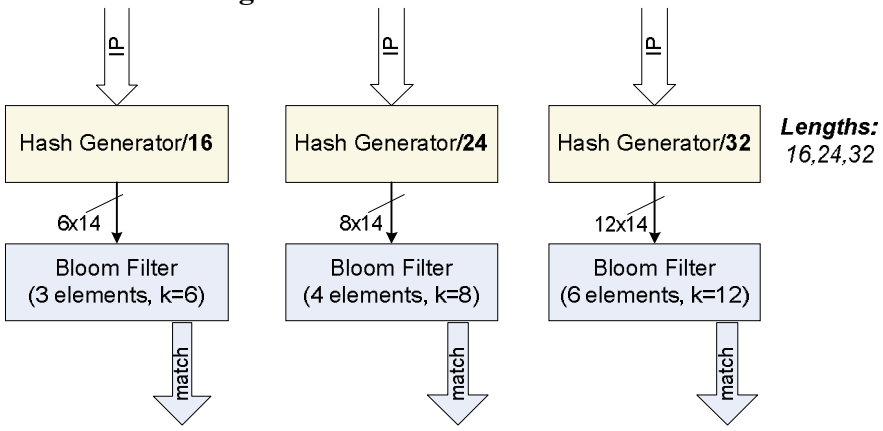
3.2.8 False Positives and Filter Tuning

Although the false positive matches in the first level of the single filed classifiers are easily resolved by the second bloom filter level, they produce a large number of redundant permutations. This can result in decreased performance. For that reason we have appropriate adjusted the parameters of the BF in order to reduce the false positive rate. For the IP fields (source , destination) we chose to use some dedicated filters. Each of those classifier module uses 1 dedicated bloom filter for the most common prefix lengths (16,24,32). A statistic analysis was performed in real-like data Rules Tables generated by ClassBench tool [45]. ClassBench involves a filter set generator that uses seeds from real filter sets to provide synthetic filter sets that accurately model real filters. According to this analysis in Table 3.4 we distributed 6 Bloom filter elements (BFE) for the "32-prefix length" filter, 4 BFE for the "24-prefix length" filter and finally 3 BFE for the "16-prefix length" filter. For the rest 28 (=33-3) different prefix lengths we use a 6 BFE arrangement as these prefix lengths sum up to 20% of the total rules. The zero prefix length corresponds to an all match case (* wildcard) which doesn't affect the overall false positive probability of the filter. Figure 3.4 depicts the 3 dedicated Bloom Filters used for the source and destination IP fields.

Table 3.4: Prefix Length Distribution

Prefix length	**0**	**16**	**24**	**32**
	SOURCE-IP			
Prefixes No	3255	876	1902	5993
%	**21,51**	**5,79**	**12,5**	**39,60**
	DEST-IP			
Prefixes No	1948	395	1247	9469
%	**12,87**	**2,61**	**8,24**	**62,57**
Total set size (rules)	**15133**			

Figure 3.4: Dedicated Bloom Filters

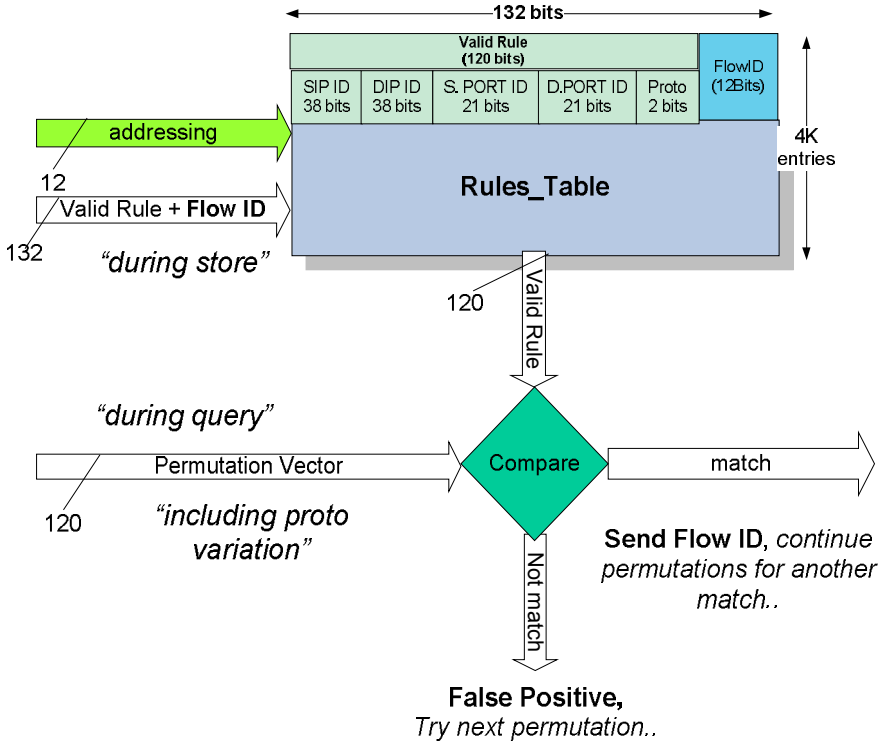


3.2.9 FlowID Resolving and bloom filter collisions

To locate the FlowID we use the 4K-entry RulesTable that give us the matched FlowID. As mentioned in subsection 3.2.6, once we have a valid permutation we use a hash indexing scheme to locate the appropriate RulesTable entry. After this we should determine whether it is a false positive match and in case it is not, we have to return the corresponding FlowID. In order to accomplish this task efficiently, during the store procedure, we keep the whole information of the rule in the RulesTable in combination with its FlowID. In that way, for each valid permutation, we read the whole rule from

the appropriate entry of the Rules Table and then we compare each field with the corresponding field of each permutation. If we have a full match then we send the 12-bit FlowID to the output pins; in the opposite case a false positive match has occurred. Then, we continue with the testing of the rest of the permutations. At this specific stage of the procedure we deal with the protocol field, by comparing as described before, the appropriate fields. More specific we compare the information carried in the packet (proto field) with this stored in each valid rule (including wildcard). If this field is a match (as well the as the other fields) then we have a valid rule match. Figure 3.5 demonstrates the whole procedure.

Figure 3.5: Organization of Rules Table scheme



In a single query it is possible to have more than one matching rules. To find out which of these matching rules is the best-matching we have to set some priorities among the 5-tuples first. These priorities may change from time to time so we leave this additional check to an upper lever of control. For that reason our classifier could reply with more than one matching result, until a Query_Done signal is set to 1.

3.3 Indexing the Rules Table and Incremental Updates

Indexing the Rules_Table requires a hash function and obviously this function may produce collisions. Resolving these collisions can be performed by using variable size blocks (such as in [49]) that hold the colliding FlowIDs. In the proposed implementation and for simplicity we have just one memory block where we put all the colliding FlowID and in case there is a collision we search all of them sequentially. In the next version of the classifier we plan to adopt a scheme similar to [49].

A property of Bloom filters is that it is "very hard to delete a message stored in the filter" [50]. This happens due to the hash based structure of BF which may causes collisions to some specific hashed bits in the bit vector. Deleting a particular entry requires that the corresponding hashed bits in the bit vector to be set to zero. This could disturb other messages programmed into the filter which hash to any of these bits. Therefore, in the current scheme we don't support incremental updates. In the next version and in order to cope with this problem we plan to maintain a vector of counters. Whenever a message is added to or deleted from the filter, the counters corresponding to the k hashed values are incremented of decremented, respectively. We delete a particular entry only when the corresponding counter reaches zero.

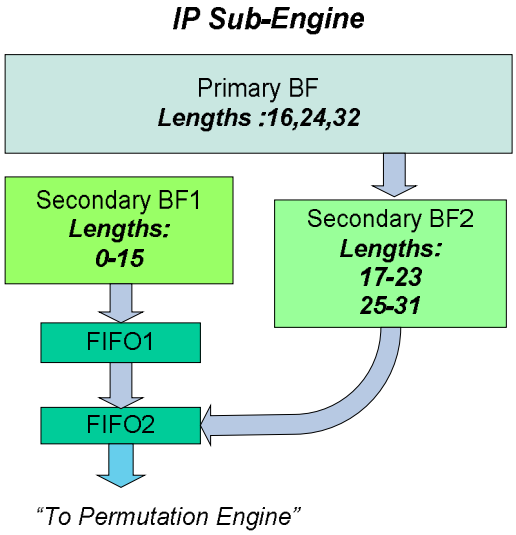
3.4 Improving Memory Access

As described in the last section, the lookup performance of the proposed scheme mainly depends on the set-membership queries in the Bloom filters. This is because the set-membership derives from the overall product of the single field results. In contrast the performance of the single field sub-engines remains practically constant as it needs 30-33 on-chip memory accesses. About 1 cycle for each prefix number which is examined and all the queries run in parallel.

In the implemented scheme we used a rather different scheme: instead of 1 we dedicate 2 bloom filter sets (BF1, BF2) for each of the *SourceIP and Dest.IP sub-engines.* This optimized scheme, improves the performance of these two sub-engines by 16 clock cycles as it splits the 32 possible prefix lengths in two smaller sets. Each set is consisted of 16 possible prefix lengths

and is applied to a separate Bloom filter. Thus the whole single-field query process finishes in about 17 cycles. The other two sub-engines (source & destination. port) always produce their results in at most 17 clock cycles. As a result, the whole single-field query procedure finishes in about17 clock cycles. Figure 6 shows the organization of our optimized IP Sub-Engine.

Figure 3.6: Optimized IP Sub-Engine



The second stage, as explained in Section C, needs one on-chip memory access per each permutation produced, as we dedicate one Bloom Filter for the entire set of permutations. Therefore the whole number of permutations depends on the data applied to the system. Simulation of ***real-world*** filter sets shows that the second stages needs 10-15 clock cycles in average.

3.5 Verification

In order to measure the efficiency of our scheme we employed a large number of realistic filter sets and test patterns.

During the early stages of the design we have used many custom made VHDL testbenches in order to check each component separately, such as the ***single field classifiers***, the ***permutation engine***, the ***BFs*** etc. Then, after the whole scheme was completed we have used more custom made testbenches with a few rules (from 10 to 100) in order to check that our system gives the

correct output for any given incoming packet. Thus for any given number of rules we have stored in our classifier we tried about the double number of packets during the query process in order to check the cross-matching rules and the wildcards as well. After this stage we were sure that our classifier was able to give the correct results for any given rule/packet. The next stage of the verification had to involve the performance factor when dealing with realistic sets as the performance of the algorithm we implement is dependent on the Rules Table characteristics.

For that reason, we used Taylor's ClassBench [44] which is a suite of tools for performance evaluation of classification algorithms and is publicly available. ClassBench contains a filter set generator that uses seeds from real-world filter sets in order to provide synthetic databases which model real filters in an accurate manner. ClasshBench produces 5d rules utilizing arbitrary ranges both on the Source Port and Destination Port field. This would be a problem for our scheme as it only supports prefix based rules. Thus, we had to transform the arbitrary ranges to prefixes and we chose to do this in an approximate manner in order to avoid the prefix expansion problem. Therefore we utilized a simple algorithm which transforms ranges to prefixes using only the one endpoint of each range.

ClasshBench, includes a packet header generator as well that produces a sequence of packet headers based on a given filter set. As we will explain in the next subsection we don't use this feature due to the approximate conversion of ranges we have made on each Rule Set before testing it in our system.

3.5.1 Simulation Results

In this subsection we discuss the simulation results based on synthetic filter sets and present our results on speed (clock cycles/packet). By using the ClassBench tool and the seeds from real filter sets that are provided by this tool we are able 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 8 synthetic filter sets of various sizes. Each of these testbenches is divided in two parts: at the first part we load the Rule Set in our scheme and at the second part we perform the query of all the rules one

by one in order to examine that our scheme gives the correct matching results. During the query we assume that each Rules Set generated with the ClassBench tool represents also the corresponding packet header file having one packet per rule. This assumption is possible if we exclude from the Rules Set file the all prefix length information and feed it back to the system as a packet header file. We expect that each packet from this file will match at least with the corresponding rule of the Rules Set we previously loaded in the classifier.

We did this because as mentioned before, we have modified all the Rules Sets in a way that each ClassBench generated packet header will no longer matches the corresponding Rules Set.

In Table 3.5 we see the performance results in clock cycles per packet according to each filter set we use. All the results are referred to the optimized design. We compute the performance of the first matching rule as well as the overall average performance according to the last matching result. We provide the performance of the first matching rule as some of the previous classification schemes are based on this performance.

Table 3.5 System performance using various filter sets

Filter name	No of Rules	Overall Average performance (last result) optimized design (cycles per packet)	Average performance (first result) optimized design (cycles per packet)
acl1	1192	56	26
ipc1	979	47	21
ipc2	618	35	27
fw1	975	42	21
fw2	653	36	21
fw3	692	34	21
fw4	1300	42	28
fw5	869	39	21

By studying the results we can draw the following conclusions:

1. The Rules Table characteristics affect the performance. As each Rules Set has a different number of overlapping prefixes per field we expect to give a different number of single field results. For example if a Rule Set

gives many single field matches in both Source and Destination IP field we expect that the overall permutations will be far more than a set that is dealing mostly with the Destination IP field. This explains the increased performance of the ***fw**** sets.

2. As the rules table size increases the performance is slightly decreased. According to the previous observation we expect that the increased size of the Rules Set will also increase the single field matches. This happens not because of the Bloom Filter's performance as we have adjusted all BFs to a minimum FPP. It happens due to the natural increase of the overlapping prefixes per field as the Rules Table getting larger.

3.5.2 Post place and Route verification

We performed a post-place and route simulation otherwise known as timing simulation on our design to verify that the functionality is correct after the place and route. This process uses the post-place and route simulation model (a structural SIMPRIM-based VHDL file) and a standard delay format (SDF) file generated by NetGen. The SDF file contains true timing delay information of the design.

After getting the timing simulation model of our design we were able to run the same tetbenches we ran during the functional verification of the design and verify the correct results.

Chapter 4

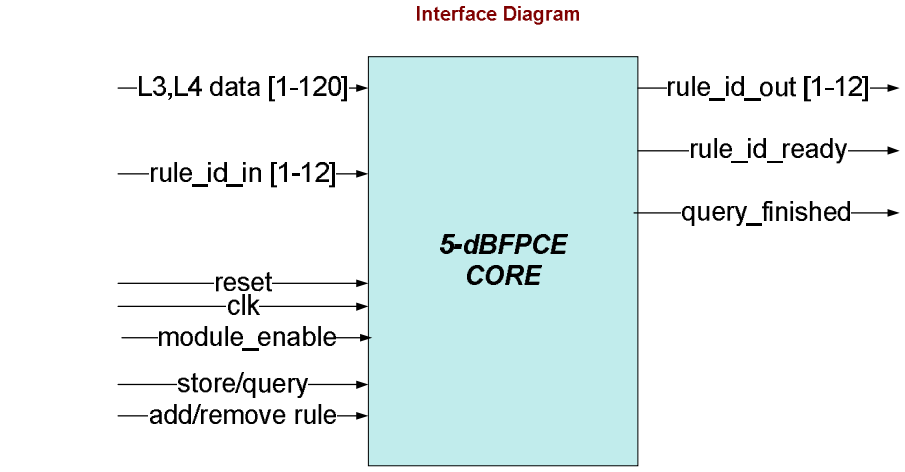
Hardware Implementation of 2sBFCE

In this chapter we present a reference hardware implementation of the 2sBFCE 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 hardware 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-4 [35].

4.1 2sBFCE Organization

The 2sBFCE consists of many internal blocks as shown in Figure 4.1. The system receives the packet information from the outside world (Network Layer) and gives as output a possible Rule Id match. Two separate function modes are supported by the system: the store and the query procedure. During the store procedure the *L3_Data signal* combined with the *rule_id_in* signal and provide the necessary information for each rule is stored in the system by the user. During query the *L3_Data* signal information is extracted from the incoming IP packet. After query is finished *rule_id_out* signal shows the rule id number (could be more than one) which matches to the incoming packet.

Figure 4.1: 2sBFCE core interface diagram



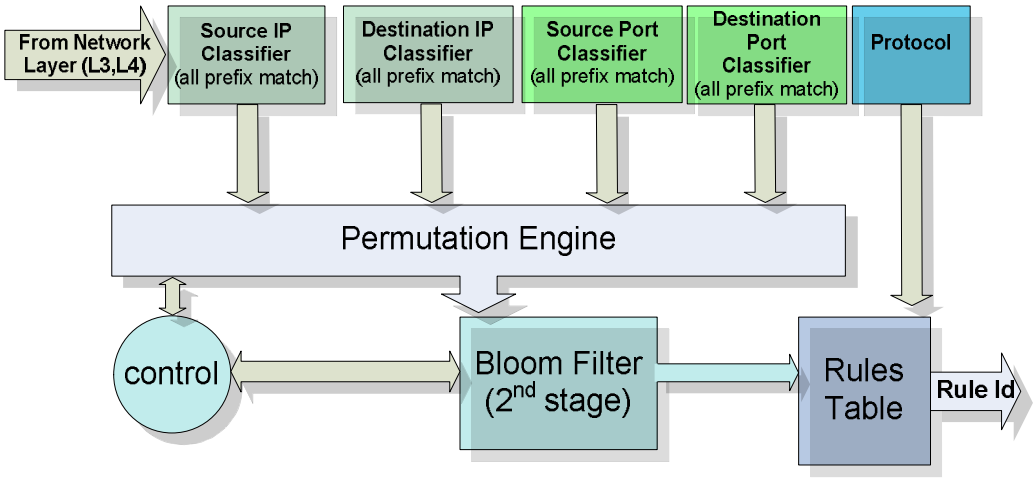


Figure 4.2: 2sBFCE module overview

The operation of the system is coordinated by the control block as shown in Figure 4.2. Upon a query procedure is started *Source IP Classifier, Destination IP Classifier, Source Port Classifier and Destination Port Classifier* are working in parallel to a single-field query (the above classifiers may referred also as *Single-Field Classification Engines or sub engines*). All the 4 single field classifiers are connected to the Permutation Engine Block in order to reconstruct all the possible 5-tuple rules that could be part of our rule set. The 2nd Stage Bloom Filter is responsible for filtering all these possible matching rules to only those which are valid, according to our rules set. In order to implement all the single field classification engines and the 2nd Stage BF as well, it was necessary to implement BF structures in hardware with adjustable

properties. The following section explains how a BF structure can be implemented using modern FPGAs.

4.2 Implementing the Bloom Filters [2]

4.2.1 Implementing Bloom Filter Elements (BFE)

In order to reduce the false positive probability considerably, a long bit vector is required. Even for supporting the programmability for hundreds of prefixes, many thousand bits are required. The cost of using the on-chip flip-flops for this purpose is too high. However, modern FPGAs have on-chip RAMs with more than one port (typically dual port) that can be exploited to create the Bloom filter vector [52]. For instance, the Xilinx Virtex 4 VFX140 chip has 552 on-chip RAMs [53] each with two ports and with the ability to operate at 333 Mhz. Each Block RAM can be configured as a single bit wide and 16,384 bits long vector requiring two 14-bit addresses and giving a bit look-up throughput of 2 bits per clock cycle. We now describe how these on-chip RAMs can be used for building the basic Bloom Filter.

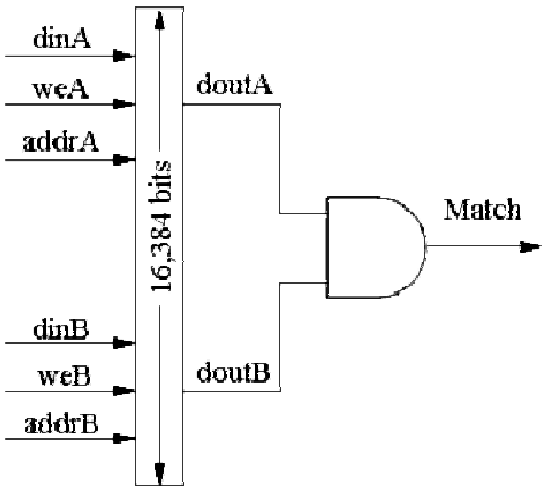


Figure 4.3: Dual Port Block Ram used as a Bloom Filter Element

Figure 4.3 shows how a Block RAM can be used to construct a Bloom filter. The Block RAM is configured as a single bit wide and 16,384 bits long bit array. It has two read/write ports, both of which can be used to look-up the bit values corresponding to two distinct hash functions. Thus, this Bloom filter can support k=2 hash function, m=16,384 in the array and hence can support

$$n = (\frac{m}{k}) \cdot \ln 2 = 5{,}678 \text{ prefixes with}$$

$$\text{P(false positive)} = (1/2)^2$$

We call this Bloom filter as a Bloom Filter Element (BFE) since multiple such Bloom filters are needed to create a Bloom filter with smaller false positive probability (P) [52] (or with larger number of supporting prefixes with the same false positive probability).

4.2.2 Creating Larger Bloom Filters

By using multiple BFE to store the same set of strings, the false positive probability can be reduced as explained previously. For instance, if two BFEs are used then the false positive probability will be

$$\left((1/2)^2\right)^2 = (1/2)^4$$

If three BFEs are used then the false positive probability will be

$$\left((1/2)^2\right)^3 = (1/2)^6$$

and so on.

For this reason we use arrangements of 3, 4 and 6 BFEs in order to create a Bloom filter with smaller false positive probability. Figure 4.4 shows the schematic of a BF constructed of 5 BFEs. We use exactly the same idea we previously used in the BFE scheme (Figure 4.5) as we configure multiple BFE to act as *single bit* wide and *n x 16,384 bits* long bit array. This 5-element BF with a given a capacity of 5678 prefixes has a false positive probability of :

$$\left((1/2)^2\right)^5 = (1/2)^{10},$$

instead of $(1/2)^2$ of a single-Element BF with the same capacity. In our scheme some of the BF used in IP-sub engines storing only a part of the rules which means that the false positive probability for these BFs decreases more .

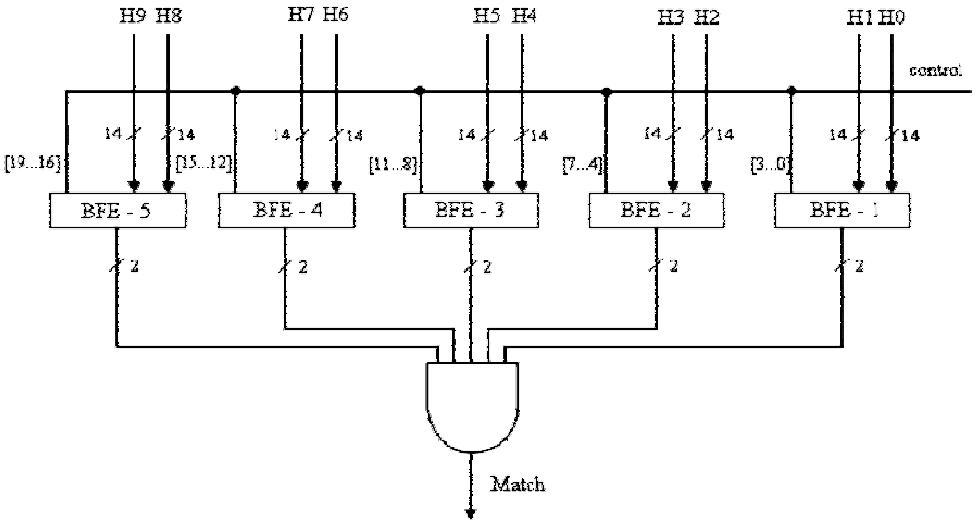


Figure 4.4: An example of a 5-element BF arrangement

4.3 Source and Destination IP Classifier Blocks

Source and Destination IP blocks are 2 identical BF based sub-engines. They use two separate BF compositions, "Primary IP BF" and "Secondary IP BF". Primary IP BF uses three dedicated BFs and handles prefixes: 16, 24, 32. The remaining prefix lengths are treated in the Secondary IP BF module which utilizes a single BF. A FIFO holds the results for both modules (33 results at the maximum). The FIFO's output is connected to the Permutation Engine module as described in the next section.

Figure 4.5: Organization of an IP Sub-Engine

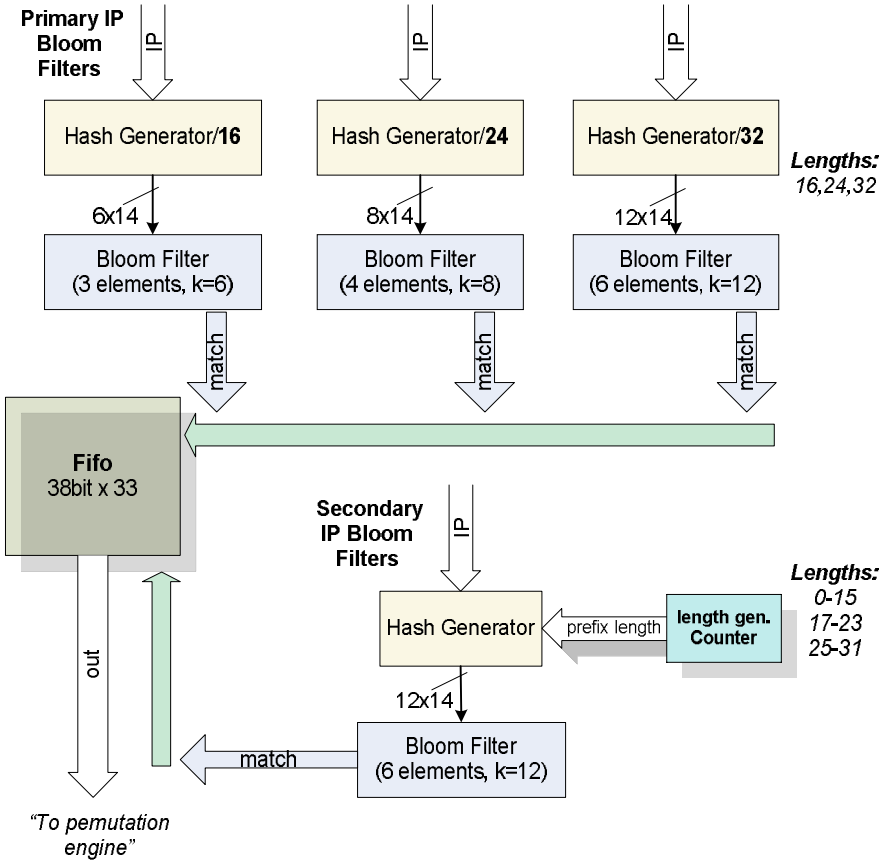


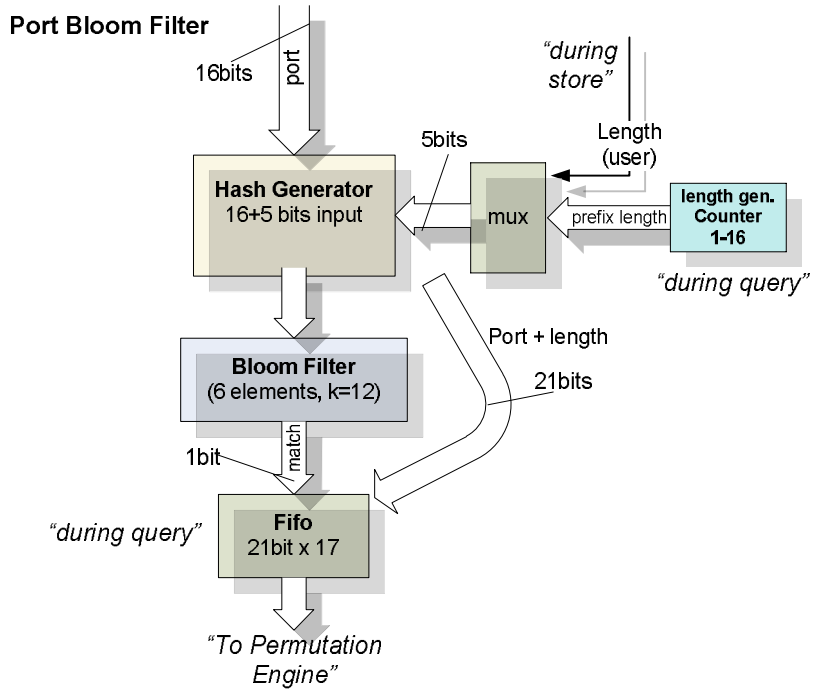
Figure 4.5 presents a general overview of an IP sub Engine. The three dedicated BFs are referred as the "*Primary IP BF"*. The remaining prefix lengths are treated in the *Secondary IP BF* module which utilizes a single BF. The *Secondary IP BF* is composed of a *Hash Generator* configured to address a 6 element BF (12x14=168 bit address) and a *counter (length gen. counter)* responsible for producing all the rest prefix lengths which are not examine in the *Primary IP BF*. A FIFO holds the results for both modules (33 results at the maximum). The FIFO's output is connected to the *Permutation Engine* module as described in Section [4.4].

4.4 Source and Destination Port Classifier Blocks

A port sub Engine works in a similar manner as an IP sub-engine; the only difference is that the port sub engine doesn't have dedicated Bloom Filters for certain prefix lengths, it has only one BF which treats all prefix lengths from 0 to 16. A single Port BF Sub-Engine is composed of a *Hash Generator* (hash

function), a Bloom Filter, a *Length Generator* (counter) and a simple *FIFO*. The hash function is properly configured to address our BF (12x14=168 bit address) using the Port information (16bits) combined with the corresponding prefix length (5bits). The *Length Generator* is used to provide all the possible prefix lengths during a given query. Each generated length is combined with the port information and examined through the BF. The matched results are stored in a FIFO which holds at most 17 results (16 possible prefix lengths plus wildcard). The FIFO's output is connected to the Permutation Engine module as described in next section.

Figure 4.6 Port Bloom Filter Organization

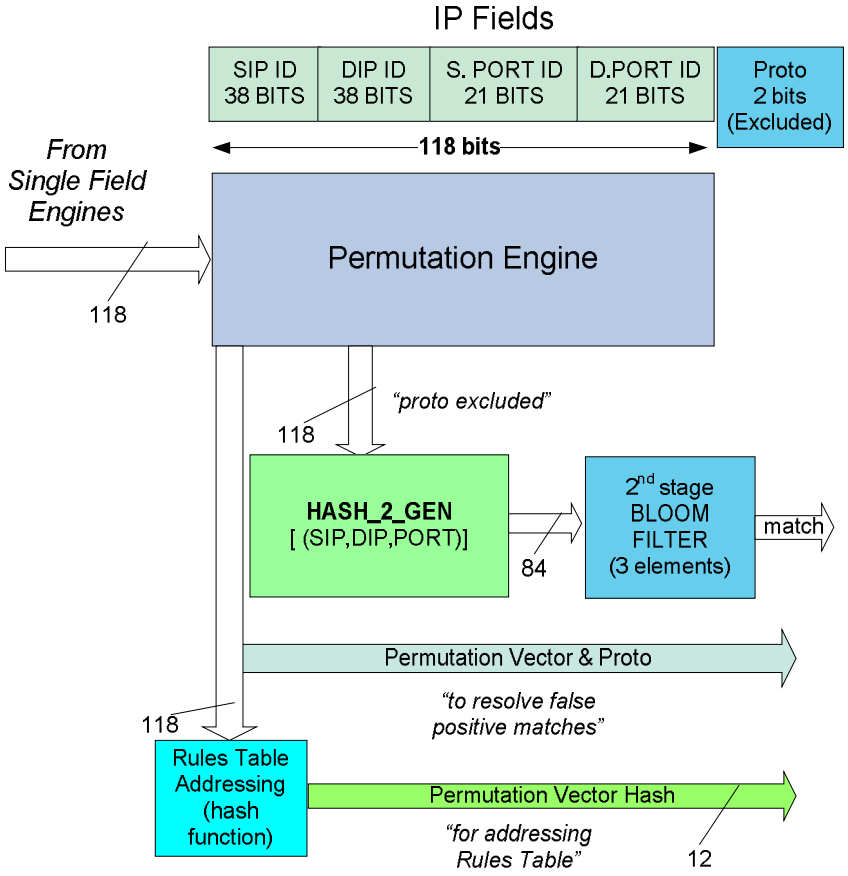


4.5 Permutation Engine Block

Figure 4.7 shows the organization of the second stage of the query modules. The permutation engine produces all the possible matching "rule vectors" (permutation vectors) which are examined in the second stage Bloom Filter. The input of the Permutation Engine is connected to the output of all the *Single Field Engines*. The *Hash_2_Gen* module is a hash function properly configured to address the Bloom Filter (2nd Stage BF). The permutation vector

information is needed at the last part of the procedure in order to compare the stored rule with the possible matching "permutation vector" as explained in Chapter 3.

Figure 4.7: Organization of Permutation Engine and 2nd stage BF



4.6 Rules Table Addressing Module

Rules Table Addressing Module hashes the permutation information in order to address the Rules Table memory in the last stage of the procedure. This hash function is similar to those used in BF addressing and uses an 118 bit input to produce a 12 bit address.

4.6.1 Hash_2_gen Module

Hash_2_gen Module as mentioned before is a H3 clash hash function which is properly configured to address the 2nd Stage BF. It takes a 118 bits input an produces a 84 bits hash output.

4.6.2 2nd Stage BF

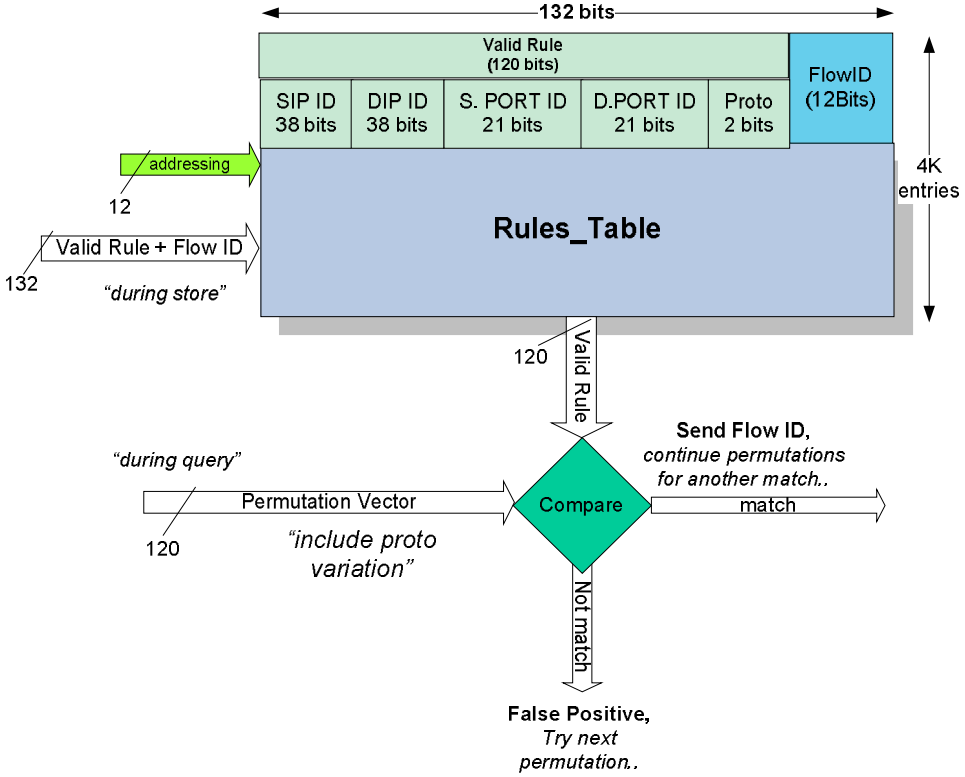
The 2nd Stage BF is a Bloom Filter data structure composed of 3 more simple Bloom Filter Elements (BFE). This BF examines each permutation vector to find out if it is actually valid. As explained in previous Chapter, there is always a false positive probability introduced by the BF which will be examined it the last part of the procedure.

4.7 Rules Table Block

Rules Table block locates the FlowID information from a possible matching permutation vector. It is composed of a *4K-entry Table* and a *comparator module* as explained in Section 3.2.9 The *4K-entry Table* is 12 bit addressable and 132 bits wide as the next Figure shows. Apart from the stored ruled information (120 bits) we reserve 12 bits for the FlowID.

The comparator module compares the 5-tuple information produced in the permutation process with this stored in each valid rule (including wildcard). If all fields are a match then we have a valid rule match.

Figure 4.8: Rules Table and Comparator Block



4.8 Implementation Analysis

4.8.1 Storage Requirements

One of our main concerns, when designing the proposed framework, was to be very memory efficient. In this subsection we present the storage requirements of 2sBFCE for all the generated filter sets. To calculate the total storage for 2sBFCE we measure the storage requirements of the single field Sub-engines, the storage of the second stage Bloom Filter and finally the size of the RulesTable. Table 4.1 shows the 2sBFCE memory requirements.

Table 4.1: Storage Requirements

Component	**No of Bloom Filter Elements**	**Memory Entries**	**Width (bits)**	**Total**
Source IP	19	16K	1bit	304Kbits
Destination IP	19	16K	1bit	304Kbits
Source Port	6	16K	1bit	96Kbits
Destination Port	6	16K	1bit	96Kbits
2nd Stage BF	6	16K	1bit	96Kbits
BF Total				**896Kbits**
Rules Table		**4K**	**132bits**	**528Kbits**
				1.424Mbits **178KBytes**

Our scheme employs only **178 KB** of on-chip SRAM, and as Table 4.4 demonstrates, it is the most memory efficient when comparing with other classification schemes.

4.8.2 Hardware Device's Cost & Performance

The proposed scheme has been Synthesized and Placed and Routed in FPGA technology and it works at 153MHz (6.56ns clock period). The target device was a Xilinx Virtex 4, **4vfx40ff672.** Table 4.2 shows the utilization of the device.

Table 4.2: 2sBFCE resource utilization

Selected Device : Xilinx Virtex 4, **4vfx40ff672** (speed grade -12)		
Number of Slices:	8067 out of 18624	43%
Number of Slice Flip Flops:	7570 out of 37248	20%
Number of 4 input LUTs:	14585 out of 37248	39%
Number of FIFO16/RAMB16s:	83 out of 144	57%

Tables 4.3a, 4.3b present the network performance (overall average performance and first result average performance respectively) of the optimized 2sBFCE which is by 16 cycles faster per query compared with the initial design. We assume 6.56ns (153 MHz) per clock cycle and it is counted both in Millions of Packets Per Second (Mpps) and in Gigabit Per Second (Gbps). If we assume the device processes only minimum-size IP-Packets (40 bytes) we get the worst case performance. Obviously, in case our classifier is employed in a real-world environment it will process IP packets with a mean size much greater than 100 bytes, as reported in [45] , and therefore it would easily be able to support network rates of **3 Gbps** or higher.

Table 4.3a: Overall Average performance for the optimized 2sBFCE.

Filter name	No of Rules	Overall Average performance (cycles per packet)	Mpps	Gbps (worst case)	Gbps (100 byte packet)
acl1	1192	56	2,72	0,87	2,18
ipc1	979	47	3,24	1,04	2,59
ipc2	618	35	4,36	1,39	3,48
fw1	975	42	3,63	1,16	2,90
fw2	653	36	4,23	1,36	3,39
fw3	692	34	4,48	1,43	3,59
fw4	1300	42	3,63	1,16	2,90
fw5	869	39	3,91	1,25	3,13
average	**909,75**	**41,38**	**3,78**	**1,21**	**3,02**

Table 4.3b: Average performance of the first result for the optimized 2sBFCE.

Filter name	No of Rules	Average performance of first result (cycles per packet)	Mpps	Gbps (worst case)	Gbps (100 byte packet)
acl1	1192	26	5,86	1,88	4,69
ipc1	979	21	7,26	2,32	5,81
ipc2	618	27	5,65	1,81	4,52
fw1	975	21	7,26	2,32	5,81
fw2	653	21	7,26	2,32	5,81
fw3	692	21	7,26	2,32	5,81
fw4	1300	28	5,44	1,74	4,36
fw5	869	21	7,26	2,32	5,81
average	**909,75**	**23,25**	**6,66**	**2,13**	**5,32**

In order to make the comparison with other schemes easier we normalize the throughput at 100 Mhz as the compared schemes are implemented using different technologies. Furthermore we introduce the ratio ***Throughput @ 100MHz/Mbyte*** to show how memory efficient is each scheme.

According to Table 4.4, it is clear that, despite the fact that RFC has the best throughput, its performance is based on large memory utilization, while it supports at most 1700 rules. On the other hand ABV does provide the highest number of Mpps/Mbytes, but it supports 80% less rules than our scheme. Considering that 4K rule set size or more is the upcoming standard, our scheme is one of the best proposed schemes along with B2PC (ASIC) [1]

Table 4.4 : Summary of Classification Schemes

Scheme	Working Frequency (MHz)	Number of Rules	Storage Requirements (Number of memories)	Throughput (Mpps)	Throughput (Mpps) @ 100Mhz	Throughput (Mpps) @ 100Mhz Per MByte
2sBFCE last result (average)	**153**	**4k**	**178Kbytes**	**3.78**	**2.47**	**14.2**
2sBFCE first result (average)	**153**	**4k**	**178Kbytes**	**6.66**	**4.35**	**25.0**
BV[27]	33	512	640KB (5)	1	3	4,8
ABV[28]	*"based on BV"*					26,5
RFC[29]	125	1700	976 KB (2) + 15,6 MB (2)	30	24	14,5
B2PC (FPGA, off chip memory) [1]	75	3300	540 KB (4)	2,7	3,6	6,8
B2PC (ASIC, on chip memory) [1]	400	3300	540 KB (4)	42,5	10,6	20,1

Therefore, we claim that our scheme provides the optimal bandwidth-to-memory approach, for any device that supports a few thousand rules. Obviously, if performance is the only issue RFC would be more appropriate, or

for embedded, very low-memory, devices the low-memory scheme of [51] would probably be preferable. Moreover, for devices supporting relatively small filter sets ABV seems to be the natural case.

4.8.3 Performance Memory efficiency and parallelization

As 2sBFCE uses very low amounts of memory it gives us the capability of parallelization if we utilize multiple cores per chip. We easily managed to fit a dual-core implementation of our scheme in a ***Xilinx Virtex 4 (4vfx60ff672-12)*** device which almost doubled the throughput at 151 Mhz while maintaining the same latency. Table 4.5 shows the utilization of our dual-core 2sBFCE version.

Table 4.5: 2sBFCE dual core resource utilization

Selected Device : Xilinx Virtex 4, **4vfx40ff672** (speed grade -12)		
Number of Slices:	8067 out of 18624	43%
Number of Slice Flip Flops:	7570 out of 37248	20%
Number of 4 input LUTs:	14585 out of 37248	39%
Number of FIFO16/RAMB16s:	83 out of 144	57%

4.8.4 IPv6 Support

As Bloom Filters are hash-based data structures they could easily be adjusted to support inputs of a different length. The IPv4 addresses are 32-bits long and the IPv6 addresses are defined to be 128-bit. Thus, with slight modifications in our hash functions input, our scheme can easily support IPV6 without further modifications in BFs arrangement.

Chapter 5

Contributions and Future Work

5.1 Summary of Contributions

We have extensively studied packet classification problem in 5-d in comparison with the related literature and worked on an efficient hardware scheme. We followed the decomposition approach of multi-field classification rules into internal single-field rules which then are combined using multi-level bloom filters.

We designed, simulated and proposed a classification solution that exploits 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 3,4 (IP/TCP headers) .

In general, the efficiency of the proposed scheme comes from the fact that it takes advantage of all the specific features of the current real-world filter databases, while it has been designed from the beginning for efficient hardware implementation. The algorithm proposed may be less efficient in throughput than the other algorithms found in the literature, in worst-case scenarios, but the hardware implementation of this scheme is the most efficient one demonstrated so far when memory requirements, number of rules and bandwidth are all taken into account. Moreover, its hardware cost (in terms of silicon covered) is very low making it an even more promising approach for low cost classification engines.

In Chapter 3 we propose an innovative classification scheme for the IP 5-tuple. The proposed solution, *2 stage Bloom Filter Classification Engine (2sBFCE),* approaches the packed classification problem in a decomposed manner. Hence single field matches of each packet field are combined to identify the matching rule. 2sBFCE uses independent Bloom Filters to provide efficient single field (all prefix) matches of 5d classification rules. In addition it represents 5d classification rules using a second-level Bloom Filter scheme so as to provide fast and efficient set membership queries.

In Chapter 4 we describe in more detail the overall hardware scheme and how it is implemented using FPGA technology. Our FPGA implementation of 2sBFCE uses on-chip BRams and operates at 153 MHZ while utilizes only 178KB of memory. That gives us a throughput of **3.78 Mpps or 3 Gbps** while supporting up to 4k rules. As 2sBFCE uses very low amounts of memory it gives us the capability of parallelization utilizing multiple cores per chip. We easily managed to fit a dual core implementation of our scheme in a ***Xilinx Virtex 4 (4vfx60ff672-12)*** device which doubled the throughput maintaining the same latency.

5.2 Publications

We had the following two publications as full papers in IEEE ICC and IEEE FCCM Conferences:

- Antonis Nikitakis, Ioannis Papaefstathiou: *"A memory-efficient FPGA-based classification engine",* IEEE ICC 2008
- Antonis Nikitakis, Ioannis Papaefstathiou: *"A Multi Gigabit FPGA-based 5-tuple classification system",* IEEE FCCM 2008

5.3 Future Work

Even though our packet classification solution, not only supports a few thousand rules, but it is scalable to larger rules sets, its performance is very dependent on rules sets characteristics. It would be rather interesting if we tried to utilize a 3-level BF architecture in order to achieve a more stable performance irrelevant to the rules set's characteristics. A **3-level architecture** apart from the single-field searches will support set membership queries using two fields and then in the next level covering all the 5 fields. This means that the Source IP and Destination IP would have to "match" together as well as the Source and Destination Port before we try matches on 5 fields. In that way we will radically reduce the total number of permutation in those cases we have many single field results as most of the permutations will be "excluded" due to this intermediate filtering level.

References

[1] Vassilios Papaefstathiou "*Design and Implementation of Network Packet Classification Engines",* Master's Thesis, March 2005 Heraklion, Greece

[2] Marios A. Eliofotou *"Hardware for IPv6 Longest Prefix Matching",* Diploma Thesis, Technical University Of Crete, Chania, Greece

[3] Pankaj Gupta "*ALGORITHMS FOR ROUTING LOOKUPS AND PACKET CLASSIFICATION*". Ph.D. Thesis, Stanford University, December 2000

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

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

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

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

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

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

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

[11] 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.

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

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

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

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

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

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

[18] 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.

[19] 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

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

[21] 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.

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

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

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

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

[26] 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.

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

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

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

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

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

[32] 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

[33] 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

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

[35] XilinxVirtex-4 Multi-Platform FPGA, http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/virtex4/index.htm

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

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

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

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

[40] 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

[41] 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

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

[43] 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.

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

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

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

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

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

[49] I. Papaefstathiou, V. Papaefstathiou, *"An innovative low-cost Classification Scheme for combined multi-Gigabit IP and Ethernet Networks",* in IEEE ICC'06, June 2006

[50] S. Dharmapurikar, P. Krishnamurthy, D.E. Taylor, *"Longest Prefix Matching Using Bloom Filters"*, in ACM SIGCOMM'03, August 2003

[51] Xuehong Sun, S.K. Sahni, Y.Q. Zhao, "*Packet classification consuming small amount of memory"*, in IEEE/ACM Transactions on Networking, Volume: 13, Issue: 5, pp 1135- 1145, Oct. 2005

[52] Sarang Dharmapurikar, Michael Attig and John Lockwood, "*Design and Implementation of a String Matching System for Network Intrusion Detection using FPGA-based Bloom Filters*". Technical Report, Washington University in Saint Louis, March 2004.

[53] Datasheets Xilinx Virtex-4 Series FPGAs. http://www.xilinx.com/virtex4. Winter 2005.