Report Number: WUCSE-2004-40

2004-07-28

Models, Algorithms, and Architectures for Scalable Packet Classification

David Edward Taylor and Jonathan S. Turner

The growth and diversification of the Internet imposes increasing demands on the performance and functionality of network infrastructure. Routers, the devices responsible for the switch-ing and directing of traffic in the Internet, are being called upon to not only handle increased volumes of traffic at higher speeds, but also impose tighter security policies and provide support for a richer set of network services. This dissertation addresses the searching tasks performed by Internet routers in order to forward packets and apply network services to packets belonging to defined traffic flows. As these searching tasks must be performed for each packet... Read complete abstract on page 2.

Models, Algorithms, and Architectures for Scalable Packet Classification

David Edward Taylor and Jonathan S. Turner

Complete Abstract:

The growth and diversification of the Internet imposes increasing demands on the performance and functionality of network infrastructure. Routers, the devices responsible for the switch-ing and directing of traffic in the Internet, are being called upon to not only handle increased volumes of traffic at higher speeds, but also impose tighter security policies and provide support for a richer set of network services. This dissertation addresses the searching tasks performed by Internet routers in order to forward packets and apply network services to packets belonging to defined traffic flows. As these searching tasks must be performed for each packet traversing the router, the speed and scalability of the solutions to the route lookup and packet classification problems largely determine the realizable performance of the router, and hence the Internet as a whole. Despite the energetic attention of the academic and corporate research communities, there remains a need for search engines that scale to support faster communication links, larger route tables and filter sets and increasingly complex filters. The major contributions of this work include the design and analysis of a scalable hardware implementation of a Longest Prefix Matching (LPM) search engine for route lookup, a survey and taxonomy of packet classification techniques, a thorough analysis of packet classification filter sets, the design and analysis of a suite of performance evaluation tools for packet classification algorithms and devices, and a new packet classification algorithm that scales to support high-speed links and large filter sets classifying on additional packet fields.

2004-40

Models, Algorithms, and Architectures for Scalable Packet Classification, Doctoral Dissertation, August 2004

Authors: Taylor, David E.

July 28, 2004

Abstract: The growth and diversification of the Internet imposes increasing demands on the performance and functionality of network infrastructure. Routers, the devices responsible for the switching and directing of traffic in the Internet, are being called upon to not only handle increased volumes of traffic at higher speeds, but also impose tighter security policies and provide support for a richer set of network services. This dissertation addresses the searching tasks performed by Internet routers in order to forward packets and apply network services to packets belonging to defined
traffic flows. As these searching tasks must be performed for each packet traversing the router, the speed and scalability of the solutions to the route lookup and packet classification problems largely determine the realizable performance of the router, and hence the Internet as a whole. Despite the energetic attention of the academic and corporate research communities, there remains a need for search engines that scale to support faster communication links, larger route tables and filter sets, and increasingly complex filters. The major contributions of this work include the design and analysis of a scalable hardware implementation of a Longest Prefix Matching (LPM) search engine for route lookup, a survey and taxonomy of packet classification techniques, a thorough analysis of
packet classification filter sets, the design and analysis of a suite of performance evaluation tools for packet classification algorithms and devices, and a new packet classification algorithm that scales to support high-speed

WASHINGTON UNIVERSITY

SEVER INSTITUTE OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

MODELS, ALGORITHMS, AND ARCHITECTURES FOR SCALABLE PACKET CLASSIFICATION

by

David Edward Taylor, M.S.Co.E., M.S.E.E., B.S.Co.E., B.S.E.E.

Prepared under the direction of Dr. Jonathan S. Turner

A dissertation presented to the Sever Institute of
Washington University in partial fulfillment
of the requirements for the degree of

Doctor of Science

August, 2004

Saint Louis, Missouri

WASHINGTON UNIVERSITY

SEVER INSTITUTE OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

ABSTRACT

MODELS, ALGORITHMS, AND ARCHITECTURES FOR SCALABLE PACKET CLASSIFICATION

by David Edward Taylor

ADVISOR: Dr. Jonathan S. Turner

August, 2004

Saint Louis, Missouri

The growth and diversification of the Internet imposes increasing demands on the performance and functionality of network infrastructure. Routers, the devices responsible for the switching and directing of traffic in the Internet, are being called upon to not only handle increased volumes of traffic at higher speeds, but also impose tighter security policies and provide support for a richer set of network services. This dissertation addresses the searching tasks performed by Internet routers in order to forward packets and apply network services to packets belonging to defined traffic flows. As these searching tasks must be performed for each packet traversing the router, the speed and scalability of the solutions to the route lookup and packet classification problems largely determine the realizable performance of the router, and hence the Internet as a whole. Despite the energetic attention of the academic and corporate research communities, there remains a need for search engines that scale to support faster communication links, larger route tables and filter sets, and increasingly complex filters. The major contributions of this work include the design and analysis of a scalable hardware implementation of a Longest Prefix Matching (LPM) search engine for route lookup, a survey and taxonomy of packet classification techniques, a thorough analysis of packet classification filter sets, the design and analysis of a suite of performance evaluation tools for packet classification algorithms and devices, and a new packet classification algorithm that scales to support high-speed links and large filter sets classifying on additional packet fields.

Soli Deo Gloria

to God alone be the glory

Contents

List of Tables

List of Figures

Acknowledgments

I not only use all the brains that I have, but all that I can borrow.
Woodrow Wilson, 28th President of the United States of America

My humble measure of intelligence and creativity are not solely responsible for the "novel contributions to the body of knowledge" contained in this dissertation. I have been blessed many times over with loving and supportive family and friends, and a long line of dedicated teachers and mentors. The fruit of this dissertation is a direct result of their selfless acts on my behalf. While it is impossible (and overly tedious) to thank everyone, I will attempt to make mention of those most directly involved in my graduate education and those who kept me sane and happy throughout this adventure.

I would like to start by thanking those serving on my dissertation committee. Inexpressible thanks go to my research advisor, Dr. Jonathan S. Turner, for his tremendous patience and diligent mentorship. I sincerely appreciate the academic freedom he provided throughout my graduate studies, especially early in my studies when I was a rather naïve researcher. His consummate emphasis on clarity and understanding nurtured and encouraged me to produce the highest quality research that I could. Were it not for my academic advisor, trusted friend, and savvy agent, Dr. William D. Richard, I most likely would not have become a graduate student at Washington University. I will forever be thankful for his selfless actions to provide me with wonderful opportunities to learn and contribute. His valued advice always goes well beyond the realms of academics and research; he provides truly useful wisdom. I would like to thank Dr. John Lockwood for offering valuable suggestions and insight, supporting a portion of my graduate studies, involving me in the early development of the Field-programmable Port eXtender (FPX), and demonstrating a commitment and enthusiasm for making concepts "real" in hardware. I would like to thank Dr. Robert E. Morley for serving on my proposal committee. I will always remember his engineering mantra, "there's no such thing as magic", and his probing question about the status of my design projects, "would you get on the airplane?" It has also been an honor to have Dr. Fred U. Rosenberger as a professor and member of my committee. I will always value his insight into fundamental aspects of digital circuit design, healthy skepticism of performance claims (I highly recommend viewing his "Gallery of Perpetual Motion"), and wise advice to employ due caution when designing *anything*. Finally, I would like to thank Dr. Daniel R. Fuhrmann for serving on my committee and offering his insight on short notice.

A number of other Washington University faculty and Applied Research Laboratory staff have generously provided their wisdom, encouragement, and assistance. Specifically, I would like to thank John DeHart for his enthusiasm, patience, and invaluable assistance with verification and performance measurement of the Fast IP Lookup (FIPL) search engine. I am an appreciative beneficiary of his vast engineering talent. Many thanks go to Dave Zar for providing invaluable assistance with Mentor Graphics and Xilinx CAD tools, answering many VHDL questions, and giving me my first research job, publishing opportunity, and conference presentation experience.

My graduate student experience was significantly enhanced by the wonderfully talented group of graduate students in ARL. I would like to sincerely thank Jeyashankher Ramamirtham for being an amiable office-mate and tolerating my numerous interruptions and requests for help with C++ code debugging. I would like to thank Ed Spitznagel for offering his invaluable insight to countless discussions on packet classification techniques, providing assistance with filter set parsing, and being a willing and responsive test case for my *ClassBench* tools. Many thanks to (Dr.) Tilman Wolf for participating in many lively discussions over coffee and fostering a vibrant "culture" in the laboratory and department. Likewise, I would like to thank (Dr.) Dan Decasper and (Dr.) Ralph Keller for fostering a *very* vibrant "culture" when they were at Washington University. Thanks to Anshul Kantawala and the rest of the lunchtime crowd for many enlightening discussions/arguments. Finally, thanks to all the ARL "fools" (you know who you are) for making graduate student life more fun than it ought to be.

I also would like to acknowledge William Eatherton and Zubin Dittia as the developers of the *Tree Bitmap* algorithm. Their design efforts and analyses made a portion of this research possible. I would like to acknowledge Todd Sproull as the developer of the control software and web interfaces for the FIPL search engine. I also would like to thank Tucker Evans and Ed Spitznagel for their contributions to the FIPL Memory Manager software. I would like to thank Sarang Dharmapurikar and Praveen Krishnamurthy for introducing me to Bloom filters and inviting me to work with them in developing the "Longest Prefix Matching using Bloom Filters" technique. I would like to thank Venkatachary Srinivasan, William Eatherton, and others for making several real filter sets available for study.

I would also like to send my sincere thanks "across the pond" to Andreas Herkersdorf and other members of the Network Processor Hardware team at the IBM Zurich Research Laboratory for a rewarding educational and cultural experience. I thoroughly enjoyed my summer in Switzerland and gained a new level of respect for my peers in the international research community.

On a more personal note, I would like to offer my most heartfelt thanks to my wife, Sara Jane Taylor. Her love and companionship have brought me immeasurable joy over the past three years. It has been a tremendous blessing to have someone to empathize with me in the challenges of a doctoral program. I would like to thank my parents who have offered consummate support and encouragement not only in my five years as a graduate student, but throughout my 24 years of formal education. It is impossible to list all that they have done for me, for like God's love, so much of it goes unnoticed and unacknowledged. So, I offer my thanks for everything I have forgotten to thank them for, and specifically: helping me with countless homeworks, lauding my achievements, offering consolation in my defeats, sending me to college, and being shining examples of loving parents. I also offer heartfelt thanks to my dear friends who have been an essential source of inspiration, support, refreshment, and guidance.

Our scientific power has outrun our spiritual power. We have guided missiles and misguided men.
Martin Luther King Jr.

I offer my eternal thanks and praise to the Lord Jesus Christ for the saving love and mercy that He demonstrates every day of my life. He has redeemed my life from countless pits, continues to freely extend His grace to me, and showers me with undeserved blessings. All of the contributions and novel ideas in this dissertation are products of His grace in response to my prayers when I reached the end of my natural ability. I also would like to thank my brothers and sisters in Christ at New City Fellowship of St. Louis for their companionship, discipleship, and prayers.

David Edward Taylor

Washington University in Saint Louis
August 2004

Preface

The Internet - a conglomeration of military, academic, and commercial computer communication networks - is arguably the most pervasive technology in recent history. Started as an experimental project by the Defense Advanced Research Projects Agency (DARPA) of the United States Department of Defense in 1973, the Internet continues to expand and diversify [1]. The scope of its use has moved beyond ubiquitous communication and dissemination of information to include new commercial, academic, and private-sector services. Originally the brainchild of the research community and a novelty for the technology hobbyist, the Internet has radically transformed the way the world communicates. It has become essential infrastructure for the global economy, entrenched itself in the cultures of industrialized nations, and penetrated the most remote locations on earth.

While statistics regarding Internet size and use are notoriously difficult to pin down, even the rough estimates are staggering. As of January 2004, there were approximately 233 million Internet hosts [2]. A host refers to any device communicating over the Internet: personal computers, workstations, servers, Personal Digital Assistants (PDAs), etc. At that time, the United States accounted for 144 million hosts with over seven thousand Internet Service Providers (ISPs). Roughly 945 million people use the Internet world-wide, and the number of users is projected to exceed 1.1 billion in 2005 [3]. Spending for online content increased to \$1.56 billion in 2003 [4], and consumers transacted over \$2.2 billion over the Internet in the one week period following the Thanksgiving holiday in 2003 [5]. These figures could easily double in the next few years as the Internet penetrates the two most populous countries in the world - India and China.

The growth and diversification of the Internet imposes increasing demands on the performance and functionality of network infrastructure. The Internet may be thought of as a global postal system for delivering digital letters, or packets; thus, the task of packet forwarding is akin to sorting mail. In the context of the Internet, the challenge is that packets are transmitted at roughly the speed of light and arrive at rates exceeding a hundred million packets per second. Furthermore, routers, the devices responsible for the switching and directing of traffic in the Internet, may need to sort packets into thousands of different “bins” by consulting a complex directory containing tens of thousands of entries. Routers are being called upon to not only handle increased volumes of traffic at higher speeds, but also impose tighter security policies and provide support for a richer set of network services. A critical issue in realizing the latter set of goals is identifying the traffic belonging to a particular flow or set of flows. A flow may be thought of as the communication traffic generated by a specific application traveling between a specific set of hosts or subnetworks. Flow identification is computationally intensive and the task is complicated by the continually increasing volume and speed of traffic traversing routers.

In this dissertation, we address the packet forwarding and flow identification problems, more commonly known as route lookup and packet classification. Due to their fundamental role in the functionality and performance of Internet routers, both problems are well-studied. Despite the energetic attention of a broad community of researchers in industry and academia, there remains a need for good solutions. In this context, a solution's "goodness" is evaluated along the classical engineering criteria of performance, size, cost, and power consumption. The contributions of this work include a high-performance implementation of a route lookup search engine, an in-depth study of the filter sets used to classify packets, a suite of performance evaluation tools, and a new algorithm for packet classification that scales to larger filter sets and more complex filters.

The value of this work goes beyond prototypes, research tools, and algorithms of academic interest. A number of companies are beginning to offer packet classification search engines as products, and the industry is also gaining interest and investing in algorithmic solutions to the packet classification problem. According to a leading market analyst, the search engine device market grew 14% from $83 million in 2002 to $95 million in 2003 [6]. More profound than the total market growth is that the leading company offering algorithmic search engines gained 11% market share while the leading TCAM vendor lost 18% market share. Ternary Content Addressable Memory (TCAM) is a memory technology that searches all entries in the filter set in a single cycle. This strategy results in fast packet classification, but the devices are extremely expensive and power hungry.

Chapter 1

Introduction

Computer Science is no more about computers than astronomy is about telescopes.
Edsger W. Dijkstra

The world is in the midst of a major paradigm shift in the role and importance of communications technology. Many contemporary historians have already dubbed this the "Information Age". Codified by the protocols produced by the DARPA Internet Architecture project begun in 1973, the Internet has emerged as a global communications service of ever increasing importance. The expanding scope of Internet users and applications requires network infrastructure to carry larger volumes of traffic, tightening already challenging performance constraints. This dissertation addresses the searching tasks performed by Internet routers in order to forward packets and apply network services to packets belonging to a particular traffic flows. As these searching tasks must be performed for each packet traversing the router, the speed and scalability of the solutions to these problems largely determine the realizable performance of the router, and hence the Internet as a whole.

1.1 State of the Internet

The Internet refers to the global "network of networks" that utilizes the suite of internetworking protocols developed by the DARPA Internet Architecture project initiated in 1973. The original aim of this project was to enable communication across the original ARPANET and the ARPA packet radio network, but the original architects were tasked with developing protocols to enable communication across a wide variety of heterogeneous networks [1]. Due to the nature of the ARPA packet radio network and the set of foreseeable applications, the protocols employ datagrams, or packets, as the fundamental unit of communication, and thus the Internet is a connection-less packet-switched network. The use of datagrams endowed the protocols with a simplicity and flexibility that is largely responsible for the tremendous growth and development that the Internet has enjoyed.

The building blocks of the Internet are essentially networks, each consisting of combinations of possibly heterogeneous hosts, links, and routers. Figure 1.1 provides a simple example of the Internet architecture. Hosts produce and consume packets, or datagrams, which contain chunks of data - a piece of a file, a digitized voice sample, etc. Hosts may be personal computers, workstations, servers, Personal Digital Assistants (PDAs), IP-enabled mobile phones, or satellites. 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 link technologies such as radio, microwave, or infrared. There are a variety of strategies for allocating links in a network. These strategies often take into consideration bandwidth and latency requirements of applications, geographical location, deployment and operating costs. The fundamental role of routers is to switch packets from incoming links to the appropriate outgoing links depending on the destination of the packets. Note that a packet may traverse many links, often called hops, in order to reach its destination. Due to the transient nature of network links (failure, congestion, additions, removals), routing protocols allow the routers to continually exchange information about the state of the network. Based on this information, routers decide on which link to forward packets destined for a particular host, network, or subnetwork. Note that the dynamic nature of the routing protocols allows packets from a single host addressed to a common destination to follow different paths through the network.

The original Internet protocol suite was comprised 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 with each packet transmitted through the network, much like the forwarding and return addresses on a letter. IP specifies the format of this information which is prepended to the content of each packet. The information prepended by each protocol is referred to as a packet header and the data content of the packet is referred to as the payload. 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) which assigns 32-bit addresses to Internet hosts. As shown in Figure 1.2, the IPv4 header prepended to packets includes the IP address of the source and destination host. For the purpose of our discussion, the other IPv4 header field of interest is the *protocol* field which identifies 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.

Rather than individually assign addresses to every host, 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; the remaining set of bits is called the host address. This allocation strategy provided decentralized control of address allocation; each organization was free to make allocation decisions for the addresses within its assigned block. As shown in Figure 1.3, IPv4 addresses were originally

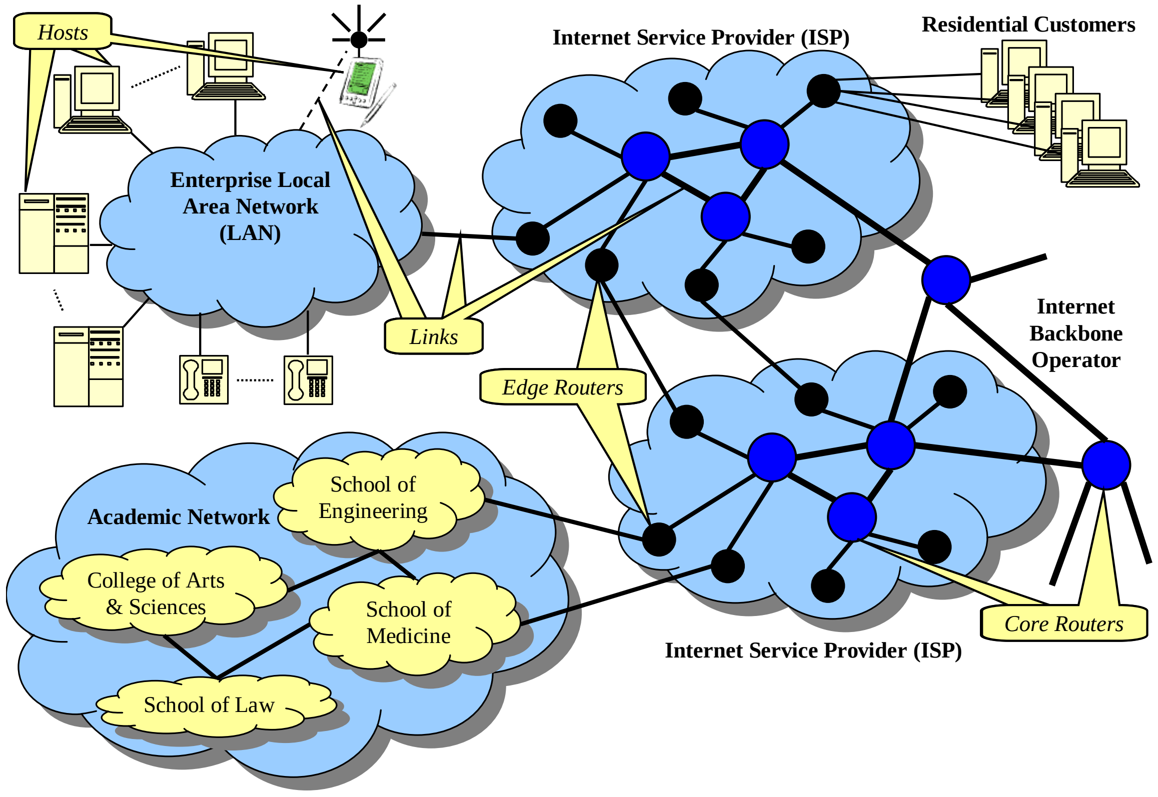

Hosts
Enterprise Local Area Network (LAN)
Links
Internet Service Provider (ISP)
Residential Customers
Internet Backbone Operator
Edge Routers
Academic Network
School of Engineering
College of Arts & Sciences
School of Medicine
School of Law
Core Routers
Internet Service Provider (ISP)

Figure 1.1: Simple diagram of Internet architecture.

assigned in blocks of three sizes: Class A (16 million hosts), Class B (64 thousand hosts), and Class C (254 hosts). Note that there are also blocks of Class D addresses for multicast (one-to-many transmission) and reserved Class E addresses. Most organizations which required a larger address space than Class C were allocated a block of Class B addresses, even though their network consumed only a fraction of the addresses. This waste of available address space combined with the explosive growth of the Internet prompted concerns over the impending shortage of unassigned IP addresses. Classless Inter-Domain Routing (CIDR) was introduced in order to prolong the life of IPv4 [7]. CIDR essentially allows a network 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) [8]. Among other issues, the designers of IPv6 addressed the address space issue via the use of 128-bit addresses. Despite the relief provided by CIDR, adoption of IPv6 is probable given the continued increase in the number of Internet hosts and deployment initiatives by influential research and commercial groups [9].

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

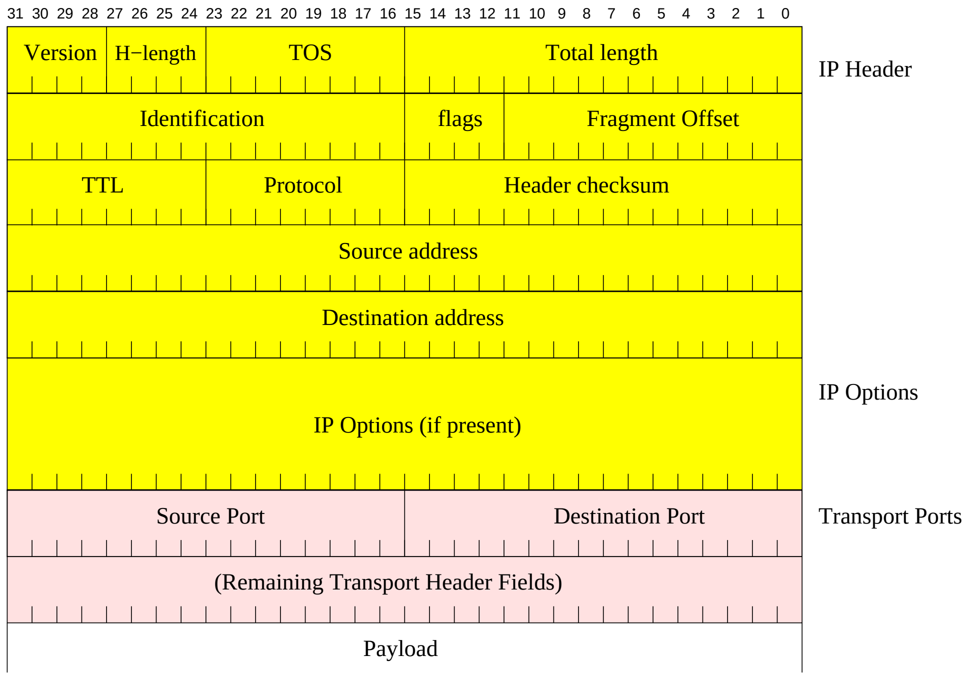


Figure 1.2: Format of Internet Protocol Version 4 (IPv4) packet headers with appended transport protocol header fields.

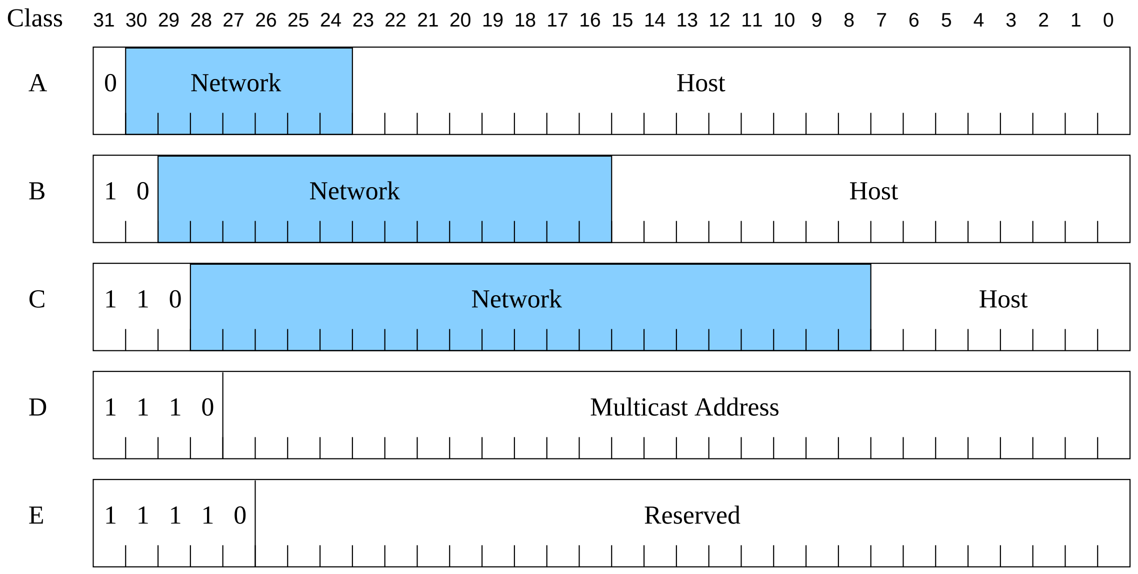


Figure 1.3: Internet Protocol Version 4 (IPv4) address space allocation.

use of small acknowledgment packets transmitted from the destination host to the source host, TCP detects packet loss and paces the transmission of packets in order to adjust to network congestion. When the source host detects 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 initial development of TCP, a third protocol, the User Datagram Protocol (UDP), was added to the original suite in order to provide additional flexibility. UDP essentially allows applications or higher level protocols to dictate transmission behavior. For example, a streaming video application may wish to ignore transient 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 1.2. In order to provide services to unknown hosts, servers must have static “contact ports” for each application. Port numbers for widely-used applications fall in the range of well-known *system* ports which are assigned by the Internet Assigned Numbers Authority (IANA). Prior to 1993, the well-known port numbers were in the range $[0 \dots 255]$ while port numbers $[256 \dots 1023]$ were used in Unix systems for Unix-specific services. Since 1993, port numbers in the range $[0 \dots 1023]$ form the set of well-known *system* port numbers managed by IANA. A “living document” of *system* port number assignments is available at `http://www.iana.org/assignments/port-numbers`. For applications where either TCP or UDP may be used, port number assignments are typically identical. Unlike servers, clients only need to guarantee that running applications use free port numbers. The range of port numbers that may be freely assigned by clients are referred to as ephemeral *user* ports due to their short-lived and unmanaged nature. The set of *user* port numbers span the range $[1024 \dots 65535]$. IANA does maintain a list of *registered* user port numbers in the range $[1024 \dots 49151]$ for popular applications which do not have an assigned *system* port.

1.2 The “Next Generation” Internet

While the protocols produced by the Internet Architecture project achieved the original goals set forth by DARPA and the pioneering group of researchers, the use of datagrams also presents challenges for those striving to deploy the next-generation of Internet services, particularly real-time services such as Internet telephony and video conferencing. It is important to note that the choice of datagrams and packet-switching represents a significant departure from the circuit-switched networks originally developed and deployed by the telecommunications industry. While the Internet protocols simplify the task of combining heterogeneous networks, the use of packet-switching complicates the provision of bandwidth and quality of service guarantees. As mentioned above, packets flowing between a fixed set of hosts may take different paths through the network. Due to the heterogeneous nature of the Internet, packets following different paths will likely experience different hop counts and congestion resulting in unpredictable latency and bottleneck link capacity. Circuit-switched networks allow data to flow along a fixed path, offering predictable performance. The major drawback of circuit-switching is the need to negotiate an end-to-end path through the network. In the case of the Internet, this would require coordination across many heterogeneous networks operated by independent parties with potentially competing interests.

Enabling quality of service and real-time performance guarantees are just a couple of the challenges facing the community architecting the “next-generation” Internet. As the Internet becomes increasingly essential infrastructure for the global economy, security is a major concern. Due to their roots in academic research, many network protocols were developed and implemented with little if any consideration of security issues. As a result, many academic and commercial institutions have suffered from destructive network intrusions by hackers, viruses, and worms. Those holding a vested interest in the security of the Internet now find themselves in a perpetual “arms race” with nefarious programmers. Furthermore, IP has essentially become a victim of its own popularity. The amount of investment in the IP infrastructure by Internet Service Providers (ISPs) has yielded significant resistance to changing the architecture. This hardening of the Internet architecture also presents a significant challenge to realizing the “next-generation” Internet.

Despite concerns over security and ossification of the Internet protocols, many in the research community have put forth grand visions of the “next-generation” Internet. While specifics invariably differ, common goals include: retaining the flexibility provided by IP while enabling the performance guarantees made available by circuit-switching, providing a level of security that warrants greater economic reliance, and enabling more rapid development and deployment of services. Some go so far as to set forth the goal that the Internet become reliable enough to support the air traffic control system [10].

1.3 The Packet Classifi cation Problem

In a circuit-switched network, the task of identifying the traffic associated with a particular application session between two hosts or subnetworks is trivial from the router’s perspective. A simple, fixed-length flow identifier can be prepended to each unit of data that identifies the established end-to-end connection. For each unit of data, a router simply performs an exact match search over a table containing the flow identifiers for established connections. The table entries for each flow identifier contain the output link on which to forward the data and may also specify quality of service guarantees or special processing the router should perform.

The flow identification task in a packet-switched network is significantly more challenging. The primary task of routers is to forward packets from input links to the appropriate output links. In order to do this, Internet routers must consult a *route table* containing a set of network addresses and the 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 overly difficult if the Internet Protocol (IP) address hierarchy were strictly maintained. A simple lookup in three tables, one for each Class of networks, would be sufficient. The wide adoption of CIDR allows the network addresses in route tables to be any size. Performing a search

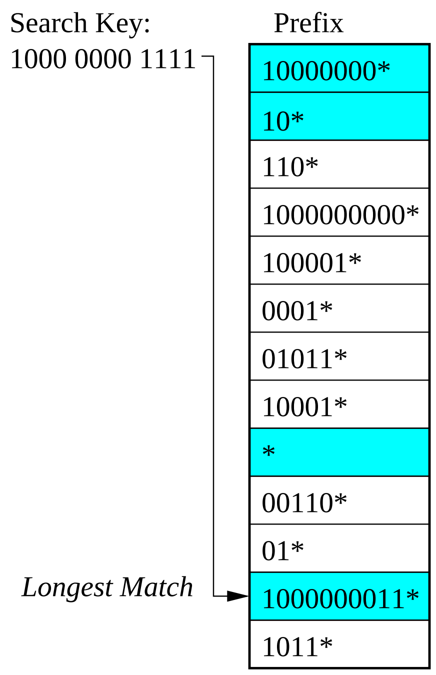


Figure 1.4: Example of Longest Prefix Matching for a 12-bit search key; all shaded prefixes match the key, but $1000000011*$ is the longest matching prefix.

in 32 tables, one for each possible network address length, for every packet traversing the router is not a viable option. If we store all the variable-length network addresses in a single table, a route lookup requires finding the longest matching prefix (network address) in the table for the given destination address.

Stated formally, a prefix is a subset of initial bits of a key value, the IP destination address in the case of route lookups. By definition, key values that share a common prefix have the same contiguous subset 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. Each prefix p_i can be thought of as the combination of a b-bit key and a corresponding b-bit mask which identifies the valid bits in the key. By definition, the mask is contiguous in LPM; i.e. the most significant invalid bit in the mask must be succeeded by invalid bits. Prefixes can be succinctly 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 12-bit search key is provided in Figure 1.4. Note that the four shaded prefixes match the search key, but $1000000011*$ is the longest matching prefix. The throughput of an Internet router largely depends upon the speed at which it can perform Longest Prefix Matching (LPM).

If an Internet router is to provide more advanced services than packet forwarding, it must perform finer grained flow identification. In the Internet context, the process of identifying the packets belonging to a specific application session or group of sessions between a source and destination

Table 1.1: Example filter set of 16 filters classifying on four fields; each filter has an associated flow identifier (*Flow ID*) and priority tag (*PT*) where † denotes a non-exclusive filter; wildcard fields are denoted with ∗.

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

host or subnetwork is typically referred to as the packet classification problem. Note that the route lookup problem may be viewed as a sub-problem of the more general packet classification problem. Applications for Quality of Service, security, monitoring, and multimedia communications typically operate on flows, thus each packet traversing a router must be classified in order to assign a flow identifier, *FlowID*. Packet classification entails searching a table of filters for the highest priority filter or set of filters which match the packet. Filters bind a flow or set of flows to a *FlowID*. Note that filters are also referred to as rules in some of the packet classification literature. At minimum, filters contain multiple field values that specify an exact packet header or set of headers and the associated *FlowID* for packets matching all the field values. The type of field values are typically prefixes for IP address fields, an exact value or wildcard for the transport protocol number and flags, 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 headers fields: 8-bit source and destination addresses, transport protocol, and a 4-bit destination port number. The packet fields most commonly used for packet classification are referred to as the IP 5-tuple and include the 8-bit protocol, 32-bit source address, and 32-bit destination address in the IPv4 header as well as the 16-bit source port and 16-bit destination port in the TCP and UDP transport protocol headers.

Note that the filters in Table 1.1 also contain an explicit priority tag *PT* and a non-exclusive flag denoted by †. Priority tags allow filter priority to be independent of filter ordering, providing for simple and efficient dynamic updates. Non-exclusive flags allow filters to be designated as either exclusive or non-exclusive. A search returns the single highest-priority exclusive filter, allowing Quality of Service and security applications to specify a single action for the packet. Packets may also match several non-exclusive filters, providing support for transparent monitoring and usage-based accounting applications. Note that a parameter may control the number of non-exclusive filters, *r*, returned by the packet classifier. Like exclusive filters, the priority tag is used to select the *r* highest priority non-exclusive filters. We argue that packet classifiers should support these additional filter values and point out that many existing algorithms preclude their use. The packet classification problem may be stated formally as follows:

> Given a packet P containing fields P^j and a collection of filters F with each filter F_i containing fields F_i^j, select the highest priority exclusive filter and r highest priority non-exclusive filters where for each filter $\forall j : F_i^j$ matches P^j.

Consider the example of searching Table 1.1 for the highest-priority exclusive filter and single highest-priority non-exclusive filter, $(r = 1)$, for a packet with the following header field values:

- *SA*: 1001 1100
- *DA*: 0110 1010
- *Prot*: UDP
- *DP*: 5

The exclusive filters with *FlowIDs* 3 and 15 match the packet, but *FlowID* 3 is the highest priority filter (minimum PT value). The non-exclusive filters with *FlowIDs* 5 and 7 match the packet, but *FlowID* 5 is the highest priority filter. The search would return *FlowIDs* 3 and 5.

1.3.1 Constraints

Computational complexity is not the only challenging aspect of the packet classification problem. Increasingly, traffic in large ISP networks and 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 dictate the throughput, the number of packets per second, routers must support. A majority of Internet traffic utilizes the Transmission Control Protocol which transmits 40 byte acknowledgment packets. In the worst case, a router could receive a long stream 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.

Many algorithmic solutions to the route lookup and packet classification problems provide sufficient performance on average. Most techniques suffer from poor performance for a pathological search. For example, a technique might employ a decision tree where most paths through the tree are short, however one path is significantly long. If a sufficiently long sequence of packets that follows the longest path through the tree arrives at the input port of the router, then the throughput is determined by the worst-case search performance. It is this set of worst-case assumptions that imposes the so-called “wire speed requirement” for route lookup and packet classification solutions. In essence, solutions to these search problems are almost always evaluated based on the time it takes to perform a pathological search. In the context of networks that provide performance guarantees, engineering for the worst case logically follows. In the context of the Internet, the protocols make no performance guarantees and provide “best-effort” service to all traffic. Furthermore, the switching technology at the core of routers cannot handle pathological traffic. Imagine a sufficiently long sequence of packets in which all the packets arriving at the input ports are destined for the same output port. When the buffers in the router ports fill up, it will begin dropping packets. Thus, the “wire speed requirement” for Internet routers does not logically follow from the high-level protocols or the underlying switching technology; it is largely driven by network management and marketing concerns. Quite simply, it is easier to manage a network with one less source of packet losses and it is easier to sell an expensive piece of network equipment when you don’t have to explain the conditions under which the search engines in the router ports will begin backlogging. It is for these reasons that solutions to the route lookup and packet classification problems are typically evaluated by their worst-case performance.

Achieving tens of millions of lookups per second is not the only challenge for route lookup and packet classification search engines. Due to the explosive growth of the Internet, backbone route tables have swelled to over 100k entries. Likewise, the constant increase in the number of security filters and network service applications causes packet classification filter sets to increase in size. Currently, the largest filter sets contain a few thousand filters, however dynamic resource reservation protocols could cause filter sets to swell into the tens of thousands. Scalability to larger table sizes is a crucial property of route lookup and packet classification solutions; it is also a critical concern for search techniques whose performance depends upon the number of entries in the tables.

As routers achieve aggregate throughputs of trillions of bits per second, power consumption becomes an increasingly critical concern. Both the power consumed by the router itself and the infrastructure to dissipate the tremendous heat generated by the router components significantly contribute to the operating costs. Given that each port of high-performance routers must contain route lookup and packet classification devices, the power consumed by search engines is becoming an increasingly important evaluation parameter. While we do not provide an explicit evaluation of power consumption in this dissertation, we present solutions to the route lookup and packet classification techniques that employ low-power memory technologies.

1.4 Organization of the Dissertation

The remainder of the dissertation is organized as follows. The next chapter provides an overview of single field search techniques, including Longest Prefix Matching (LPM) techniques specifically developed in response to the route lookup problem. The other types of searches covered in Chapter 2 have relevance for the types of searches dictated by the packet classification problem. In order to demonstrate the level of performance and efficiency achievable via high-performance implementations of algorithms, Chapter 3 provides a description of the Fast Internet Protocol Lookup (FIPL) search engine. Targeted to open-platform research systems designed and developed at Washington University, FIPL is a high-performance hardware implementation of the Tree Bitmap algorithm developed by Eatherton and Dittia [11].

Chapter 4 presents a survey of solutions to the packet classification problem using a taxonomy that frames each solution according to its high-level approach to the problem. Motivated by recent packet classification algorithms that leverage properties of real filter sets in order to achieve better performance, Chapter 5 contains a detailed analysis of 12 real filter sets collected from fellow researchers, Internet Service Providers (ISPs), and a network equipment vendor. Unlike the field of computer architecture, there are no standard filter sets or performance evaluation tools that provide a uniform scale for comparing competing packet classification solutions. In response, we developed a suite of benchmarking tools that includes a *Synthetic Filter Set Generator*. A description and analysis of the *ClassBench* tools is contained in Chapter 6. Based on the results of the analysis presented in Chapter 5, we developed a new packet classification algorithm that leverages the structure of real filter sets and the capabilities of modern hardware technology. Chapter 7 presents a description and performance analysis of the new technique, *Distributed Crossproducting of Field Labels*, which provides favorable scaling properties for larger filter sets and more complex filters. We provide a summary of the contributions in this dissertation and discussion of future work in Chapter 8.

Chapter 2

Single-Field Search Techniques

Computers are useless. They can only give you answers.
Pablo Picasso

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 three different match conditions on the corresponding packet header fields: (1) a fully specified value, or exact matching, (2) partially specified value, or prefix matching, (3) a range of values, or range matching. In this chapter, we provide a survey of the prominent solutions to these three types of search problems, focusing on the most frequently used solutions and those solutions specifically tailored to networking applications. We begin with a survey of solutions for exact matching in Section 2.1, followed by a discussion of Longest Prefix Matching (LPM) techniques in Section 2.2. LPM has been the focus of intensive study in recent years due to the fundamental role it plays in IP address lookups for packet forwarding. Note that LPM is a special case of the more general All Prefix Matching (APM) problem discussed in Section 2.3. Various packet classification techniques require an efficient solution to the APM problem. Finally, we address the more challenging problem of range matching. Fortunately, range matching is a problem that arises in many contexts. We provide a survey of range matching solutions drawn from the fields of computational geometry, database design, and networking in Section 2.4.

2.1 Exact Matching

The simplest form of exact matching is the set membership query: determine if 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 unique identifiers or processing directives. 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 transport protocol identifier. 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 this task. We provide a brief introduction to two classical data structures that seek 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. The space efficiency comes at the price of allowing false positive matches, as well as not storing any additional information with the keys in the set.

2.1.1 B-Trees

B-Trees were originally designed to limit the number of accesses to direct access storage units such as disks and drums [12, 13]. 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 a node may have is typically referred to as the *degree* of the tree. The number 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. Note that the keys stored in a node are arranged in non-decreasing order. Each internal node also stores a set of pointers interspersed with the keys. Each pointer points to a child node storing keys greater than the key to the “left” of the pointer and less than or equal to the key to the “right” of the pointer. Note that each node may also store additional information for each key[1] Finally, the height h of a B-Tree containing n keys is bounded by:

$$h \leq \log_B \frac{n+1}{2} \tag{2.1}$$

Thus, given a maximum table size the value of B can be selected to meet a given access budget. Note that we assume a pointer to additional data may be stored along with each key. Another common B-Tree organization stores all pointers to additional data in the leaves and only stores keys and child pointers in the internal nodes in order to maximize the branching factor of the internal nodes.

2.1.2 Hashing

Hashing is a technique that can provide excellent average performance when the number of keys, n, in the set X is much less than the number of keys, $|U|$, in the universe of possible key values, U. For example, assume that X contains 100 keys where the keys may take on 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

[1] Each B-Tree node could also store a pointer to a table of information that could be indexed by the matching key’s position in the node.

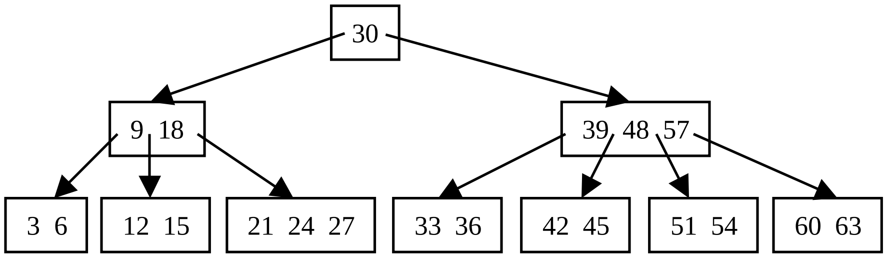


Figure 2.1: Example of a B-Tree storing multiples of three, where $t = 3$.

technique, *direct addressing*, is only viable when the number of keys n in the set X approaches the number of possible key values $|U|$.

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 $m \ll |U|$. 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 [12, 13].

There is a variety of methods for constructing hash functions. Often, the low-order bits of key values are sufficiently uniform in distribution such that the *hash index* may be constructed by selecting low order bits of the key. Such hash functions are trivial to construct in hardware. Figure 2.2 shows 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 and/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*, $\alpha = \frac{n}{m}$, which specifies the average number of keys in a chain. Thus, the average search time for a hash table where chaining is used for collision resolution is $\Theta(1 + \alpha)$. There is a variety of much more sophisticated hash functions and collision resolution techniques. We refer the reader to the previously mentioned textbooks for a more complete discussion [12, 13].

2.1.3 Bloom Filters

A Bloom filter is a data structure used for efficiently representing a set of keys. Via implicit representations of the keys in the set, the data structure supports membership queries but is not capable of storing additional information for each stored key. This technique was formulated by Burton H. Bloom in 1970 [14], and has received renewed attention in the research community for various applications such as web caching, intrusion detection, and content based routing [15]. A Bloom filter is essentially a bit-vector of length m where a key x is represented by a subset of the m bits.

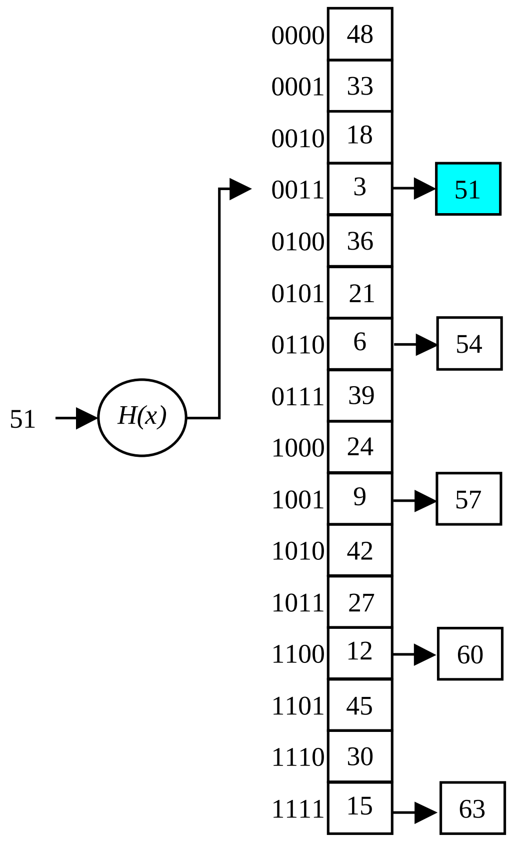


Figure 2.2: Example of hashing with chaining using the four low-order bits as a hash index.

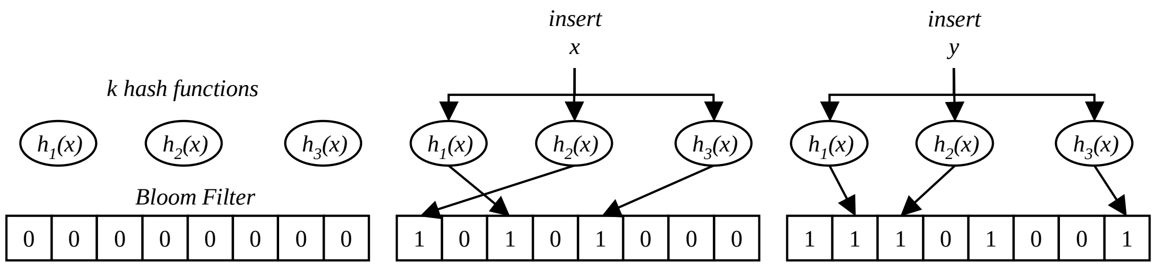


Figure 2.3: Example of inserting two keys, x and y, into a Bloom filter.

Given a set of keys X with n members, we insert a key $x_i \in X$ into the Bloom filter as follows[2]. 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 addresses a bit that is already set to 1, that bit is not changed.

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

[2]Inserting a key into a Bloom filter is also referred to as "programming" the filter in the literature.

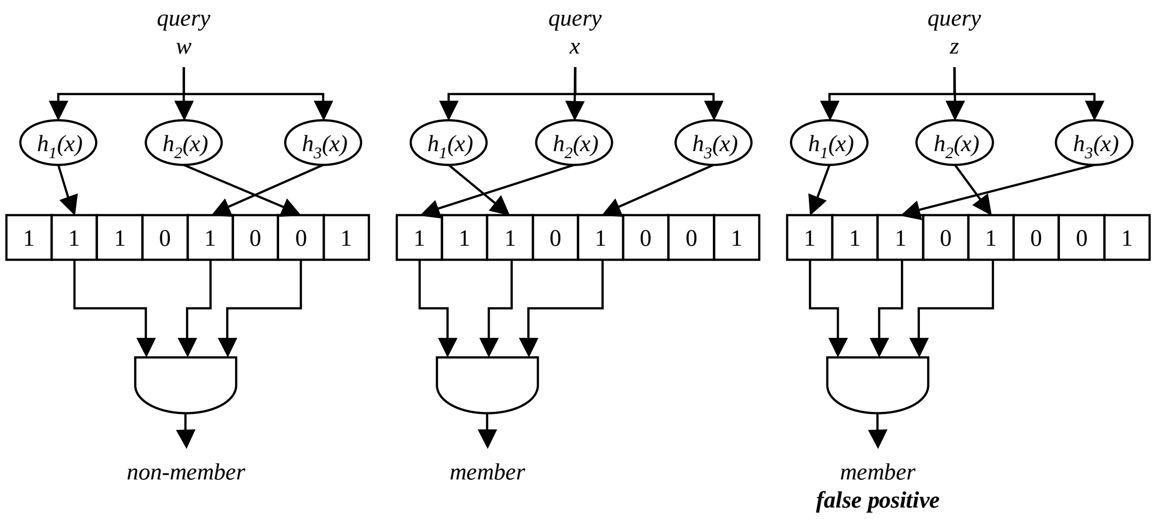


Figure 2.4: Example of querying a Bloom filter; w is a non-member, x is a correct match; z is a false positive match.

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 we declare the key to be a non-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 were a member then all k-bits would have been set to 1 when the key was inserted into the Bloom filter. Examples of a non-match, correct match, and false positive match are shown in Figure 2.4.

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 $(1 - \frac{1}{m})^{nk}$. Hence, the probability that this bit is found to be 1 is $1 - (1 - \frac{1}{m})^{nk}$. 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

$$f = \left(1 - \left(1 - \frac{1}{m}\right)^{nk}\right)^{k} \tag{2.2}$$

for large values of m the above equation reduces to

$$f \approx \left(1 - e^{\frac{-nk}{m}}\right)^{k} \tag{2.3}$$

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

$$k = \left\{ \left\lfloor \frac{m}{n} \ln 2 \right\rfloor , \left\lceil \frac{m}{n} \ln 2 \right\rceil \right\} \tag{2.4}$$

The false positive probability at this optimal point is given by

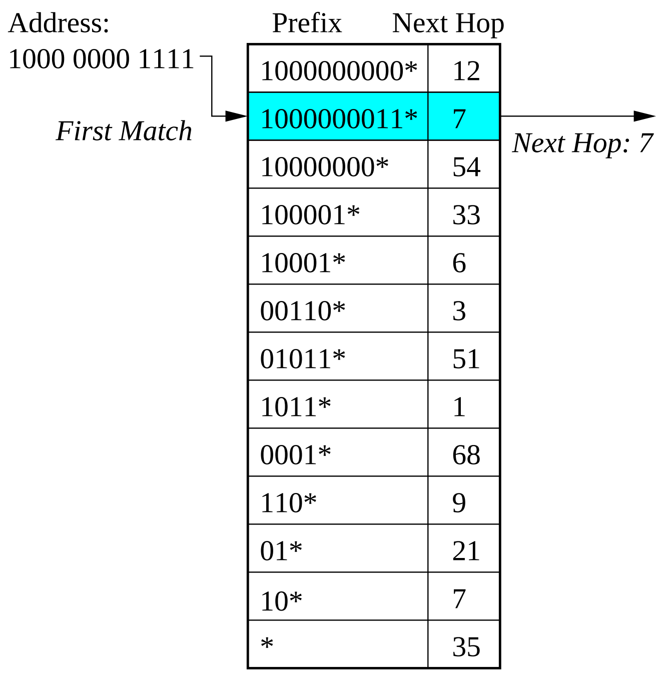
$$f = \left(\frac{1}{2} \right)^k \tag{2.5}$$

It should be noted that if the false positive probability is to be fixed, 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. [16]. 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 zero to one, the corresponding bit in the bit-vector is set. 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. If updates to the set of stored keys arrive at a reasonable rate, then the counters may be stored in slower, cheaper memory technology such as DRAM.

2.2 Longest Prefix Matching (LPM)

Longest Prefix Matching (LPM) has received significant attention in the literature over the past ten years. This is due to the fundamental role it plays in the performance of Internet routers. Due to the explosive growth of the Internet, Classless Inter-Domain Routing (CIDR) was widely adopted to prolong the life of Internet Protocol Version 4 (IPv4) [7]. 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. We will use IP lookup as the example application for Longest Prefix Matching for the remainder of the section. The



Prefix	Next Hop
1000000000*	12
1000000011*	7
10000000*	54
100001*	33
10001*	6
00110*	3
01011*	51
1011*	1
0001*	68
110*	9
01*	21
10*	7
*	35

Figure 2.5: Example of Longest Prefix Matching for a 12-bit address using linear search; prefixes are sorted in decreasing order of prefix length; the first matching prefix is the longest.

following sections discuss the major developments in LPM techniques for IP lookup, categorized by their general approach to the problem.

2.2.1 Linear Search

If the set of prefixes is small, a linear search through a list of the prefixes sorted in order of decreasing length is sufficient. The sorting step guarantees that the first matching prefix in the list is the longest matching prefix for the given search key. An example of Longest Prefix Matching (LPM) using linear search is shown in Figure 2.5. Linear search is commonly touted as the most memory efficient of all LPM techniques in that 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 a viable approach for IP lookup when the set of prefixes grows beyond a few dozen prefixes.

2.2.2 Content Addressable Memory (CAM)

Many commercial router designers have chosen to use Content Addressable Memory (CAM) for IP address lookups in order to keep pace with 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 [17], 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 thereby enabling them to retain single clock cycle lookups for arbitrary prefix lengths. We believe that this "brute-force" approach is no longer necessary for IP lookup due to the significant advances that have been made in algorithmic LPM techniques. TCAMs remain competitive choices for packet classification on multiple fields; therefore, we provide a more detailed analysis of these devices in Section 4.2.2.

2.2.3 Trie Based Schemes

Search techniques which build decision trees using the bits of prefixes to make branching decisions allow the worst-case search time to be independent of the number of prefixes in the set. An example of a binary trie[3] constructed from the set of prefixes in Figure 1.4 is shown in Figure 2.6. Shaded nodes denote a stored prefix; the corresponding next hop is shown adjacent to the node. A search is conducted by traversing the trie using the bits of the address, starting with the most significant bit. As in the previous examples, the best matching prefix is $1000000011*$ and the corresponding next hop is seven. 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.

One of the first IP lookup techniques to employ *tries* is Sklower's implementation of a Patricia trie in the BSD kernel [18]. The Patricia trie is a binary radix tree that compresses paths with one-way branching into a single node. The BSD kernel implementation was designed to be general enough to support any hierarchical routing scheme or link layer address translation such as the Address Resolution Protocol (ARP). 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* [19]. *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 need only store a pointer or next hop information. While this technique reduces memory usage, it also increases incremental update overhead. The authors also

[3]A trie is an ordered tree in which the key stored at each node is specifi ed by its position in the tree.

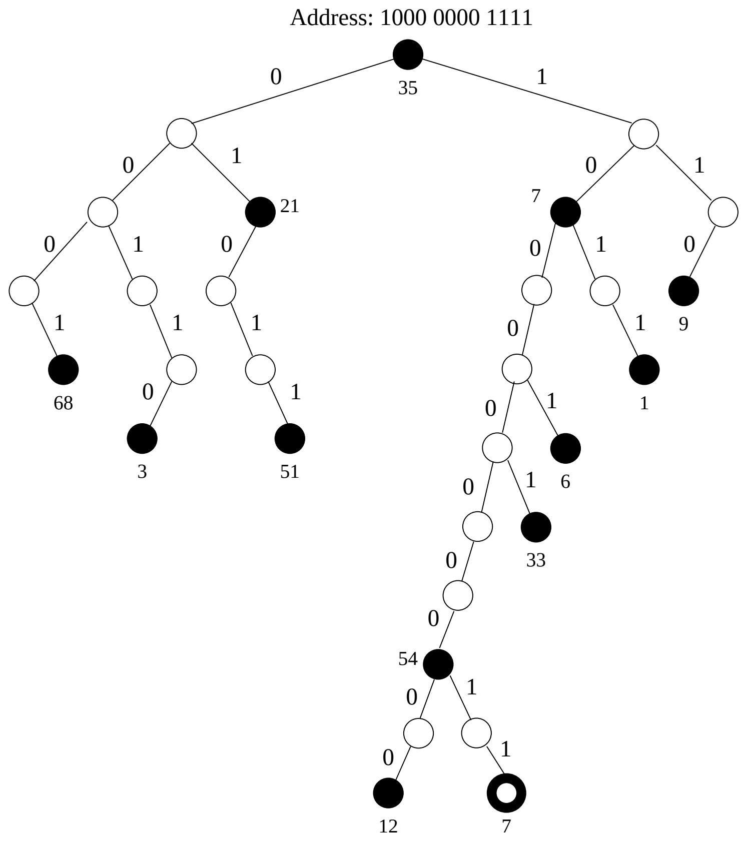


Figure 2.6: Example of Longest Prefix Matching using a binary trie.

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 [20]. Arguing that DRAM is such a plentiful and inexpensive resource, their technique sacrifices large amounts of memory in order to bound 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 fewer than 10 memory accesses.

Other techniques such as *Lulea* [21] and Eatherton and Dittia's *Tree Bitmap* [11] employ multi-bit tries with compressed nodes. In Chapter 3 we provide a detailed description and analysis of a scalable hardware implementation of *Tree Bitmap*. We also provide an introduction to multi-bit tries, a complete description of the *Tree Bitmap* algorithm, and a comparison between *Tree Bitmap* and other approaches such as *Lulea*. 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. Storage requirements for *Tree Bitmap* are on the order of six to eight bytes per address prefix, worst-case memory accesses can be held to less than eight with optimizations, and updates require modifications to a few memory words resulting in excellent incremental update performance [22].

The fundamental issue with trie-based techniques is that performance and scalability are fundamentally tied to address length. As many in the Internet community are pushing to widely adopt IPv6, it is not clear that trie-based solutions will be capable of meeting performance demands. In the following sections, we discuss LPM algorithms that avoid this linear relationship with address length.

2.2.4 Multiway and Multicolumn Search

Several other algorithms exist with attractive properties that are not based on tries. The *Multiway and Multicolumn Search* techniques presented by Lampson, Srinivasan, and Varghese are designed to optimize performance for software implementations on general purpose processors [23]. 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.

The authors also used a popular optimization, a precomputed index array. An example of a precomputed index array[4] for the first three bits of our example prefix set is shown in Figure 2.7. We begin by storing the prefixes in a binary trie, then perform Controlled Prefix Expansion (CPE) for a stride length equal to three [19]. The next hop associated with each node at level three is written to the array slot addressed by the bits labeling the path from the root to the node. If the node has children, then a pointer to a binary trie containing the children is stored. The structure is searched by using the first three bits of the address to index into the array. If no pointer is stored, then the next hop at the array index is returned as the next hop. If a pointer is stored, then the next hop at the array index is remembered as the best match thus far and the search continues using the binary trie identified by the pointer. Note that this data structure requires $2^a \times q$ bits of memory where a is the number of bits used to index the array and q is the number of bits required for next hop and pointer storage.

[4]Precomputed index arrays are also called "initial arrays" and "direct lookup arrays" in the literature

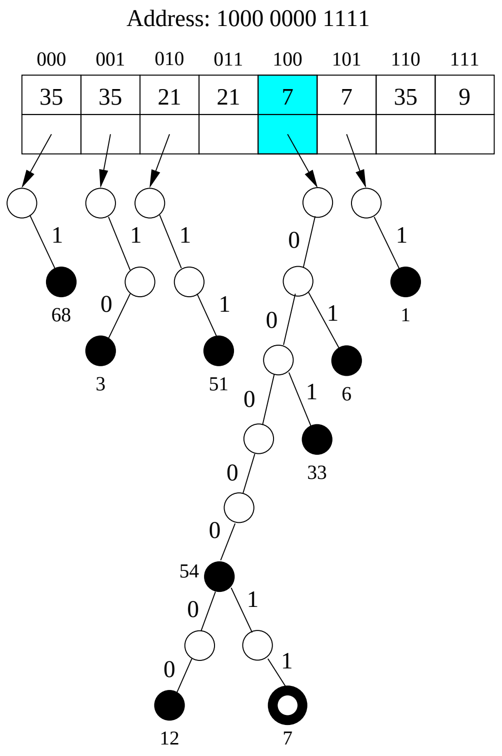


Figure 2.7: Example of a direct lookup array for the first three bits.

Finally, the authors optimize their algorithm based on the memory hierarchy of modern general purpose processors. The data structures are dimensioned to take advantage of the cache line size of the target processor. Even though it is geared to software implementation, it may not be viable for current generation network processors that do not contain full memory hierarchies. In general, the approach requires $O(W + \log N)$ time and $O(2N)$ memory, where N is the number of prefixes and W is the length of the address. Again, the primary issue with this algorithm is its linearly scaling relative to address length.

2.2.5 Binary Search on Prefi x Lengths

The most efficient lookup algorithm known, from a theoretical perspective, is *Binary Search on Prefix Lengths* which was introduced by Waldvogel, et. al. [24]. The number of steps required by this algorithm grows logarithmically in 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 implementation than hardware implementation. It also does not readily support incremental updates.

This technique bounds the number of memory accesses via significant precomputation 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. Going back to our example, a length 24 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.

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 $O(\log W_{dist})$ time and $O(N \times \log W_{dist})$ space where W_{dist} is the number of unique prefix lengths. Empirical measurements using an IPv4 route table resulted in memory requirement of about 42 bytes per entry.

2.2.6 Longest Prefi x Matching using Bloom Filters

Dharmapurikar, Krishnamurthy, and Taylor introduced the first algorithmic Longest Prefix Matching (LPM) technique to employ Bloom filters [25]. This approach, which we will refer to as *Bloom filter-based IP Lookup* (*BIPL*), is a hardware-based IP lookup solution with *average* performance superior to TCAMs. Mitigating worst-case performance requires an initial index array and *Controlled Prefix Expansion* (CPE) which causes *BIPL* to become less memory and update efficient. The performance bottleneck in any longest prefix matching technique is the number of sequential memory accesses required per lookup. The key feature of *BIPL* is that the performance, as determined by the expected number of sequential memory accesses required per lookup, can be held to a constant regardless of address length and number of unique prefix lengths. The approach is equally attractive for Internet Protocol Version 6 (IPv6) which uses 128-bit destination addresses, four times longer than IPv4.

A basic configuration of *BIPL* is shown in Figure 2.8. It begins by sorting the set of prefixes into sets according to prefix length. The system employs a set of W counting Bloom filters, where W is the maximum number of unique prefix lengths in the prefix set, and associates one Bloom filter with each unique prefix length. Each filter is "programmed" with the associated set of prefixes according to the previously procedure described in Section 2.1.3. It is important to note that while the bit-vectors associated with each Bloom filter must be stored on-chip, the counters associated with each filter can be maintained by a separate control processor responsible for route updates. Separate control processors with ample memory are typical features of high-performance routers.

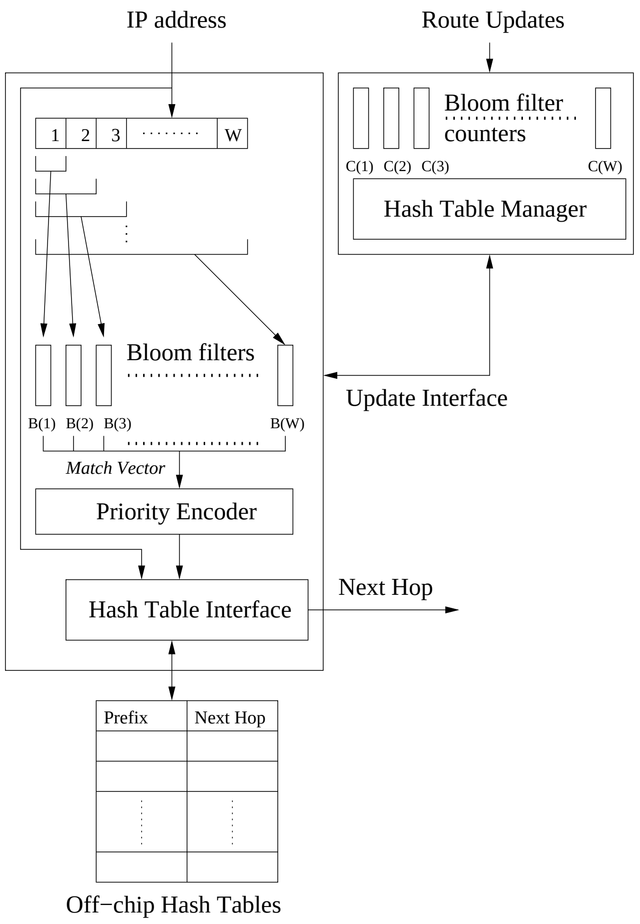


Figure 2.8: Basic configuration of Longest Prefix Matching using Bloom filters, (*BIPL*).

A hash table is also constructed for each distinct prefix length. Each hash table is initialized with the set of corresponding prefixes, where each hash entry is a (*prefix*, *next hop*) pair. The set of hash tables is stored in off-chip memory. Given that the problem of constructing hash tables to minimize collisions with reasonable amounts of memory is well-studied, the authors assume that probing a hash table stored in off-chip memory requires one memory access [24].

A search proceeds as follows. The input IP address is used to probe the set of W on-chip Bloom filters in parallel. The first bit of the address is used to probe the filter associated with length one prefixes, the first and second bits of the address are used to probe the filter associated with length two prefixes, etc. Each filter simply indicates match or no match. By examining the outputs of all filters, we compose a vector of potentially matching prefix lengths for the given address, the *match vector*. Consider an IPv4 example where the input address produces matches in the Bloom filters associated with prefix lengths 8, 17, 23, and 30; the resulting *match vector* would be [8,17,23,30]. Remember that Bloom filters may produce false positives, but never produce false negatives; therefore, if a matching prefix exists in the route table, it will be represented in the match vector. Note that the number of unique prefix lengths represented in the route table, W_{dist}, may be less than W. In this case, the Bloom filters representing empty sets will never contribute a match to the *match vector*, valid or false positive. The search proceeds until a match is found or the vector is exhausted.

The probability of a false positive is dependent upon the number of prefixes stored in a filter, the size of the filter, and the number of hash functions used to probe the filter. The authors show that with a modest amount of on-chip resources for Bloom filters, the average number of hash probes per lookup approaches one; therefore, this approach can achieve lookup rates equivalent to those offered by TCAMs. Given that commodity SRAM devices are denser and cheaper than TCAMs, this approach potentially offers lower cost and more power efficient solution. The authors also introduce asymmetric Bloom filters which dimension filters according to prefix length distribution. A system configured to support 200,000 IPv4 prefixes with an average number of 1.003 off-chip memory accesses per lookup, requires 4Mb of on-chip memory and is capable of 332 million lookups per second using a commodity SRAM device operating at 333 MHz.

2.3 All Prefi x 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 be returned. This problem arises when multi-field search techniques are decomposed into several instances of single-field search techniques. We provide a survey of multi-field search techniques in Chapter 4.

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. Similarly, the Bloom filter technique can also be easily adapted to perform APM. Referring back to Figure 2.8, the Priority Encoder can be removed and the Hash Interface simply queries every hash table associated with matching prefix lengths in the *match vector*. This does increase the number of hash probes per lookup; however, as discussed in Chapter 5 the number of prefixes in multi-field search tables which match an address is typically less than six.

While the trie-based and Bloom filter-based LPM 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. The idea is to perform an LPM search where stored prefixes contain a pointer to a node in a *nesting tree*, a separate tree of prefixes defined by parent pointers. Figure 2.9 shows an example of a *nesting tree* for the prefixes used in the LPM example of Figure 1.4. All matching prefixes for a given longest matching prefix are found

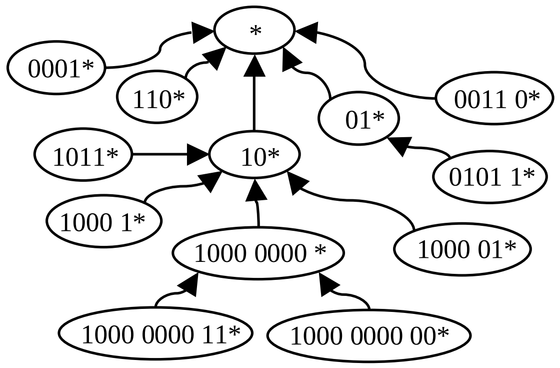


Figure 2.9: *Nesting tree* technique for finding all matching prefixes for a given longest matching prefix.

by simply following parent pointers until the root node is reached. This general technique can be made memory and update efficient, but does require additional memory accesses to find all matching prefixes. A second technique may be used that does not require additional memory accesses but sacrifices memory and update efficiency. The idea is to precompute all matching prefixes associated with each prefix in the set. The list of all matching prefixes is stored with each prefix in the LPM data structure, thus locating the longest matching prefix returns the list of all matching prefixes. Note that this suffers from memory and update inefficiency as many prefixes are stored redundantly in lists and updating an entry in the prefix set may require many updates to lists of all matching prefixes.

2.4 Range Matching

Range matching problems naturally arise in many searching problems in the areas of networking, computational geometry, and database design, and there are several forms of range matching problems. In this section we provide a brief survey of 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 one of two classical data structures, the Segment Tree and the Interval Tree [26, 27]. Another option is to convert each closed interval $[i, j]$ into a set of prefixes, then employ one of the fast Longest Prefix Matching (LPM) algorithms discussed in the previous section [28, 29]. Finally, we describe a recently proposed hardware solution for range matching.

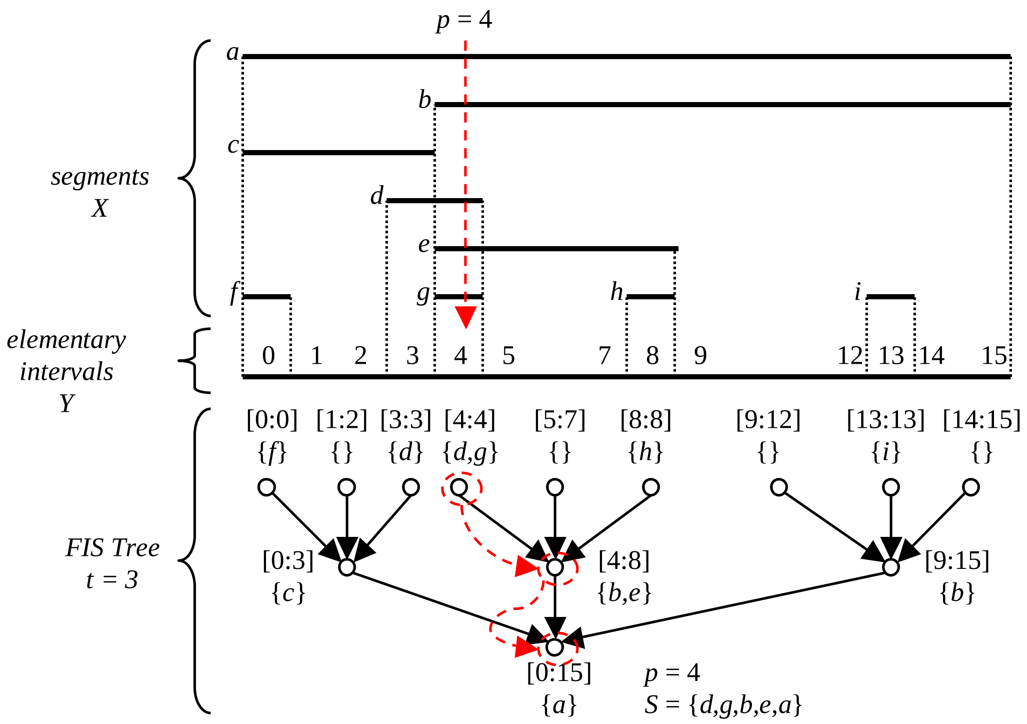


Figure 2.10: Example of projecting endpoints of intervals to form non-overlapping segments on the real line, and using the *Fat Inverted Segment* (*FIS*) *Tree* to search the set of segments.

2.4.1 Segment Tree

Extensively used in computational geometry, a Segment Tree is a data structure that stores a set of segments on the real line [30]. For the purpose of our discussion, a set of segments is a set of closed intervals X. Segment Trees typically utilize some form of a binary search tree as an underlying data structure. In order to use such data structures, the endpoints of the segments must be projected onto the real line in order to form non-overlapping *elementary intervals*. Given a set of segments X containing $|X|$ segments, the set Y of *elementary intervals* contains at most $(2|X| - 1)$ segments. An example is shown in Figure 2.10.

Balanced binary search trees or splay trees can be used in order to limit the height of a binary search tree [31]. When used to store elementary intervals, a Segment Tree can return a set of matching segments S for a given point p in $O(\log |Y|)$ time, where $\forall [i, j] \in S, i \leq p \leq j$. Balanced binary search trees enforce a balance condition, such that updates to the data structure do not cause the balance condition to be violated. Red-black trees are one example of a balanced binary tree that ensures that every path from the root node to a leaf node is no longer than twice the shortest path from the root node to a leaf node [31, 13]. Splay trees do not explicitly enforce a balance or height condition; rather, they employ a set of heuristics that prescribe a series of recursive restructuring operations each time the splay tree is accessed or updated. These heuristics have been shown to maintain data structure balance and provide logarithmic amortized search time [31]. While fascinating from a theoretical perspective and useful in other problem domains, we believe that the real time constraints for packet classification searches preclude the use of splay trees due to the restructuring operations performed during accesses.

Note that we could precompute the intervals that overlap each segment and store this information in the segment tree. While efficient for searching, the update time is $O(|Y|)$ in the worst case; consider adding or removing interval a. In order to improve the update and search performance, Feldman and Muthukrishnan proposed the *Fat Inverted Segment* (*FIS*) *Tree* [27]. The *FIS Tree* is a balanced t-ary tree with l levels, where $t = (|Y|)^{1/l}$. An example of an *FIS Tree* is shown in Figure 2.10. Each node v represents an interval $I(v)$ which is the union of the intervals represented by its children. Leaf nodes represent the *elementary intervals*. For the purpose of our discussion, the salient features of the *FIS Tree* are: (1) the height of the tree can be limited by choosing a sufficient branching parameter t, (2) each node v only stores an interval x if $I(v) \subseteq x$ and $x \subset I(parent(v))$. Note that the choice of t affects the complexity of the branching decision at each internal node[5]. The set of segments S overlapping a given elementary interval y can by found by traversing the path from the leaf representing y to the root of the tree, i.e. the "inverse" path, and appending the set of segments stored at each node v to S. An example is shown in Figure 2.10 for $p = 4$. Letting $M = (2|X| + 1)$, the *FIS Tree* requires $O(\log_t M)$ search time, $O(M \log_t M)$ update time, and $O(M \log_t M)$ space.

2.4.2 Interval Tree

An Interval Tree stores a set of closed intervals X using a balanced binary tree as the underlying data structure [13]. Its primary distinction from the Segment Tree is that the Interval Tree does not use *elementary intervals*; each node in the 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 subtree. An example of an Interval Tree is shown in Figure 2.11.

Searching for one matching interval for a given point p is straight-forward. 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[6]. 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 2.11. Letting $|S|$ be the number of matching intervals, the search requires $O(\lg |X| + |S|)$ time. The Interval Tree requires $O(\lg |X|)$ amortized update time and $O(|X|)$ space.

[5]Feldman and Muthukrishnan propose using *FIS Trees* for a multi-fi eld search; thus the search begins from the leaves and involves more intermediate steps to support multiple fi elds.

[6]This can be facilitated by storing the minimum endpoint value in the subtree rooted at each node.

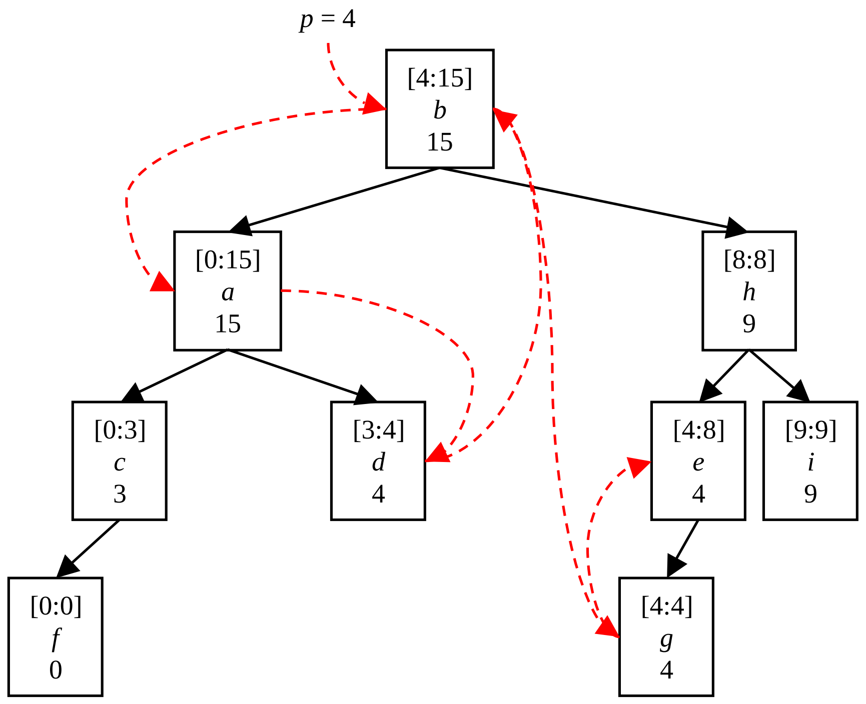


Figure 2.11: Example of an *Interval Tree* where each node stores the maximum endpoint value for all intervals in its subtree.

2.4.3 Range to Prefi x Conversion

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

For a single-field search on a reasonable number of ranges, this expansion factor is not prohibitive. As a result, several packet classification techniques use the range to prefix conversion technique to solve the range matching subproblem of the general packet classification problem [28, 29]. As discussed in Chapter 4, this conversion can become problematic for multiple-field searches due to the compounding effect on the expansion factor. Specifically, for a multiple-field filter with a fields specifying ranges on the set of b-bit numbers, converting the range fields into prefix fields results in up to $[2\times(b-1)]^a$ filters. Finally, we note that Feldman and Muthukrishnan 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.

2.4.4 Range Matching Circuits

In order to eliminate the aforementioned expansion factor when using Ternary Content Addressable Memory (TCAM) devices, range matching can be performed directly in hardware [32]. When implemented in standard CMOS technology, a range matching circuit requires $44b$ transistors where b is the number of bits required to specify a point in the range. This is considerably more than the 16 transistors per bit required for prefix matching; however, the total hardware resources saved by eliminating the expansion factor for typical packet filter sets far outweighs the additional cost per bit for hardware range matching.

Chapter 3

Fast Internet Protocol Lookup (FIPL)

Where a calculator on the ENIAC is equipped with 18,000 vacuum tubes and weighs 30 tons, computers in the future may have only 1,000 vacuum tubes and perhaps weigh 1.5 tons.
Popular Mechanics, March 1949

In this chapter we provide a detailed description of the design, implementation, and analysis of a Longest Prefix Matching (LPM) search engine using a compressed multi-bit trie algorithm. This work provides concrete evidence that high-performance implementations of clever algorithms can achieve the required search and update rates with efficient use of hardware, memory, and power. It is important to note that the value of this work reaches beyond the problem of Internet Protocol (IP) address lookups. As demonstrated in Chapter 7, packet classification techniques can make use of optimized single-field search engines.

3.1 Introduction

Forwarding of Internet Protocol (IP) packets is the primary purpose of Internet routers. The speed at which forwarding decisions are made at each router or “hop” places a fundamental limit on the performance of the network. For Internet Protocol Version 4 (IPv4), the forwarding decision is based on a 32-bit destination address carried in each packet’s header. The use of Classless Inter-Domain Routing (CIDR) complicates the lookup process, requiring a lookup engine to search a route table containing variable-length address prefixes in order to find the longest matching prefix for the destination address in each packet header and retrieve the corresponding forwarding information [7]. In high-performance routers, each port employs a separate LPM search engine. We provide a more complete introduction to the IP lookup problem in Chapter 1.

As physical link speeds grow and the number of ports in high-performance routers continues to increase, there is a growing need for efficient lookup algorithms and effective implementations of those algorithms. Next generation routers must be able to support thousands of optical links each operating at 10 Gb/s (OC-192) or more. Lookup techniques that can scale efficiently to high speeds and large lookup table sizes are essential for meeting the growing performance demands, while maintaining acceptable per-port costs.

Many techniques are available to perform IP address lookups. Perhaps the most common approach in high-performance systems is to use Ternary Content Addressable Memory (TCAM) devices. While this approach can provide excellent performance, the performance comes at a fairly high price due to the exorbitant power consumption and high cost per bit of TCAM relative to commodity memory devices. We provide an overview of LPM algorithms and devices in Section 2.2.

The Fast Internet Protocol Lookup (FIPL) search engine [22], developed at Washington University in Saint Louis, is an experimental implementation of Eatherton and Dittia's *Tree Bitmap* algorithm [11] using reconfigurable hardware and Random Access Memory (RAM). Targeted to an open-platform research router, FIPL is designed to strike a favorable balance among lookup and update performance, memory efficiency, and hardware usage. Employing a Xilinx Virtex 1000E-7 Field Programmable Gate Array (FPGA) operating at 100MHz and a single Micron 1MB Zero-Bus Turnaround (ZBT) Synchronous Random-Access Memory (SRAM)[1], a single FIPL lookup engine has a guaranteed worst case performance of 1,136,363 lookups per second. Interleaving memory accesses of eight FIPL engines over a single 36 bit wide SRAM interface exhausts the available memory bandwidth and yields a guaranteed worst case performance of 9,090,909 lookups per second.

Performance evaluations using a snapshot of the Mae-West routing table resulted in over 11 million lookups per second for an optimized eight FIPL engine configuration. Average memory usage per entry was 6.3 bytes, which is comparable to the amount of memory required to explicitly represent an individual prefix. In addition to space efficiency, the data structure used by FIPL is straightforward to update, and can support up to 100,000 updates per second with only a 7.2% degradation in lookup throughput. Each FIPL engine utilizes less than 1% of the available logic resources on the target FPGA. While this search engine currently achieves 500 Mb/s of link traffic per 1% of logic resources, still higher performance and efficiency is possible with higher memory bandwidths. Ongoing research seeks to exploit new FPGA devices and more advanced CAD tools in order to double the clock frequency and, therefore, double the lookup performance. We also are investigating optimizations to reduce the number of off-chip memory accesses. Another research effort leverages the insights and components produced by the FIPL implementation for an efficient route lookup and packet classification engine for an open-platform dynamically extensible research router [33]. Finally, we provide a brief discussion of lookup techniques closely related to Tree Bitmap in Section 3.7.

[1]Micron ZBT SRAMs allow a new read/write operation on every clock cycle. In our research system, the SRAMs are driven by the same 100MHz clock used for the FPGAs; thus, at 10ns per cycle with 36-bit memory words, the SRAMs provide a random-access throughput of 3.6 billion bits per second (Gb/s).

3.2 Tree Bitmap Algorithm

Eatherton and Dittia’s *Tree Bitmap* algorithm is a hardware-based approach that employs a compressed multibit trie data structure to perform Longest Prefix Matching (LPM) at high rates with efficient use of memory [11]. Due to the use of CIDR, IP route lookups consist of finding the longest matching prefix stored in the forwarding table for a given 32-bit IPv4 destination address and retrieving the associated forwarding information. As shown in Figure 3.1, the destination IP address is compared to the stored prefixes starting with the most significant bit. Note that this is the same example set of prefixes used in the survey of Longest Prefix Matching techniques in Section 2.2. In this example, a packet is bound for a workstation at Washington University in Saint Louis. A linear search through the table results in three matching prefixes: *, 10*, and 1000000011*. The third prefix is the longest match, hence its associated forwarding information, denoted by Next Hop 7 in the example, is retrieved. Using this forwarding information, the packet is forwarded to the specified next hop by modifying the packet header.

Prefix	Next Hop
*	35
10*	7
01*	21
110*	9
1011*	1
0001*	68
01011*	51
00110*	3
10001*	6
100001*	33
1000000000*	12
1000000011*	7

32−bit IP Address
128.252.153.160
1000 0000 1111 1100 ... 1010 0000
Next Hop
7

Figure 3.1: IP lookup table of next hops. Next hops for IP packets are found using the longest matching prefix in the table for the IP destination address of the packet.

To efficiently perform this lookup function in hardware, the *Tree Bitmap* algorithm starts by storing prefixes in a binary trie as shown in Figure 3.2. Shaded nodes denote a stored prefix. A search is conducted by using the IP address bits to traverse the trie, starting with the most significant bit of the address. To speed up this searching process, multiple bits of the destination address are compared simultaneously. In order to do this, subtrees of the binary trie are combined into single nodes producing a multibit trie; this reduces the number of memory accesses needed to perform a lookup. The depth of the subtrees combined to form a single multibit trie node is called the stride. An example of a multibit trie using 4-bit strides is shown in Figure 3.3. In this case, 4-bit nibbles of the destination address are used to traverse the multibit trie. Address_Nibble(0) of the address, 1000_2 in the example, is used for the root node; Address_Nibble(1) of the address, 0000_2 in the example, is used for the next node; etc.

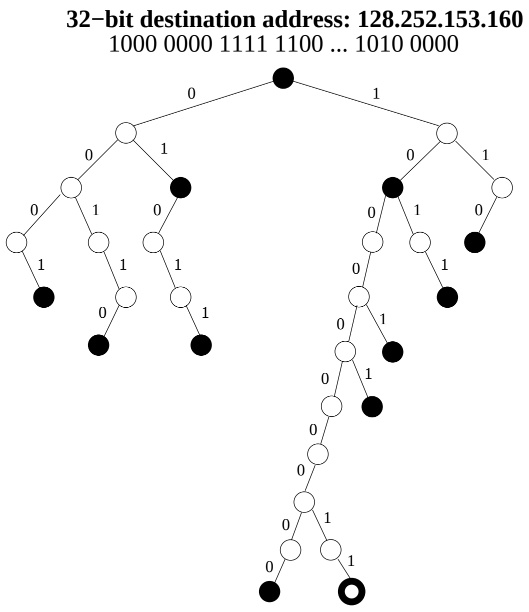


Figure 3.2: IP lookup table represented as a binary trie. Stored prefixes are denoted by shaded nodes. Next hops are found by traversing the trie.

The *Tree Bitmap* algorithm codes information associated with each node of the multibit trie using bitmaps. The *Internal Prefix Bitmap* identifies the stored prefixes in the binary sub-tree of the multi-bit node. The *Extending Paths Bitmap* identifies the "exit points" of the multibit node that correspond to child nodes. Figure 3.4 shows how the root node of the example data structure is coded into bitmaps. The 4-bit stride example is shown as a *Tree Bitmap* data structure in Figure 3.5. Note that a pointer to the head of the array of child nodes and a pointer to the set of next hop values corresponding to the set of prefixes in the node are stored along with the bitmaps for each node. By requiring that all child nodes of a single parent node be stored contiguously in memory, the address of a child node can be calculated using a single *Child Node Array Pointer* and an index into that array computed from the extending paths bitmap. The same technique is used to find the associated next hop information for a stored prefix in the node. The *Next Hop Table Pointer* points to the beginning of the contiguous set of next hop values corresponding to the set of stored prefixes

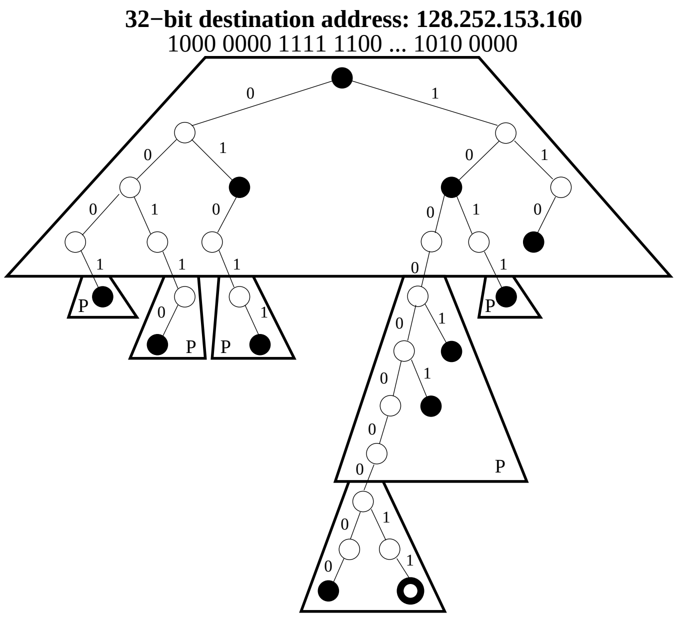


Figure 3.3: IP lookup table represented as a multibit trie. A stride, 4-bits, of the unicast destination address of the IP packet are compared at once, speeding up the lookup process.

in the node. Next hop information for a specific prefix may be fetched by indexing from the pointer location.

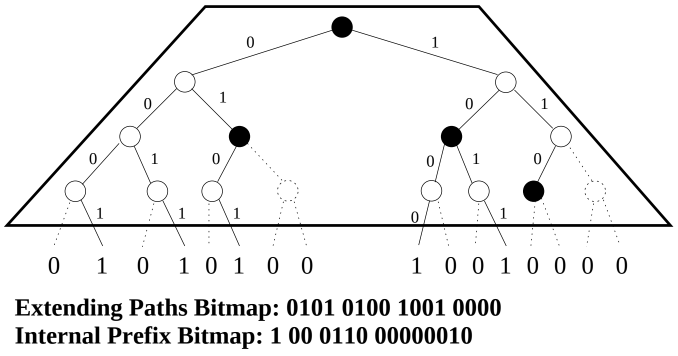

0 1 0 1 0 1 0 0 1 0 0 1 0 0 0 0
Extending Paths Bitmap: 0101 0100 1001 0000
Internal Prefix Bitmap: 1 00 0110 00000010

Figure 3.4: Bitmap coding of a multibit trie node. The internal bitmap represents the stored prefixes in the node while the extending paths bitmap represents the child nodes of the current node.

The index for the *Child Node Array Pointer* leverages a convenient property of the data structure. Note that the numeric value of the nibble of the IP address is also the bit position of the extending path in the *Extending Paths Bitmap*. For example, Address_Nibble(0) = 1000_2 = 8. Note that the eighth bit position, counting from the most significant bit, of the *Extending Paths Bitmap* shown in Figure 3.4 is the extending path bit corresponding to Address_Nibble(0) = 1000_2. The index of the child node is computed by counting the number of ones in the *Extending Paths Bitmap* to the left of this bit position. In the example, the index would be three. This operation of computing

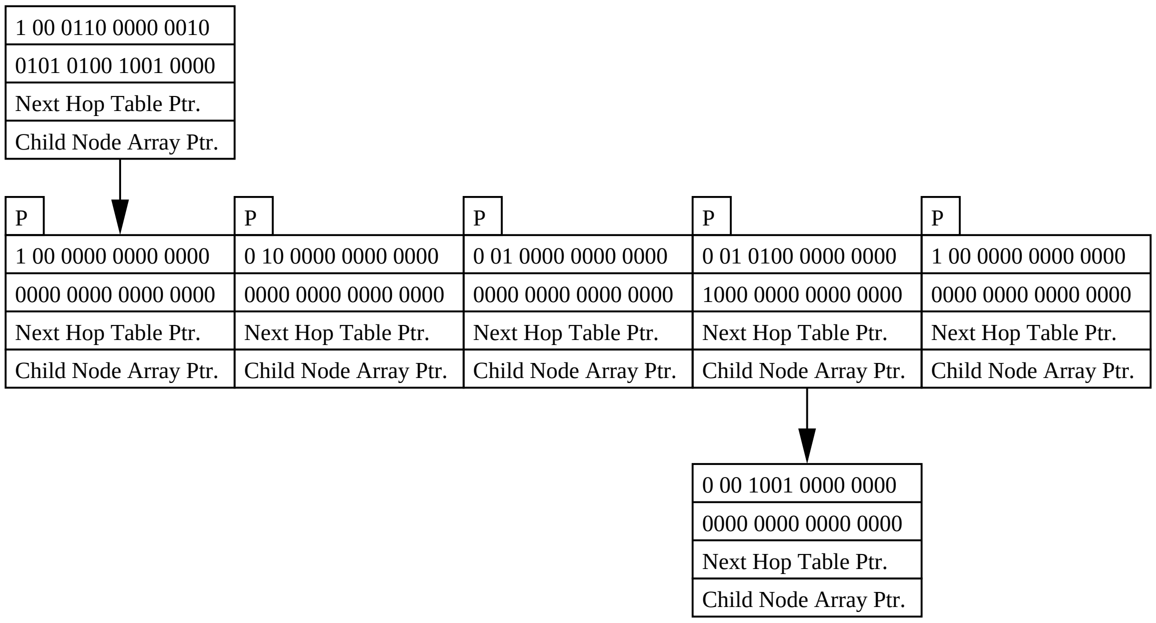


Figure 3.5: IP lookup table represented as a Tree Bitmap. Child nodes are stored contiguously so that a single pointer and an index may be used to locate any child node in the the data structure.

the number of ones to the left of a bit position in a bitmap will be referred to as *CountOnes* and will be used in later discussions.

When there are no valid extending paths, the *Extending Paths Bitmap* is all zeros, the terminal node has been reached and the *Internal Prefix Bitmap* of the node is fetched. A logic operation called *Tree Search* returns the bit position of the longest matching prefix in the *Internal Prefix Bitmap*. *CountOnes* is then used to compute an index for the *Next Hop Table Pointer*, and the next hop information is fetched. If there are no matching prefixes in the *Internal Prefix Bitmap* of the terminal node, then the *Internal Prefix Bitmap* of the most recently visited node that contains a matching prefix is fetched. This node is identified using a data structure optimization called the *Prefix Bit*.

The *Prefix Bit* of a node is set if its parent has any stored prefixes along the path to itself. When searching the data structure, the address of the last node visited is remembered. If the current node's *Prefix Bit* is set, then the address of the last node visited is stored as the best matching node. Setting of the *Prefix Bit* in the example data structure of Figure 3.3 and Figure 3.5 is denoted by a "P".

3.2.1 Split-Trie Optimization

Let s be the stride of the *Tree Bitmap* data structure and let $0 \leq i \leq \frac{32}{s}$ be an integer. In the basic configuration described above (which we will refer to as the "single-trie" configuration) the *Tree Bitmap* data structure stores prefixes of length $i \times s$ in nodes at depth $i + 1$. For example,

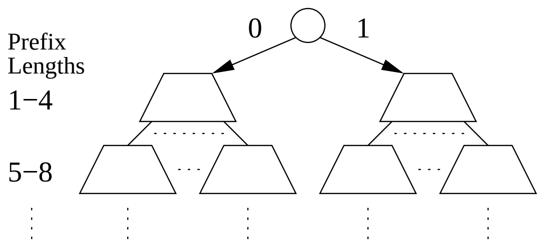


Figure 3.6: Split-trie optimization of the *Tree Bitmap* data structure.

a 24-bit prefix would be stored at level 7 in a data structure with a stride of 4. Examination of publicly available route table statistics show that a large percentage of the prefixes in the table are, in fact, multiples of four. For example, in the Mae-West database used in Section 3.5 for performance testing "multiple of four" prefixes comprise over 66% of the total prefix lengths. Often these prefixes are leaf nodes in the data structure, represented as a multibit node with a single prefix stored at the root in the "single-trie" configuration. Such nodes carry very little information and make poor use of the memory they consume.

The "split-trie" optimization seeks to speed up lookup performance and reduce memory usage for typical databases by shifting "multiple of four" prefixes up one level in the data structure. This can easily be achieved by splitting the multibit trie into two multibit-tries with a root node having a stride of 1 as shown in Figure 3.6. Implementation of this optimization requires two pointers, one to each new multibit root node, and a next hop value for the root node (default route). Searches begin by using the most significant bit of the destination address to decide from which multibit root node to perform the search. For most lookups on typical databases, this optimization saves one memory access per lookup and reduces the memory space per prefix required for the *Tree Bitmap* data structure. The lookup performance and memory utilization of both the "single-trie" and "split-trie" configurations of the FIPL architecture are evaluated in Section 3.5.

3.3 Hardware Design and Implementation

Modular design techniques are employed throughout the FIPL hardware design to provide scalability for various system configurations. Figure 3.7 details the components required to implement FIPL in the Port Processor (PP) of a router. Other components of the router include the Transmission Interfaces (TI), Switch Fabric, and Control Processor (CP). Providing the foundation of the FIPL design, the FIPL engine implements a single instance of a *Tree Bitmap* search. The FIPL Engine Controller may be configured to instantiate multiple FIPL engines in order to scale the lookup throughput with system demands. The FIPL Wrapper extracts the IP addresses from incoming packets and writes them to an address FIFO read by the FIPL Engine Controller. Lookup results are written to a FIFO read by the FIPL Wrapper which accordingly modifies the packet header. The FIPL Wrapper also handles standard IP processing functions such as checksums and header

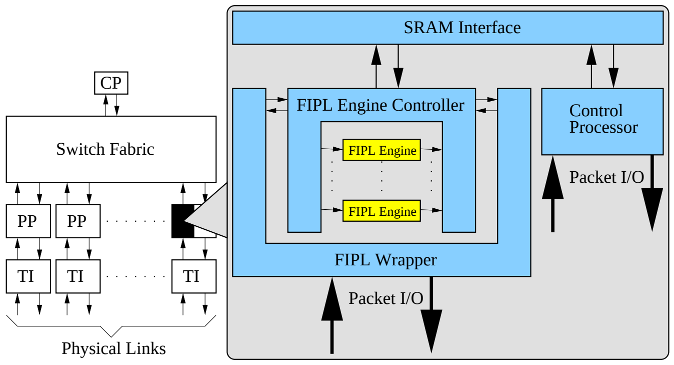


Figure 3.7: Block diagram of router with multi-engine FIPL configuration; detail of FIPL system components in the Port Processor (PP).

field updates. Specifics of the FIPL Wrapper will vary depending upon the type of switching core and transmission format. An on-chip Control Processor receives and processes memory update commands on a dedicated control channel. Memory updates are the result of route add, delete, or modify commands and are sent from the System Management and Control components. Note that the off-chip memory is assumed to be a single port device; hence, an SRAM Interface arbitrates access between the FIPL Engine Controller and Control Processor.

3.3.1 FIPL Engine

Consisting of a few address registers, a simple Finite-State Machine (FSM), and combinational logic, the FIPL Engine is a compact, efficient *Tree Bitmap* search engine. Implementation of the FIPL Engine requires only 450 lines of VHDL code. A dataflow diagram of the FIPL Engine is shown in Figure 3.8. Data arriving from memory is latched into the DATA_IN_REG register *n* clock cycles after issuing a memory read. The value of *n* is determined by the read latency of the memory device plus 2 clock cycles for latching the address out of and the data into the implementation device. The next address issued to memory is latched into the ADDR_OUT_REG *k* clock cycles after data arrives from memory. The value of *k* is determined by the speed at which the implementation device can compute the *next_hop_addr* which is the critical path in the logic. Two counters, *mem_count* and *search_count*, are used to count the number of clock cycles for memory access and address calculation, respectively. Use of multi-cycle paths allows the FIPL engine to scale with implementation device and memory device speeds by simply changing compare values in the finite-state machine logic.

In order to generate *next_hop_addr*:

- TREE_SEARCH generates *prefix_index* which is the bit position of the best-matching prefix stored in the *Internal Prefixes Bitmap*

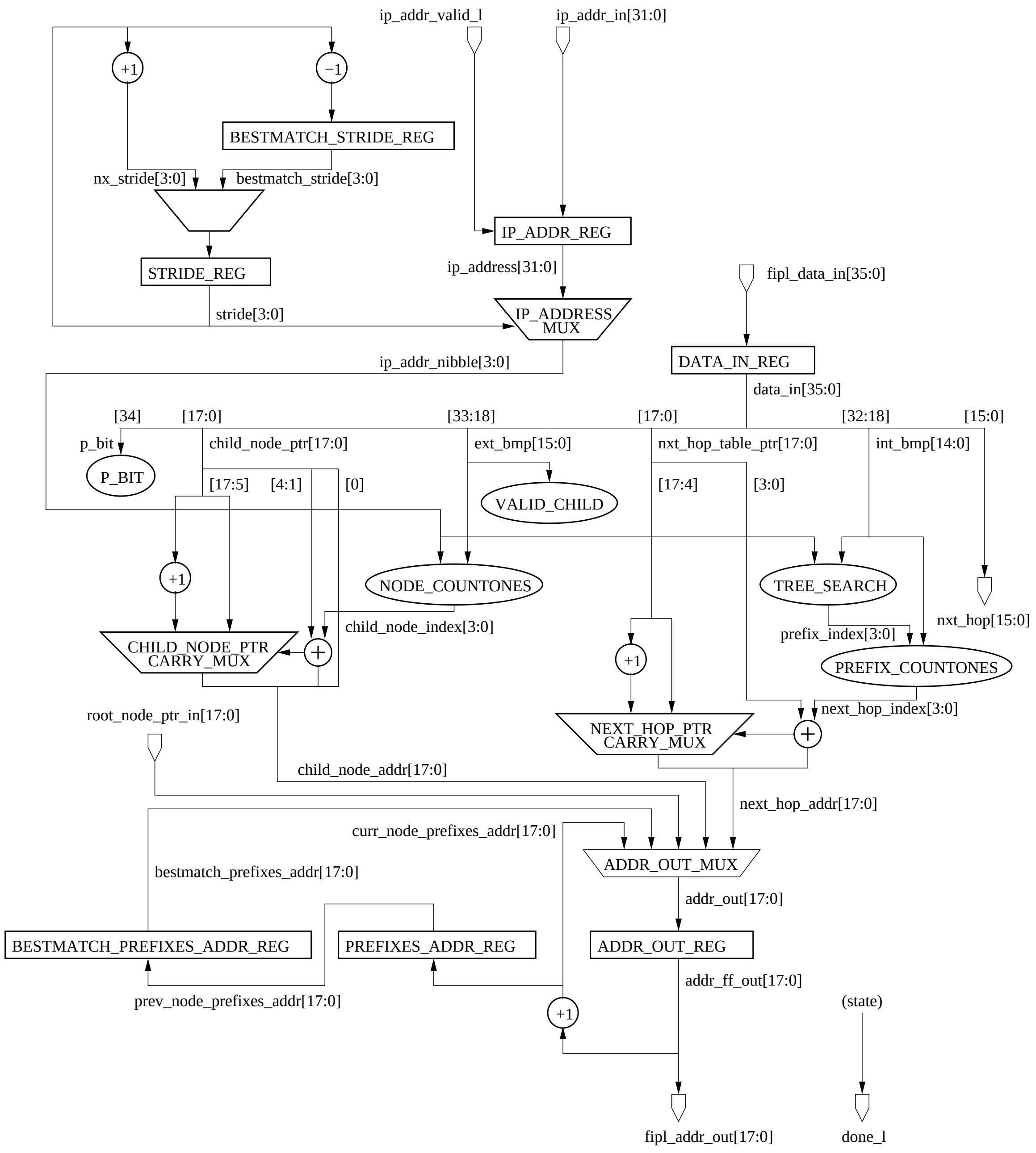


Figure 3.8: FIPL engine dataflow; multi-cycle path from input data flops to output address flops can be scaled according to target device speed; all multiplexor select lines and flip-flop enables implicitly driven by finite-state machine outputs.

- PREFIX_COUNTONES generates *next_hop_index* which is the number of 1's to the left of *prefix_index* in the *Internal Prefixes Bitmap*

- *next_hop_index* is added to the lower four bits of the *Next Hop Table Pointer*

- The carryout of the previous addition is used to select the upper bits of the *Next Hop Table Pointer* or the pre-computed value of the upper bits plus 1

The NODE_COUNTONES and identical fast addition blocks generate the *child_node_addr*, but require less time as the TREE_SEARCH block is not in the path. The ADDR_OUT_MUX selects the next address issued to memory among the addresses for the next root node's *Extending Paths Bitmap* and *Child Node Array Pointer* (*root_node_ptr*), the next child node's *Extending Paths Bitmap* and *Child Node Array Pointer* (*child_node_addr*), the current node's *Internal Prefix Bitmap* and *Next Hop Table Pointer* (*curr_node_prefixes_addr*), the forwarding information for the best-matching prefix (*next_hop_addr*), and the best-matching previous node's *Internal Prefix Bitmap* and *Next Hop Table Pointer* (*bestmatch_prefixes_addr*). Selection is made based upon the current state.

VALID_CHILD examines the *Extending Paths Bitmap* and determines if a child node exists for the current node based on the current nibble of the IP address. The output of VALID_CHILD, *prefix_index*, *mem_count*, and *search_count* determine state transitions as shown in Figure 3.9. The current state and the value of the P_BIT determine the register enables for the BESTMATCH_PREFIXES_ADDR_REG and the BESTMATCH_STRIDE_REG which store the address of the *Internal Prefixes Bitmap* and *Next Hop Table Pointer* of the node containing best-matching prefixes and the associated stride of the IP address, respectively.

3.3.2 FIPL Engine Controller

Leveraging the uniform memory access period of the FIPL Engine, the FIPL Engine Controller interleaves memory accesses of the necessary number of parallel FIPL Engines to scale lookup throughput in order to meet system throughput demands. The scheme centers around a timing wheel with a number of slots equal to the FIPL Engine memory access period. When an address is read from the input FIFO, the next available FIPL Engine is started at the next available time slot. The next available time slot is determined by indexing the current slot time by the known startup latency of a FIPL Engine. For example, assume an access period of 8 clock cycles; hence, the timing wheel has 8 slots numbered 0 through 7. Assume three FIPL Engines are currently performing lookups occupying slots 1, 3, and 4. Furthermore, assume that from the time the IP address is issued to the FIPL Engine to the time the FIPL Engine issues its first memory read is 2 clock cycles; hence, the startup latency is 2 slots. When a new IP address arrives, the next lookup may not be started at slot times 7, 1, or 2 because the first memory read would be issued at slot time 1, 3, or 4, respectively which would interfere with ongoing lookups. Assume the current slot time is 3; therefore, the next FIPL engine is started and slot 5 is marked as occupied.

As previously mentioned, input IP addresses and output forwarding information are passed between the FIPL Engine Controller and the FIPL Wrapper via FIFO interfaces. This design simplifies the design of the FIPL Wrapper by placing the burden of in-order delivery of results on the FIPL Engine Controller. While individual input and output FIFOs could be used for each engine

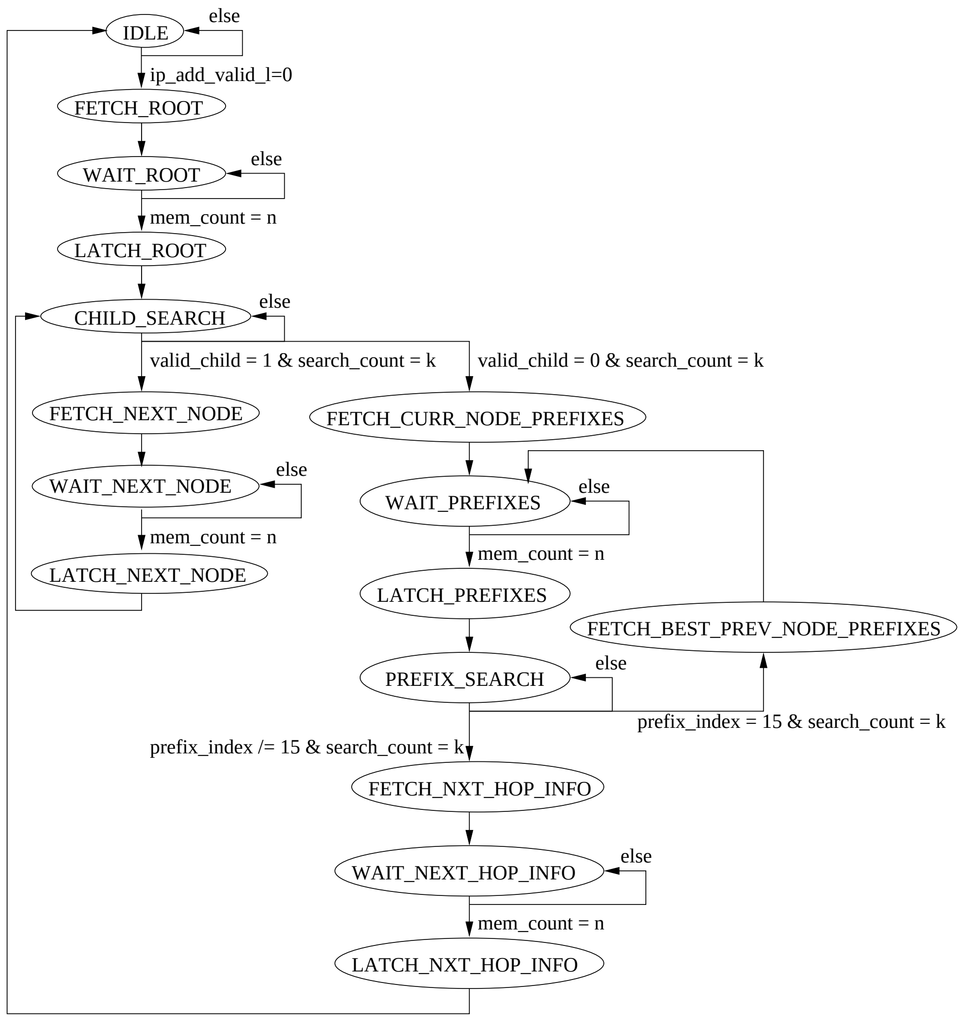


Figure 3.9: FIPL engine state transition diagram.

to prevent head-of-the-line blocking, network designers will usually choose to configure the FIPL Engine Controller assuming worst-case lookups. Also, the performance numbers reported in a subsequent section show that average lookup latency per FIPL Engine increases by less than 3% for an 8-engine configuration; therefore, lookup engine "dead-time" is negligible.

3.3.3 Implementation Platform

FIPL is implemented on open-platform research systems designed and built at Washington University in Saint Louis [34]. The WUGS 20, an 8-port ATM switch providing 20 Gb/s of aggregate throughput, provides a high-performance switching fabric [35]. This switching core is based upon a multi-stage Benes topology, supports up to 2.4 Gb/s link rates, and scales up to 4096 ports for an aggregate throughput of 9.8 Tb/s [36]. Each port of the WUGS 20 can be fitted with a Field-programmable Port eXtender (FPX), a port card of the same form factor as the WUGS transmission interface cards [37]. Each FPX contains two FPGAs, one acting as the Network Interface Device (NID) and the other as the Reprogrammable Application Device (RAD).

The RAD FPGA has access to two 1MB Zero Bus Turnaround (ZBT) SRAMs and two 64MB SDRAM modules providing a flexible platform for implementing high-performance networking applications [38]. To allow for packet reassembly and other processing functions requiring memory resources, the FIPL has access to one of the 1MB ZBT SRAMs which require 18-bit addresses and provide a 36-bit data path with a 2-clock cycle latency. Since this memory is "off-chip" both the address and data lines must be latched at the pads of the FPGA, providing for a total latency to memory of n = 4 clock cycles.

3.3.4 Memory Confi guration

Utilizing a 4-bit stride the *Extending Paths Bitmap* is 16-bits long, occupying less than a half-word of memory. The remaining 20-bits of the word are used for the *Prefix Bit* and *Child Node Array Pointer*; hence, only one memory access is required per node when searching for the terminal node. Likewise, the *Internal Prefix Bitmap* and *Next Hop Table Pointer* may be stored in a single 36-bit word; hence, a single node of the *Tree Bitmap* requires two words of memory space. 131,072 nodes may be stored in one of the 1MB SRAMs providing a maximum of 1,966,080 stored routes. Note that the memory usage per route entry is dependent upon the distribution of prefixes in the data structure. Memory usage for the experimental data structure is reported in the Section 3.5.

3.3.5 Worst-Case Performance

In this configuration, the pathological lookup requires 11 memory accesses: 8 memory accesses to reach the terminal node, 1 memory access to search the sub-tree of the terminal node, 1 memory access to search the sub-tree of the most recent node containing a match, and 1 memory access to fetch the forwarding information associated with the best-matching prefix. Since the FPGAs and SRAMs run on a synchronous 100MHz clock, all single cycle calculations must be completed in less than 10ns. The critical path in the FIPL design, resolving the *next_hop_addr*, requires more than 20 ns when targeted to the RAD FPGA of the FPX, a Xilinx XCV1000E-7; hence, k is set to 3. This provides a total memory access period of 80 ns and requires 8 FIPL engines in order to fully utilize the available memory bandwidth. Theoretical worst-case performance, all lookups requiring 11 memory accesses, ranges from 1,136,363 lookups per second for a single FIPL engine to 9,090,909 lookups per second for eight FIPL engines in this implementation environment.

3.3.6 Hardware Resource Usage

As the WUGS 20 supports a maximum line speed of 2.4 Gb/s, a 4-engine configuration is used in the Washington University system. Due to the ATM switching core, the FIPL Wrapper supports AAL5 encapsulation of IP packets inside of ATM cells [39]. Relative to the Xilinx Virtex 1000E

FPGA used in the FPX, each FIPL Engine utilizes less than 1% of the available logic resources[2]. Configured with 4 FIPL Engines, FIPL Engine Controller utilizes approximately 6% of the logic resources while the FIPL Wrapper utilizes another 2% of the logic resources and 12.5% of the on-chip memory resources. This results in an 8% total logic resource consumption by FIPL. The SRAM Interface and Control Processor which parses control cells and executes memory commands for route updates utilize another 8% of the available logic resources and 2% of the on-chip memory resources. Therefore, all input IP forwarding functions occupy 16% of the logic resources leaving the remaining 84% of the device available for other packet processing functionality.

3.4 System Management and Control Components

System management and control of FIPL in the Washington University system is performed by several distributed components. All components were developed to facilitate further research using the open-platform system. The software components described in this section were developed by Todd Sproull, and their description is included here for completeness.

3.4.1 NCHARGE

NCHARGE is the software component that controls reprogrammable hardware on a switch [40]. Figure 3.10 shows the role of NCHARGE in conjunction with multiple FPX devices within a switch. The software provides connectivity between each FPX and multiple remote software processes via TCP sockets that listen on a well-defined port. Through this port, other software components are able to communicate to the FPX using its specified API. Because each FPX is controlled by an independent NCHARGE software process, distributed management of entire systems can be performed by collecting data from multiple NCHARGE elements. [41].

3.4.2 FIPL Memory Manager

The FIPL Memory Manager is a stand alone C++ application that accepts commands to add, delete, and update routing entries for a hardware-based Internet router. The program maintains the previously discussed *Tree Bitmap* data structure in a shared memory between hardware and software . When a user enters route updates, the FIPL Memory Manager Software returns the corresponding memory updates needed to perform that operation in the FPX hardware.

```
Command options:
        [A]dd
        [D]elete
        [C]hange
```

[2]If targeted to the low-cost Xilinx Spartan-3 family of FPGAs (less than $12 USD for a one million gate device), each engine would cost approximately $0.12 USD.

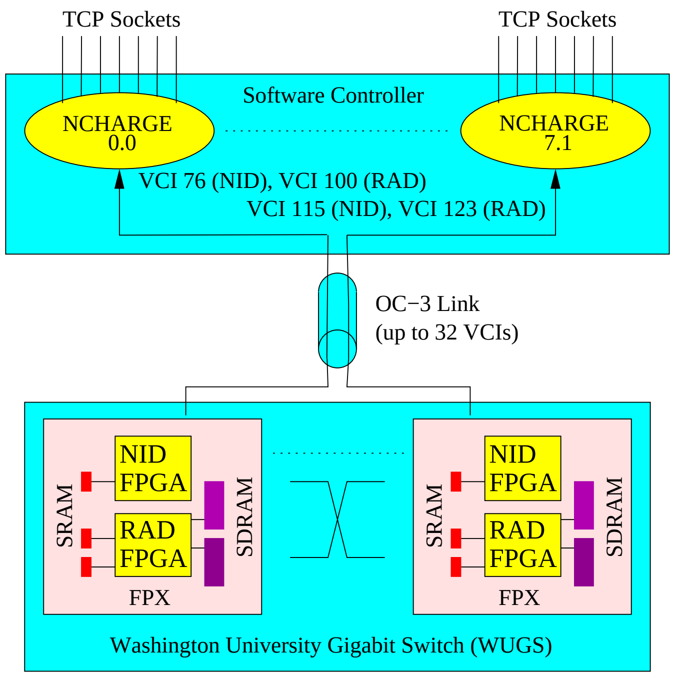

TCP Sockets
TCP Sockets
Software Controller
NCHARGE 0.0
NCHARGE 7.1
VCI 76 (NID), VCI 100 (RAD)
VCI 115 (NID), VCI 123 (RAD)
OC−3 Link (up to 32 VCIs)
SRAM
NID FPGA
RAD FPGA
SDRAM
FPX
SRAM
NID FPGA
RAD FPGA
SDRAM
FPX
Washington University Gigabit Switch (WUGS)

Figure 3.10: Control of the Field-programmable Port eXtender (FPX) via NCHARGE software. Each FPX is controlled by an instance of NCHARGE which provides an API for FPX control via remote software process.

```
        [P]rint
        [M]emoryDump
        [Q]uit

Enter command (h for help): A
You entered add

Enter prefix x.x.x.x/s
(x = 0-255, s is significant bits 0-32) :
192.128.1.1/8

Enter Next Hop value: 4
******
Memory Update Commands:

w36 0 4 2 000000000 100000006
w36 0 2 2 200000004 000000000
w36 0 0 2 000200002 000000000
```

In the example shown here a single add route command requires three 36-bit memory write commands, each consisting of 2 consecutive locations in memory at addresses 4, 2, and 0, respectively.

3.4.3 Sockets Interfaces

In order to access the FIPL Memory Manager as a daemon process, support software needs to be in place to handle standard input and output. Socket software was developed to handle incoming route updates to pass along to the FIPL Memory Manager. A socket interface was also developed to send the resulting output of a memory update to the NCHARGE software. These software processes handling input and output are called Write_Fip and Read_Fip, respectively. Write_Fip is constantly listening on a well known port for incoming route update commands. Once a connection is established the update command is sent as an ASCII character string to Write_Fip. This software prints the string as standard output which is redirected to the standard input of FIPL Memory Manager. The memory update commands needed by NCHARGE software to perform the route update are issued at the output of FIPL Memory Manager. Read_Fip receives these commands as standard input and sends all of the memory updates associated with one route update over a TCP socket to the NCHARGE software.

3.4.4 Remote User Interface

The current interface for performing route updates is via a web page that provides a simple interface for user interaction. The user is able to submit single route updates or a batch job of multiple routes in a file. Another option available to users is the ability to define unique control cells. This is done through the use of software modules that are loaded into the NCHARGE system.

In the current FIPL Module, a web page has been designed to provide a simple interface for issuing FIPL control commands, such as changing the *Root Node Pointer*. The web page also provides access to a vast database of sample route table entries taken from the Internet Performance Measurement and Analysis project's website [42]. This website provides daily snapshots of Internet backbone routing tables including traditional Class A, B, and C addresses. Selecting the download option from the FIPL web page executes a Perl script to fetch the router snapshots from the database. The Perl script then parses the files and generates an output file that is readable by the Fast IP Lookup Memory Manager.

3.4.5 Command Flow

The overall flow of data with FIPL and NCHARGE is shown in Figure 3.11. Suppose a user wishes to add a route to the database. The user first submits either a single command or submits a file containing multiple route updates. Data submitted from the web page, Figure 3.12, is passed

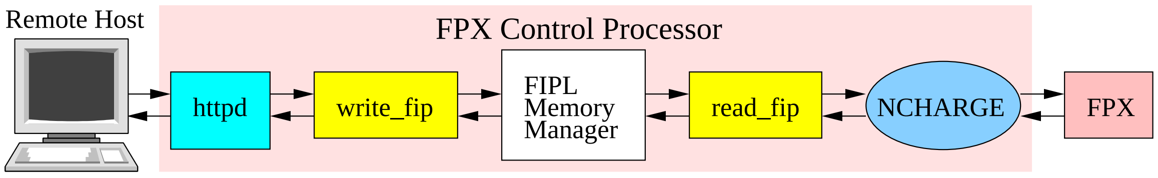
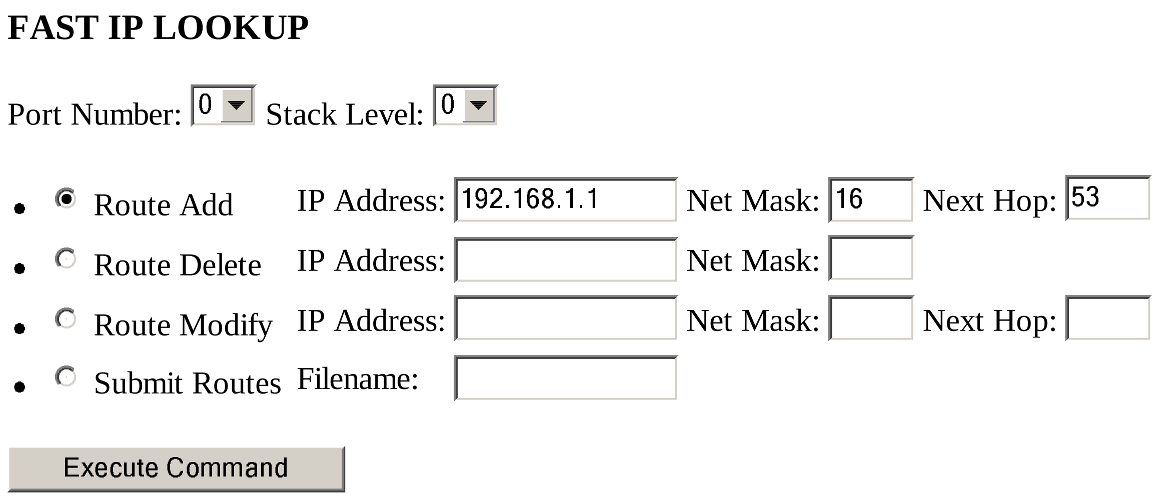


Figure 3.11: Command flow for control of FIPL via a remote host.



FAST IP LOOKUP

Port Number: 0 Stack Level: 0

● Route Add	IP Address:	192.168.1.1	Net Mask:	16	Next Hop:	53
○ Route Delete	IP Address:		Net Mask:			
○ Route Modify	IP Address:		Net Mask:		Next Hop:	
○ Submit Routes	Filename:					

Figure 3.12: FPX Web Interface for FIPL route updates.

to the Web Server as a form. Local scripts process the form and generate an Add Route command that the software understands. These commands are ASCII strings in the form "Add route $A_1.A_2.A_3.A_4$/netmask nexthop". The script then sets up a TCP Socket and transmits each command to the Write_Fip software process. As mentioned before Write_fip listens on a TCP port and relays messages to standard output in order to communicate with the FIPL Memory Manager. FIPL Memory Manager takes the standard input and processes the route command in order to generate memory updates for an FPX board. Each memory update is then passed as standard output to the Read_Fip process.

After this process collects memory updates it establishes a TCP connection with NCHARGE to transmit the commands. Read_Fip is able to detect individual route commands and issues the set of memory updates associated with each. This prevents Read_Fip from creating a socket for every memory update. From here memory updates are sent to NCHARGE software process to be packed into control cells to send to the FPX. NCHARGE packs as many memory commands as it can fit into a 53 byte ATM cell while preserving order between commands. NCHARGE sends these control cells using a stop-and-wait protocol to ensure correctness, then issues a response message to the user.

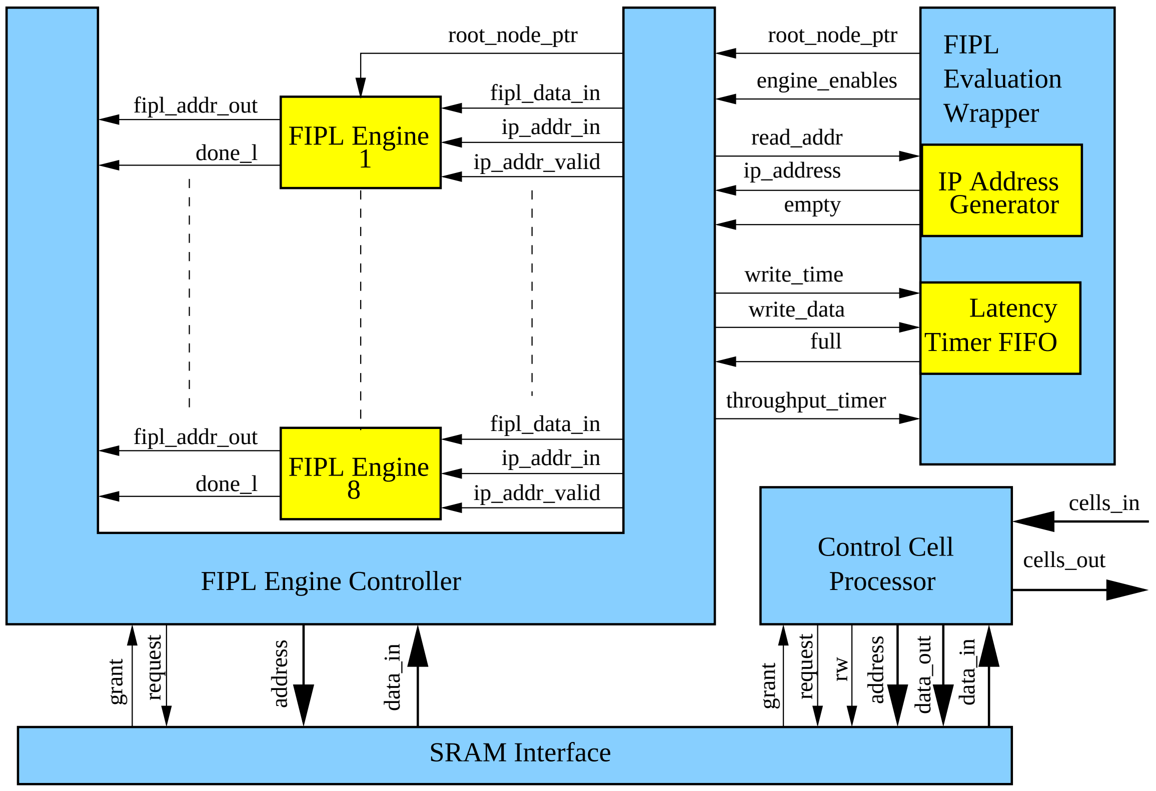


Figure 3.13: Block diagram of FIPL evaluation environment.

3.5 Performance Measurements

While the worst-case performance of FIPL is deterministic, an evaluation environment was developed in order to benchmark average FIPL performance on actual router databases. The evaluation environment was used to extract lookup and update performance as the number of parallel FIPL Engines was scaled up, as well as determine the performance gain of the split-trie optimization. As shown in Figure 3.13, the evaluation environment includes a modified FIPL Engine Controller, 8 FIPL Engines, and a FIPL Evaluation Wrapper. The FIPL Evaluation Wrapper includes an IP Address Generator which uses on-chip BlockRAMs in the Xilinx FPGA to implement storage for 16,384 IPv4 destination addresses. The IP Address Generator interfaces to the FIPL Engine controller like a FIFO. When a test run is initiated, an empty flag is driven to FALSE until all 16,384 addresses are read.

Control cells sent to the FIPL Evaluation Wrapper initiate test runs of 16,384 lookups and specify how many FIPL Engines should be used during the test run. The FIPL Engine Controller contains a latency timer for each FIPL Engine and a throughput timer that measures the number of clock cycles required to complete each lookup and the test run of 16,384 addresses, respectively. Latency timer values are written to a FIFO upon completion of each lookup. The FIPL Evaluation Wrapper packs latency timer values into control cells which are sent back to the system control

Table 3.1: Memory usage for the *Tree Bitmap* data structure and next hop information using a snapshot of the Mae-West database from March 15, 2002 consisting of 27,609 routes.

Type	*Total (bytes)*	*Total (bytes/prefix)*	*Next Hop (bytes)*	*Next Hop (bytes/prefix)*	*Tree Bitmap (bytes)*	*Tree Bitmap (bytes/prefix)*
Single-Trie	409,937	14.8	124,241	4.5	285,696	**10.3**
Split-Trie	298,822	10.8	124,241	4.5	174,582	**6.3**

software where the contents are dumped to a file. The throughput timer value is included in the final control cell.

A snapshot of the Mae-West database from March 15, 2002 consisting of 27,609 routes was used for all tests. The on-chip memory read by the IP Address Generator was initialized with 16,384 IPv4 destination addresses created via random selections from the route table snapshot. Two evaluation environments were synthesized, one including "single-trie" FIPL engines and one including "split-trie" FIPL engines. Each evaluation environment was downloaded to the RAD FPGA of the FPX and subjected to a series of test vectors.

3.5.1 Memory Utilization

Two *Tree Bitmap* data structures were generated from the Mae-West snapshot, one for the "single-trie" FIPL engines and one for the "split-trie" FIPL engines. As previously mentioned, our experimental implementation allocated an entire 36-bit memory word for next hop information. As shown in Table 3.1, the total memory utilization for each variation of the data-structure is broken down into usage for the *Tree Bitmap* and next hop information. Note that the size of the *Tree Bitmap* data structure is reduced by approximately 30% via the split-trie optimization.

3.5.2 Lookup Rate

The "single-trie" and "split-trie" evaluation environments were downloaded to the RAD FPGA of the FPX and subjected to a series of test vectors. Prior to each test run, the *Tree Bitmap* data structure generated from the Mae-West database of 27,609 routes was loaded into the off-chip SRAM. The on-chip memory read by the IP Address Generator was initialized with 16,384 IPv4 destination addresses created via random selections from the route table snapshot. Test runs were initiated using configurations of 1 through 8 engines.

Each evaluation environment was first tested with no intervening updates. Figure 3.14 plots the number of lookups per second versus the number of parallel FIPL engines for the single-trie and split-trie versions. The theoretical worst-case performance is also included for reference. With no intervening update traffic, lookup throughput for the "single-trie" configuration ranged from 1.46 million lookups per second for a single FIPL engine to 10.09 million lookups per second for 8 FIPL engines; an 11% increase in performance over the theoretical worst-case. Under identical

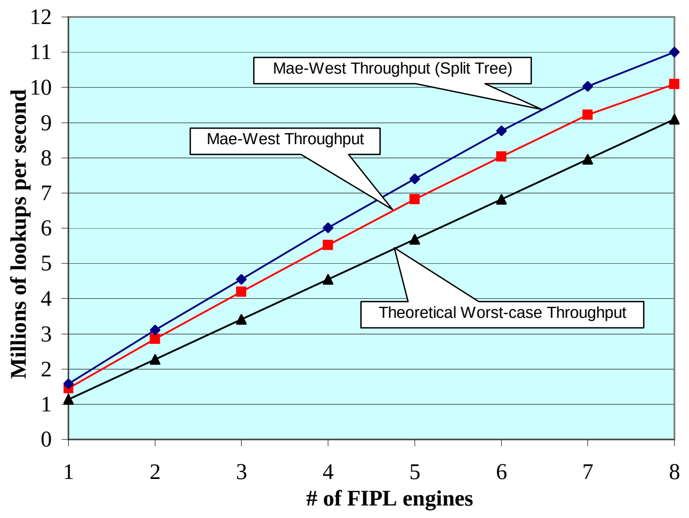


Figure 3.14: FIPL performance: measurements used a snapshot of the Mae-West database from March 15, 2002 consisting of 27,609 routes. Input IPv4 destination addresses were created by randomly selecting 16,384 prefixes from the Mae-West database.

conditions, lookup throughput for the "split-trie" configuration ranged from 1.58 million lookups per second for a single FIPL engine to 11 million lookups per second for 8 FIPL engines; a 9% increase in performance over the "single-trie" configuration. Average lookup latency for "single-trie" FIPL engines ranged from 656 ns for a single FIPL engine to 674 ns for 8 FIPL engines. Average lookup latency for "split-trie" FIPL engines ranged from 603 ns for a single FIPL engine to 619 ns for 8 FIPL engines.

In order to evaluate performance under update load, updates were transmitted to the evaluation environment at various rates during test runs. Update traffic consisted of an alternating pattern of a 24-bit prefix and a 24-bit prefix delete. For the the "single-trie" configuration, the 24-bit prefix add required 25 memory write operations which were packed into 4 control cells. The 24-bit prefix delete required 14 memory write operations which were packed into 3 control cells. For the the "split-trie" configuration, the 24-bit prefix add required 21 memory write operations which were packed into 4 control cells. The 24-bit prefix delete required 12 memory write operations which were packed into 2 control cells. Test runs were executed for both configurations with updates rates ranging from 1,000 updates per second to 1,000,000 updates per second. Note that the upper end of the range, one update per microsecond, represents a highly unrealistic situation as update frequencies rarely exceed 1,000 updates per second.

Results of test runs of the "single-trie" FIPL configuration with intervening update traffic are shown in Figure 3.15. Results of test runs of the "split-trie" FIPL configuration with intervening update traffic are shown in Figure 3.16. For both configurations, update frequencies up to 10,000 updates per second had no noticeable effect on lookup throughput performance. For an update frequency of 100,000 updates per second, the "single-trie" configuration exhibited a maximum

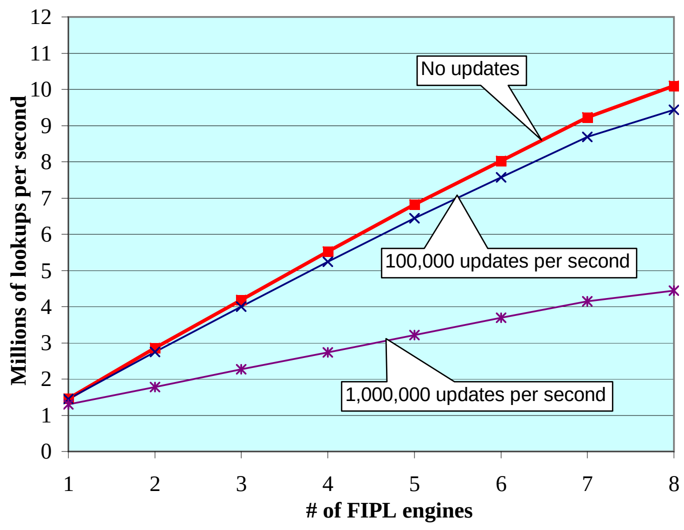


Figure 3.15: FIPL performance under update load: measurements used a snapshot of the Mae-West database from March 15, 2002 consisting of 27,609 routes. Input IPv4 destination addresses were created by randomly selecting 16,384 prefixes from the Mae-West database. Updates consisted of alternating addition and deletion of a 24-bit prefix.

performance degradation of 6.5% while the "split-trie" throughput was reduced by 7.2%. For an update frequency of 1,000,000 updates per second, the "single-trie" configuration exhibited a maximum performance degradation of 56% while the "split-trie" throughput was reduced by 58.9%. FIPL not only demonstrates no noticeable performance degradation under normal update loads, but it also remains robust under excessive update loads.

Based on the test results, a FIPL configuration employing four parallel search engines was synthesized for the WUGS/FPX research platform in order to support 2 Gb/s links. Utilizing custom traffic generators and bandwidth monitoring software, throughput for minimum length packets was measured at 1.988 Gb/s. Note that the total system throughput is limited by the 32-bit WUGS/FPX interface operating at 62.5 MHz. Additional tests injected route updates to measure update performance while maintaining 2 Gb/s of offered lookup traffic. The FIPL configuration experienced only 12% performance degradation at update rates of 200,000 updates per second.

3.6 Towards Better Performance

Ongoing research efforts seek to leverage the components and insights gained from implementing Fast IP Lookup (FIPL) on the open research platforms developed at Washington University in Saint Louis [33, 43]. In this section we discuss two optimizations that can significantly improve the performance of the FIPL engine. In Section 3.6.1 we discuss design and device optimizations to reduce the critical path delay in the FIPL engine. In Section 3.6.2 we apply a common data structure optimization to reduce the worst case number of off-chip memory accesses.

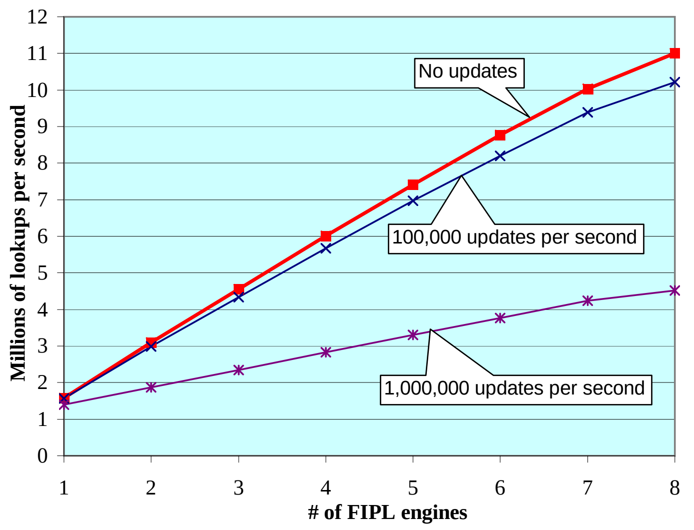


Figure 3.16: FIPL Split-Trie performance under update load: measurements used a snapshot of the Mae-West database from March 15, 2002 consisting of 27,609 routes. Input IPv4 destination addresses were created by randomly selecting 16,384 prefixes from the Mae-West database. Updates consisted of alternating addition and deletion of a 24-bit prefix.

3.6.1 Implementation Optimizations

Coupled with advances in FPGA device technology, implementation optimizations of critical paths in the FIPL engine circuit hold promise of increasing the system clock frequency in order to take full advantage of the memory bandwidth offered by modern SRAMs. Existing SRAMs are capable of operating at 200 MHz or faster; note that modern FPGAs are capable of running at this frequency [44] and no throughput is gained via an ASIC implementation since off-chip SRAM accesses are the performance bottleneck. Doubling of the clock frequency of FIPL directly translates to a factor of two increase in lookup performance to a guaranteed worst case throughput of over 18.2 million lookups per second. DDR SRAMs essentially double the size of the memory word accessed per clock cycle; this provides the opportunity for further optimizations by allowing us to double the amount of information stored in node. We can take advantage of this by extending the stride length of nodes and/or performing path compression.

3.6.2 Root Node Extension & Caching

By caching the root node in on-chip memory and extending its stride length, the number of off-chip memory accesses can be reduced. Extending the stride length of the root node increases the number of bits required for the extending paths and internal prefix bitmaps. The increase in the number of extending paths also requires a larger chunk of contiguous memory for storing the second level of multibit nodes in the child node array. In general, the size of the bitmap required for a stride of

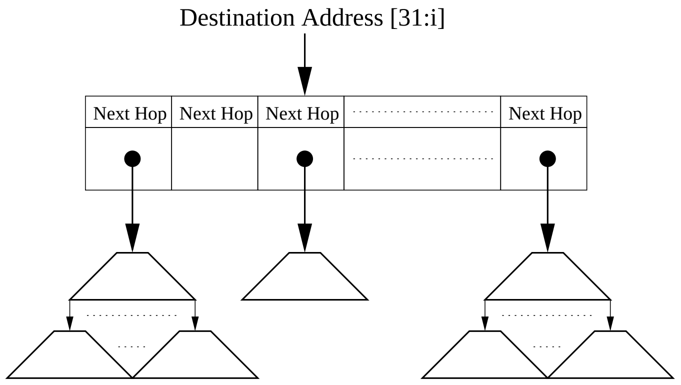


Figure 3.17: Root node extension using an on-chip array and multiple sub-tries.

length n is $2^{n+1} - 1$ bits. The maximum number of contiguous memory spaces needed for the child node array is 2^n.

Selecting the stride length for the cached root node mainly depends upon the amount of available on-chip memory and logic. In the case of ample on-chip memory, one would still want to bound the stride length to prevent the amount of contiguous memory spaces necessary for the child node array from becoming too large. Selection of a stride length which is a factor of four plus one (i.e. 5, 9, 13, ...) provides the favorable property of implementing the "multiple-of-stride" case efficiently. Selecting a root node stride length of eight requires extending paths and internal prefix bitmap lengths of 8192 and 8191 bits, respectively. Given that current generations of FPGAs implement 16kb blocks of memory, the bitmap storage requirement does not seem prohibitively high. However, the *CountOnes* and *Tree Search* functions consume exorbitant amounts of logic for such large bitmaps.

Another approach is to simply represent the root node as an on-chip array indexed by the first i bits of the destination address, where i is determined by the stride length of the root node. This technique was formally introduced by Lampson, Srinivasan, and Varghese [23] and is discussed in Section 2.2.4. As shown in 3.17, each array entry stores the next hop information for the best-matching prefix in the n-bit path represented by the index, as well as a pointer to an extending path sub-tree. Searches simply examine the extending path sub-tree pointer to see if a sub-tree exists for the given address. This may be done by designating a null pointer value or using a valid extending path bit. If no extending path sub-tree exists, the next hop information stored in the on-chip array entry is applied to the packet. If an extending path sub-tree exists, the extending path sub-tree pointer is used to fetch the "root node" of the extending path sub-tree and the search continues in the normal Tree Bitmap fashion. If no matching prefix is found in the sub-tree, the next hop information stored in the on-chip array entry is applied to the packet.

Obviously, the performance gain comes at the cost of on-chip resource usage and update speed, as a single update may require updates to several array slots. Table 3.2 shows the following:

- *Array Size (AS)*: number of array slots.

Table 3.2: Memory usage for root node array optimization.

Stride (i)	*As*	*On-CM (bits)*	*WC Off-CMA*	*WC Tp (10ns,5ns)*
4	16	512	10	10, 20
5	32	1024	10	10, 20
8	256	8,192	9	11.1, 22.2
9	512	16,384	9	11.1, 22.2
12	4096	131,072	8	12.5, 25
13	8192	262,144	8	12.5, 25

- *On-chip Memory (On-CM)*: the amount of on-chip memory needed in order to allocate the root node array.
- *Worst Case Off-chip Memory Accesses (WC Off-CMA)*: the amount of off-chip memory required to store sub-trees.
- *Worst Case Throughput (WC Tp)*: millions of lookups per second assuming a 100MHz clock (T=10ns) and 200MHz clock (T=5ns).

We assume that all sub-tree pointers and next hop information are 16-bits each. If more next-hop information is required, the on-chip memory may be scaled accordingly or the information may be stored off-chip and the 16-bit field used as a pointer. Note that extending the root node stride to 9 still allows the initial array to fit in a single 18kb BlockRAM in the current generation of FPGAs [44].

3.7 Related Work

One way to accelerate IP packet forwarding is to avoid performing IP lookups. Protocols such as IP-Switching and MPLS/Tag-Switching attempt to avoid lookups in the network core by establishing a path between ingress and egress routers [45, 46, 47, 48]. In all cases, the decision at core routers is simplified to an indexed or exact match lookup on a table of ATM virtual circuit identifiers, “tags”, or “labels” depending on the protocol in use. While these protocols have enjoyed limited success, two major issues prevent them from obviating longest prefix match lookups. First, the ingress and egress routers are still required to perform a full IP lookup in order to make a routing decision. Even if ingress and egress routers are restricted to network edges, increasing bandwidth demands require high performance IP lookup techniques. The second major issue is coordination between multiple Autonomous Systems (AS). Due to issues like security, trust, resource allocations, and differing views of the network, end-to-end coordination in the Internet is difficult. Terminating and re-establishing connections at AS boundaries requires full routing decisions by each AS router at the boundary.

Numerous research and commercial IP lookup techniques exist. On the commercial front, several companies have developed high speed lookup techniques using Ternary Content Addressable Memory (TCAM) and Application Specific Integrated Circuit (ASIC) technologies. Some current products, targeting OC-768 (40 Gb/s) and quad OC-192 (10 Gb/s) link configurations, claim throughputs of over 100 million lookups per second and storage for 100 million entries [49]. However, the advertised performance comes at an extreme cost. 16 ASICs containing embedded TCAMs must be cascaded in order to achieve the advertised throughput and support the more realistic storage capacity of one million table entries. We provide a more detailed analysis of the size, power consumption, and cost of TCAM devices in Section 4.2.2.

An overview of the most prominent Longest Prefix Matching algorithms is provided in Section 2.2. The *Lulea* algorithm is the most similar of published algorithms to the *Tree Bitmap* algorithm used in our FIPL engine [21]. Like *Tree Bitmap*, the *Lulea* algorithm uses a type of compressed trie to limit the number of memory accesses required to traverse the data structure. While similar at a high level, the two algorithms differ in a variety of specifics, that allow *Tree Bitmap* to offer comparable lookup performance with more efficient support of dynamic incremental updates. Due to its relative simplicity, *Tree Bitmap* is also more amenable to hardware implementation. A detailed comparison of the *Tree Bitmap* algorithm to other published lookup techniques is provided in [11]; but, we highlight the most important distinctions here.

The design focus of the *Lulea* algorithm is to provide high lookup rates using a software implementation on a general purpose processor or network processor. In order to accomplish this, the algorithm employs compression techniques that allow the forwarding table to fit in a processor's cache and limit computations to simple indexing operations. The lack of support for dynamic incremental updates is a byproduct of the focus on extremely compact table size and limited number of memory accesses. The *Lulea* algorithm begins by constructing a three level multibit trie with strides of 16, 8, and 8. Searching each level of the *Lulea* data structure may require up to four memory accesses, hence the worst case number of memory accesses is 12. Recall that our implementation of *Tree Bitmap* is an eight level multibit trie with a constant stride of 4 requiring at most 11 memory accesses. Note that the *Tree Bitmap* algorithm does not preclude the use of variable strides, and as we show in Section 3.6.2 the worst case number of memory accesses can be reduced via further optimization.

The *Lulea* encoding requires that the trie be *complete*, thus every node must have two or no children. This requirement yields the following property: every prefix is stored in a leaf and every leaf stores a prefix. The algorithm then employs an implicit form of *leaf pushing* [19] that removes redundant entries from the set of stored values. In essence, the best matching prefix or pointer to the next multibit node is pre-computed for each possible path through each multibit node. For each multibit node, this information is encoded using arrays of *code words* and *base indices*. A precomputed table of indices is used to compute the pointer to the next hop information or next multibit node along the search path.

In contrast, the *Tree Bitmap* algorithm avoids pre-computation by computing pointer indices "on-the-fly" using the *CountOnes* operation. It also avoids pre-computation in the form of *leaf pushing* by explicitly representing the set of prefixes stored in each multibit node via bitmap encoding. These design choices allow *Tree Bitmap* to remain competitively memory efficient while supporting dynamic incremental updates. While the requirement that all child nodes of a parent node be stored contiguously slightly complicates the memory management, updates to the forwarding table typically require reads or writes to only a few memory words. In summary, *Tree Bitmap* offers equal or better lookup performance with comparable memory requirements. Our implementation provides concrete evidence that *Tree Bitmap* is a viable option for high-performance systems and can supporting dynamic incremental updates at rates far exceeding the current maximum update rates observed in the Internet.

3.8 Discussion

IP address lookup is one of the primary functions of the router and often is a significant performance bottleneck. In response, we have presented the Fast Internet Protocol Lookup (FIPL) search engine which utilizes Eatherton and Dittia's *Tree Bitmap* algorithm. Striking a favorable balance between lookup and update performance, memory efficiency, and hardware resource usage, each FIPL engine supports over 500 Mb/s of link traffic while consuming less than 1% of available logic resources and approximately 10 bytes of memory per entry. Utilizing only a fraction of a reconfigurable logic device and a single commodity SRAM, FIPL offers an attractive alternative to expensive commercial solutions employing multiple Content Addressable Memory (CAM) devices and Application Specific Integrated Circuits (ASICs). By providing high-performance with low resource consumption, FIPL is a prime candidate for a System-On-Chip (SoC) route lookup solution or an LPM engine in a packet classification device.

Chapter 4

Multiple Field Search Techniques

If we knew what it was we were doing, it would not be called research, would it?
Albert Einstein

In this chapter we provide a survey and taxonomy of the major advances in multiple field search techniques for packet classification. Due to the complexity of the search, packet classification is often a performance bottleneck in network infrastructure; therefore, it has received much attention in the research community. In general, there have been two major threads of research addressing this problem: algorithmic and architectural. A few pioneering groups of researchers posed the problem, provided complexity bounds, and offered a collection of algorithmic solutions [50, 51, 52, 53]. Subsequently, the design space has been vigorously explored by many offering new algorithms and improvements upon existing algorithms [54, 27, 29]. Given the inability of early algorithms to meet performance constraints imposed by high speed links, researchers in industry and academia devised architectural solutions to the problem. This thread of research produced the most widely-used packet classification device technology, Ternary Content Addressable Memory (TCAM) [55, 56, 17, 57].

Some of the most promising algorithmic research embraces the practice of leveraging the statistical structure of filter sets to improve average performance [50, 54, 58, 51, 59]. Several algorithms in this class are amenable to high-performance hardware implementation. We discuss these observations in more detail and provide motivation for packet classification on larger numbers of fields in Chapter 5. New architectural research combines intelligent algorithms and novel architectures to eliminate many of the unfavorable characteristics of current TCAMs [32]. We observe that the community appears to be converging on a combined algorithmic and architectural approach to the problem [32, 60, 28]. In order to lend structure to our discussion, we develop a taxonomy in Section 4.1 that frames each technique according to its high-level approach to the problem. The presentation of this taxonomy is followed by a survey of the seminal and recent solutions to the packet classification problem. Throughout our presentation we attempt to use a minimal set of running examples to provide continuity to the presentation and highlight the distinctions among the various solutions.

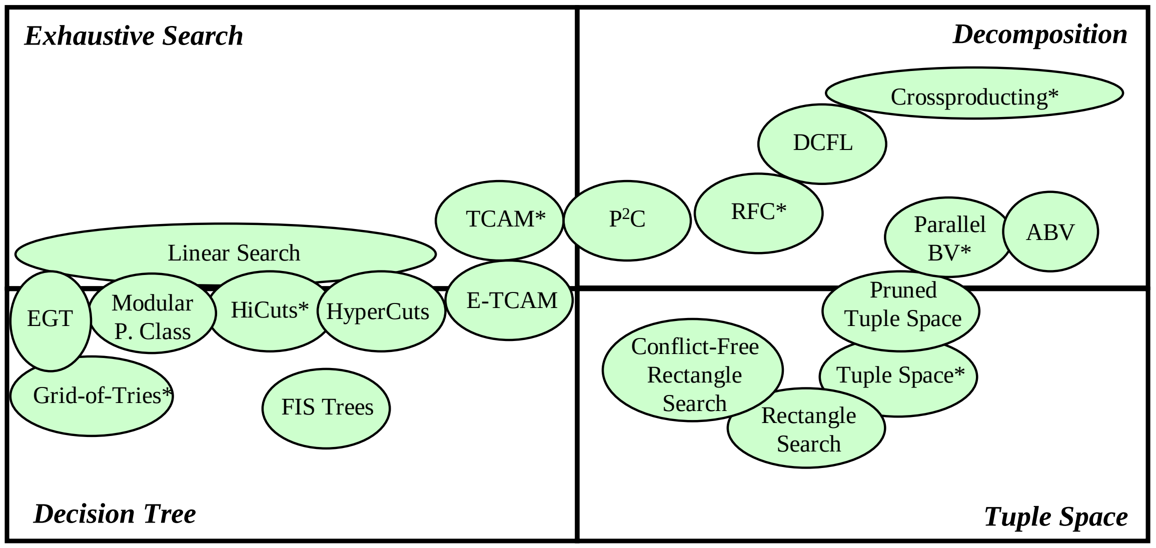


Figure 4.1: Taxonomy of multiple field search techniques for packet classification; adjacent techniques are related; hybrid techniques overlap quadrant boundaries; $*$ denotes a seminal technique.

4.1 Taxonomy

Given the subtle differences in formalizing the problem and the enormous need for good solutions, numerous algorithms and architectures for packet classification have been proposed. Rather than categorize techniques based on their performance, memory requirements, or scaling properties, we present a taxonomy that breaks the design space into four regions based on the high-level approach to the problem. We feel that such a taxonomy is useful, as a number of the salient features and properties of a packet classification technique are consequences of the high-level approach. We frame each technique as employing one or a blend of the following high-level approaches to finding the best matching filter or filters for a given packet:

- **Exhaustive Search**: examine all entries in the filter set
- **Decision Tree**: construct a decision tree from the filters in the filter set and use the packet fields to traverse the decision tree
- **Decomposition**: decompose the multiple field search into instances of single field searches, perform independent searches on each packet field, then combine the results
- **Tuple Space**: partition the filter set according to the number of specified bits in the filters, probe the partitions or a subset of the partitions using simple exact match searches

Figure 4.1 presents a visualization of our taxonomy. Several techniques, including a few of the most promising ones, employ more than one approach. This is reflected in Figure 4.1 by overlapping quadrant boundaries. Relationships among techniques are reflected by proximity.

In the following sections, we discuss each high-level approach in more detail along with the performance consequences of each. We also present a survey of the specific techniques using each approach. We note that the choice of high-level approach largely dictates the optimal architecture for high-performance implementation and a number of the scaling properties. Commonly, papers introducing new search techniques focus on clearly describing the algorithm, extracting scaling properties, and presenting some form of simulation results to reinforce baseline performance claims. Seldom is paper "real estate" devoted to flushing out the details of a high-performance implementation; thus, our taxonomy provides valuable insight into the potential of these techniques. In general, the choice of high-level approach does not preclude a technique from taking advantage of the statistical structure of the filter set; thus, we address this aspect of each technique individually.

4.2 Exhaustive Search

The most rudimentary solution to any searching problem is simply to search through all entries in the set. For the purpose of our discussion, assume that the set may be divided into a number of subsets to be searched independently. The two most common embodiments of the exhaustive search approach for packet classification are a linear search through a list of filters or a massively parallel search over the set of filters. Interestingly, these two solutions represent the extremes of the performance spectrum, 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 such that each subset contains only one entry. We discuss both of these solutions in more detail below. The intermediate option of exhaustively searching subsets containing more than one entry is not a common solution, thus we do not discuss it directly. It is important to note that a number of recent solutions using the decision tree approach use a linear search over a bounded subset of filters as the final step. These solutions are discussed in Section 4.3.

Computational resource requirements for exhaustive search generally scale linearly with the degree of parallelism. Likewise, the realized throughput of the solution is proportional to the degree of parallelism. Linear search requires the minimum amount of computation resources while TCAMs require the maximum, thus linear search and TCAM provide the lowest and highest performance exhaustive search techniques, respectively.

Given that each filter is explicitly stored once, exhaustive search techniques enjoy a favorable linear memory requirement, $O(N)$, where N is the number of filters in the filter set. Here we seek to challenge a commonly held view that the $O(N)$ storage requirement enjoyed by these techniques is optimal. We address this issue by considering the redundancy among filter fields and the number of fields in a filter. These are vital parameters when considering a third dimension of scaling: filter size. By filter size we mean the number of bits required to specify a filter. A filter using the standard IPv4 5-tuple requires about 168 bits to specify explicitly. With that number of bits, we can specify 2^{168} distinct filters. Typical filter sets contain fewer than 2^{20} filters, suggesting that there is potential for a factor of eight savings in memory.

Here we illustrate a simple encoding scheme that represents filters in a filter set more efficiently than explicitly storing them. Let a filter be defined by fields $f_1 \ldots f_d$ where each field f_i requires b_i bits to specify. For example, a filter may be defined by a source address prefix requiring 64 bits[1], a destination address prefix requiring 64 bits, a protocol number requiring 8 bits, etc. By this definition, the memory requirement for the exhaustive search approach is

$$N \sum_{i=1}^{d} b_i \tag{4.1}$$

Now let $u_1 \ldots u_d$ be the number of unique field values in the filter set for each filter field i. If each filter in the filter set contained a unique value in each field, then exhaustive search would have an optimal storage requirement. Note that in order for a filter to be unique, it only must differ from each filter in the filter set by one bit. As we discuss in Chapter 5, there is significant redundancy among filter fields. Through efficient encoding, the storage requirement can be reduced from linear in the number of filters to logarithmic in the number of unique fields. Consider the example shown in Figure 4.2. Note that all 8 filters are unique, however there are only two unique values for each field for all filters in the filter set. In order to represent the filter set, we only need to store the unique values for each field once. As shown in Figure 4.2, we assign a locally unique label to each unique field value. The number of bits required for each label is $\lg(u_i)$, only one bit in our example. Note that each filter in the filter set can now be represented using the labels for its constituent fields. Using this encoding technique, the memory requirement becomes

$$\sum_{i=1}^{d} (u_i \times b_i) + N \sum_{i=1}^{d} \lg u_i \tag{4.2}$$

The first term accounts for the storage of unique fields and the second term accounts for the storage of the encoded filters. The savings factor for a given filter set is simply the ratio of Equation 4.1 and Equation 4.2. For simplicity, let $b_i = b \forall i$ and let $u_i = u \forall i$; the savings factor is:

$$\frac{Nb}{ub + N \lg u} \tag{4.3}$$

In order for the savings factor to be greater than one, the following relationship must hold:

$$\frac{u}{N} + \frac{\lg u}{b} < 1 \tag{4.4}$$

[1] We are assuming a 32-bit address where an additional 32 bits are used to specify a mask. There are more effi cient ways to represent a prefi x, but this is tangential to our argument.

SA	*DA*	*Prot*
11*	001*	TCP
11*	001*	UDP
11*	101*	TCP
11*	101*	UDP
111*	001*	TCP
111*	001*	UDP
111*	101*	TCP
111*	101*	UDP

	SA
a	11*
b	111*

	DA
a	001*
b	101*

	Prot
a	TCP
b	UDP

filters
(*a*,*a*,*a*)
(*a*,*a*,*b*)
(*a*,*b*,*a*)
(*a*,*b*,*b*)
(*b*,*a*,*a*)
(*b*,*a*,*b*)
(*b*,*b*,*a*)
(*b*,*b*,*b*)

Figure 4.2: Example of encoding filters by unique field values to reduce storage requirements.

Note that $u \leq 2^b$ and $u \leq N$. Thus, the savings factor increases as the number of filters in the filter set and the size (number of bits) of filter fields increases relative to the number of unique filter fields. For our simple example in Figure 4.2, this encoding technique reduces the storage requirement from 1088 bits to 296 bits, or a factor of 3.7. As discussed in Section 5.8, we anticipate that future filter sets will include filters with more fields. It is also likely that the additional fields will contain a handful of unique values. As this occurs, the linear memory requirement of techniques explicitly storing the filter set will become increasingly sub-optimal.

4.2.1 Linear Search

Performing a linear search through a list of filters has $O(N)$ storage requirements, but it also requires $O(N)$ memory accesses per lookup. For even modest sized filter sets, linear search becomes prohibitively slow. It is possible to reduce the number of memory accesses per lookup by a small constant factor by partitioning the list into sub-lists and pipelining the search where each stage searches a sub-list. If p is the number of pipeline stages, then the number of memory accesses per lookup is reduced to $O(\frac{N}{p})$ but the computational resource requirement increases by a factor of p. While one could argue that a hardware device with many small embedded memory blocks could provide reasonable performance and capacity, latency increasingly becomes an issue with deeper pipelines and higher link rates. Linear search is a popular solution for the final stage of a lookup when the set of possible matching filters has been reduced to a bounded constant [51, 29, 59].

4.2.2 Ternary Content Addressable Memory (TCAM)

Taking a cue from fully-associative cache memories, Ternary Content Addressable Memory (TCAM) devices perform a parallel search over all filters in the filter set [57]. TCAMs were developed with the ability to store a “Don’t Care” state in addition to a binary digit. Input keys are compared against every TCAM entry, thereby enabling them to retain single clock cycle lookups for arbitrary bit mask

value	***a1***	***a2***
Don’t Care	0	0
1	0	1
0	1	0
undefined	1	1

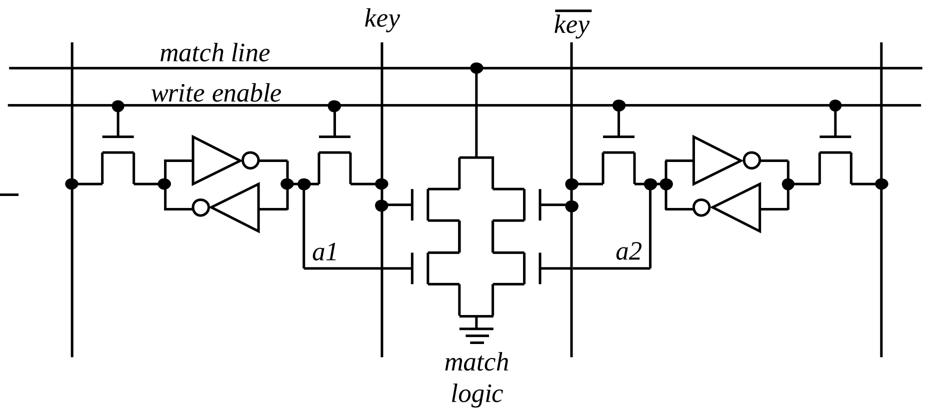


Figure 4.3: Circuit diagram of a standard TCAM cell; the stored value (0, 1, Don’t Care) is encoded using two registers *a1* and *a2*.

matches. TCAMs do suffer from four primary deficiencies: (1) high cost per bit relative to other memory technologies, (2) storage inefficiency, (3) high power consumption, (4) limited scalability to long input keys. With respect to cost, a current price check revealed that TCAM costs about 30 times more per bit of storage than DDR SRAM. While it is likely that TCAM prices will fall in the future, it is unlikely that they will be able to leverage the economy of scale enjoyed by SRAM and DRAM technology.

The storage inefficiency comes from two sources. First, arbitrary ranges must be converted into prefixes. In the worst case, a range covering w-bit port numbers may require $2(w-1)$ prefixes. Note that a single filter including two port ranges could require $2(w-1)^2$ entries, or 900 entries for 16-bit port numbers. As discussed in Section 5.3.3, we performed an analysis of 12 real filter sets and found that the *Expansion Factor*, or ratio of the number of required TCAM entries to the number of filters, ranged from 1.0 to 6.2 with an average of 2.32. This suggests that designers should budget at least seven TCAM entries per filter, compounding the hardware and power inefficiencies described below. The second source of storage inefficiency stems from the additional hardware required to implement the third “Don’t Care” state. 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 and a cell 2.7 times larger than a standard SRAM cell [57]. A circuit diagram of a standard TCAM cell is shown in Figure 4.3. Some proprietary architectures allow TCAM cells to require as few as 14 transistors [55] [56].

The massive parallelism inherent in TCAM architecture is the 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 result in access times approximately three times longer than SRAM [61]. Additionally, power consumption per bit of storage is on the order of 3 micro-Watts per “bit” [62] compared to 20 to 30 nano-Watts per bit for SRAM [63]. In summary, TCAMs consume 150 times more power per bit than SRAM.

Spitznagel, Taylor, and Turner recently introduced *Extended TCAM* (E-TCAM) which implements range matching directly in hardware and reduces power consumption by over 90% relative to standard TCAM [32]. We discuss E-TCAM in more detail in Section 4.3.6. While this represents promising new work in the architectural thread of research, it does not address the high cost per bit or scalability issues inherent in TCAMs for longer search keys. TCAM suffers from limited scalability to longer search keys due to its use of the exhaustive search approach. As previously discussed, the explicit storage of each filter becomes more inefficient as filter sizes increase and the number of unique field values remains limited. If the additional filter fields require range matches, this effect is compounded due to the previously described inefficiency of mapping arbitrary ranges to prefixes.

4.3 Decision Tree

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 make 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 by 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 filter fields to a single type of match. Several techniques convert all filter fields to bit vectors with arbitrary bit masks, i.e. bit vectors where each bit may be a 1, 0, or $*$ ("Don't Care"). Recall that filters containing arbitrary ranges do not readily map to arbitrary bit masks; therefore, this conversion process results in filter replication. Likewise, the use of wildcards may cause a filter to be stored at many leaves of the decision tree.

To better illustrate these issues, we provide an example of a naïve construction of a decision tree in Figure 4.4. The five filters in the example set contain three fields: 3-bit address prefix, an arbitrary range covering 3-bit port numbers, and an exact 2-bit value or wildcard. We first convert the five filters into bit vectors with arbitrary bit masks which increases the number of filters to eight. Viewing the construction process as progressing in a depth-first manner, a decision tree path is expanded until the node covers only one filter or the bit vector is exhausted. Nodes at the last level may cover more than one filter if filters overlap. We assume that leaf nodes contain the action to be applied to packets matching the filter or subset of filters covered by the node. Due to the size of the full decision tree, we show a portion of the data structure in Figure 4.4. If we evaluate this data structure by its ability to distinguish between potentially matching filters for a given packet, we see that this naïve construction is not highly effective. As the reader has most likely observed already, there are numerous optimizations that could allow a decision tree to more effectively distinguish

10*	[0:2]	01
0*	[3:7]	01
111	[0:7]	*
11*	[0:2]	10
*	[0:7]	10

a	10*	00*	01
b	10*	010	01
c	0**	011	01
d	0**	1**	01
e	111	***	**
f	11*	00*	10
g	11*	010	10
h	11*	010	10

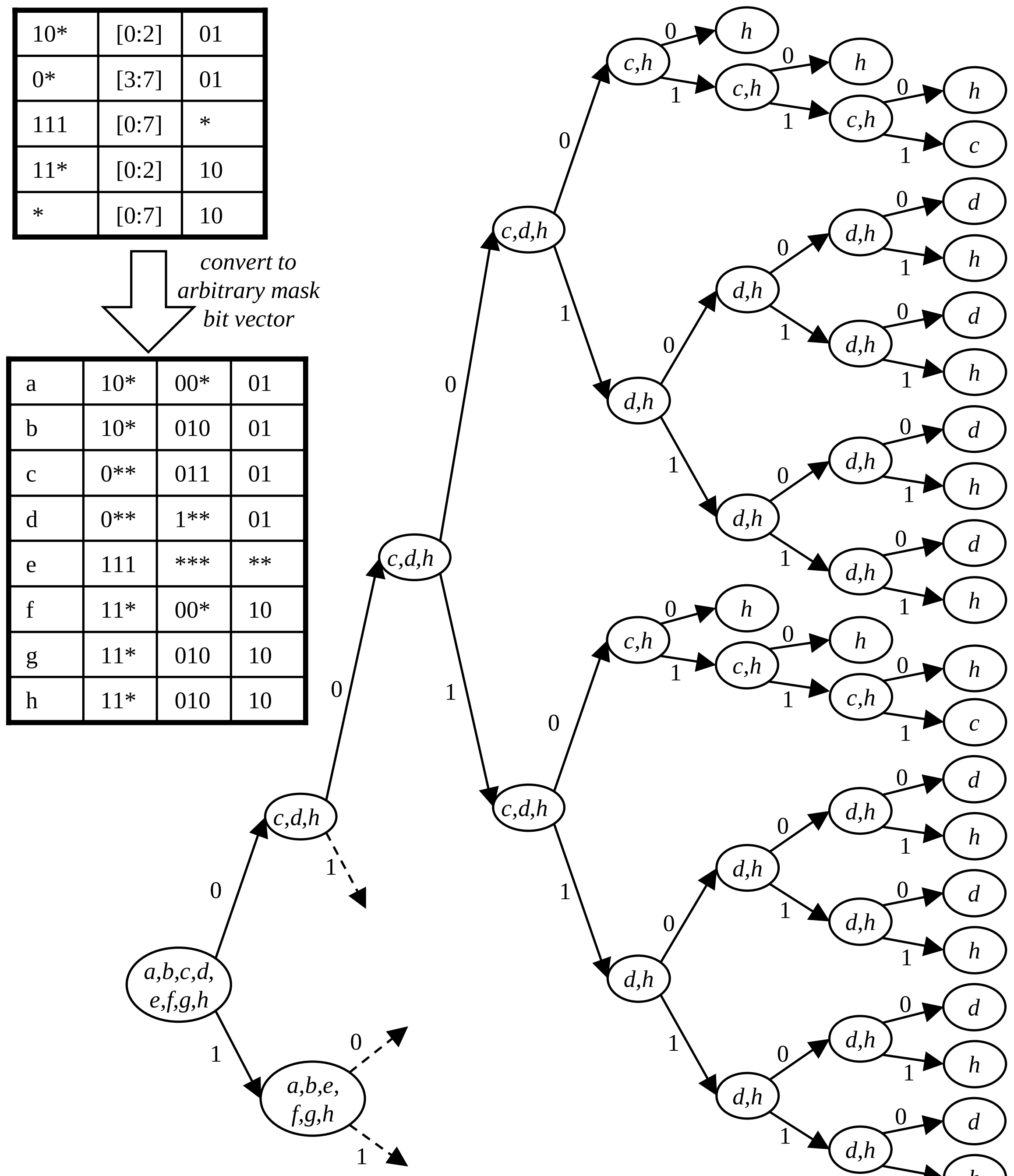


Figure 4.4: Example of a naïve construction of a decision tree for packet classification on three fields; all filter fields are converted to bit vectors with arbitrary bit masks.

between potentially matching filters. The algorithms and architectures discussed in the following subsections explore these optimizations.

Several of the algorithms that we classify as using a decision tree approach are more commonly referred to as "cutting" algorithms. These algorithms view filters with d fields as defining d-dimensional rectangles in d-dimensional space; thus, a "cut" in multi-dimensional space is isomorphic to a branch in a decision tree. The branching decision in a cutting algorithm is typically more complex than examining a single bit in a bit vector. Note that the E-TCAM approach discussed in Section 4.3.6 employs a variant on the cutting algorithms that may be viewed as a parallel search of several decision trees containing different parts of the filter set. Thus, we view some cutting algorithms as relaxing the constraints on classical decision trees.

Due to the many degrees of freedom in decision tree approaches, the performance characteristics and resource requirements vary significantly among algorithms. In general, lookup time is $O(W)$, where W is the number of bits used to specify the filter. Given that filters classifying on the standard 5-tuple require a minimum of 104 bits, viable approaches must employ some optimizations in order to meet throughput constraints. The memory requirement for our naïve construction is $O(2^{W+1})$. In general, memory requirements vary widely depending upon the complexity of the branching decisions employed by the data structure. One common feature of algorithms employing the decision tree approach is memory access dependency. Stated another way, the decision tree searches are inherently serial; a matching filter is found by traversing the tree from root to leaf. The serial nature of the decision tree approach precludes fully parallel implementations. If an algorithm places a bound on the depth of the decision tree, then implementing the algorithm in a pipelined architecture can yield high throughput. This does require an independent memory interfaces for each pipeline stage.

4.3.1 Grid-of-Tries

Srinivasan, Varghese, Suri, and Waldvogel introduced the seminal *Grid-of-Tries* and *Crossproducting* algorithms for packet classification [53]. In this section we focus on *Grid-of-Tries* which applies a decision tree approach to the problem of packet classification on source and destination address prefixes. *Crossproducting* was one of the first techniques to employ decomposition and we discuss it in Section 4.4.3. For filters defined by source and destination prefixes, *Grid-of-Tries* improves upon the directed acyclic graph (DAG) technique introduced by Decasper, Dittia, Parulkar, and Plattner [64]. This technique is also called set pruning trees because redundant subtrees can be "pruned" from the tree by allowing multiple incoming edges at a node. While this optimization does eliminate redundant subtrees, 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.

Figure 4.5 highlights the differences between set pruning trees and *Grid-of-Tries* using the example filter set shown in Table 4.1. Note that we have restricted the classification to two fields, destination address prefix followed by source address prefix. Assume we are searching for the best matching filter for a packet with destination and source addresses equal to 0011. In the *Grid-of-Tries*

Table 4.1: Example filter set; port numbers are restricted to be an exact value or wildcard.

Filter	*DA*	*SA*	*DP*	*SP*	*PR*
F_1	$0*$	$10*$	$*$	80	TCP
F_2	$0*$	$01*$	$*$	80	TCP
F_3	$0*$	$1*$	17	17	UDP
F_4	$00*$	$1*$	$*$	$*$	$*$
F_5	$00*$	$11*$	$*$	$*$	TCP
F_6	$10*$	$1*$	17	17	UDP
F_7	$*$	$00*$	$*$	$*$	$*$
F_8	$0*$	$10*$	$*$	100	TCP
F_9	$0*$	$1*$	17	44	UDP
F_{10}	$0*$	$10*$	80	$*$	TCP
F_{11}	$111*$	$000*$	$*$	44	UDP

structure, we find the longest matching destination address prefix $00*$ and follow the pointer to the source address tree. Since there is no 0 branch at the root node, we follow the *switch pointer* to the $0*$ node in the source address tree for destination address prefix $0*$. Since there is no branch for $00*$ in this tree, we follow the *switch pointer* to the $00*$ node in the source address tree for destination address prefix $*$. Here we find a stored filter F_7 which is the best matching filter for the packet.

Grid-of-Tries bounds memory usage to $O(NW)$ while achieving a 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 used multi-bit tries sampling 8 bits at a time for the destination trie; each of the source tries started 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. Memory requirements for 20k filters was around 2MB.

While *Grid-of-Tries* is an efficient technique for classifying on address prefix pairs, it does not directly extend to searches with additional filter fields. Consider searching the filter set in Table 4.1 using the following header fields: destination address 0000, source address 1101, destination port 17, source port 17, protocol UDP. Using the *Grid-of-Tries* structure in Figure 4.5, we find the longest matching prefix for the destination address, $00*$, followed by the longest matching prefix for the source address, $11*$. Filter F_5 is stored at this node and there are no *switch pointers* to continue the search. Since the remaining three fields of F_5 match the packet header, we declare F_5 is the best matching filter. Note that F_3, F_4, and F_9 also match. F_3 and F_9 also have more specific matches on the port number fields. Clearly, *Grid-of-Tries* does not directly extend to multiple field searches beyond address prefix matching.

The authors do propose a technique using multiple instances of the *Grid-of-Tries* structure for packet classification on the standard 5-tuple. The general approach is to partition the filter set into classes based on the tuple defined by the port number fields and protocol fields. An example

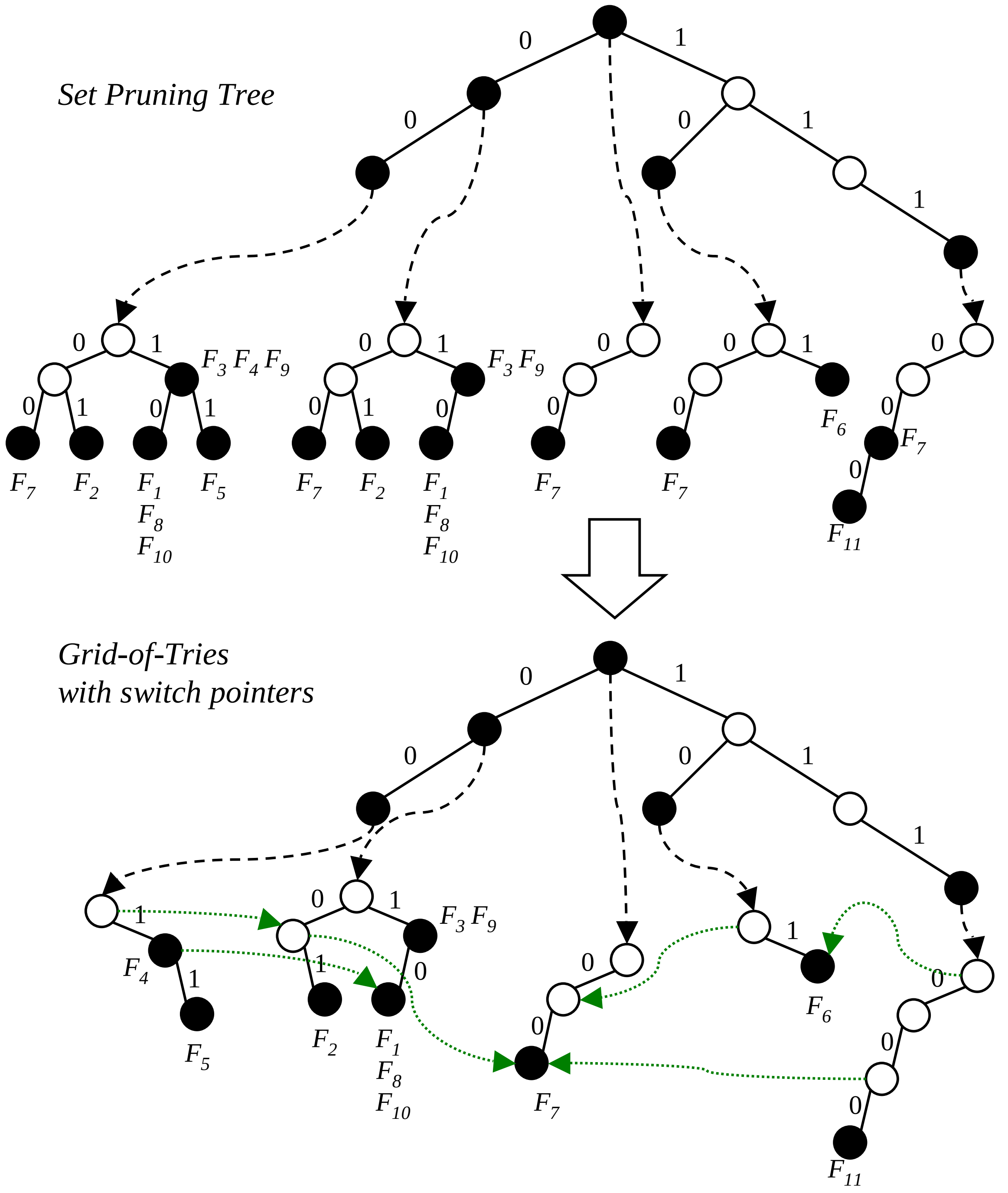


Figure 4.5: Example of set pruning trees and *Grid-of-Tries* classifying on the destination and source address prefixes for the example filter set in Table 4.1.

is shown in Figure 4.6. Operating under the restriction that port numbers must either be an exact

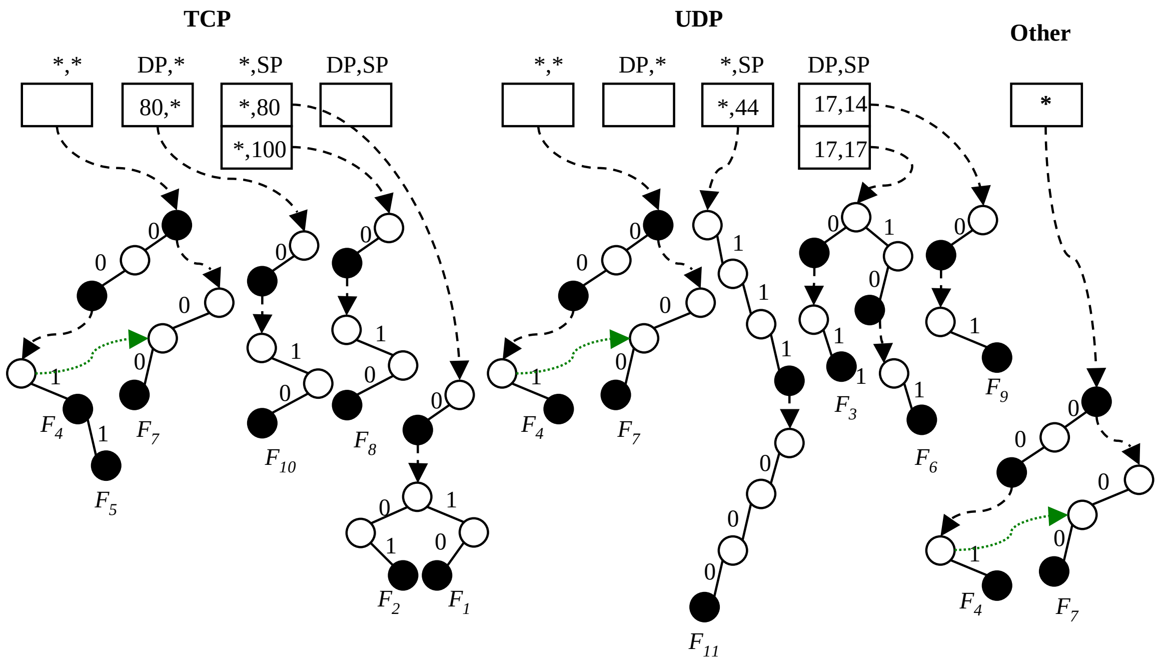


Figure 4.6: Example of 5-tuple packet classification using *Grid-of-Tries*, pre-filtering on protocol and port number classes, for the example filter set in Table 4.1.

port number or wildcard[2], we first partition the filter set into three classes according to protocol: TCP, UDP, and "other". Filters with a wildcard are replicated and placed into each class. We then partition the filters in the "other" class into sub-classes by protocol specification. For each "other" sub-class, we construct a *Grid-of-Tries*. The construction for the TCP and UDP classes are slightly different due to the use of port numbers. For both the UDP and TCP classes, we partition the constituent filters into four sub-classes according to the port number tuple: both ports specified; destination port specified, source port wildcard; destination port wildcard, source port specified; both ports wildcard. For each sub-class, we construct a hash table storing the unique combinations of port number specifications. Each entry contains a pointer to a *Grid-of-Tries* constructed from the constituent filters. Ignoring the draconian restriction on port number specifications, this approach may require $O(N)$ separate data-structures and filters with a wildcard protocol specification are replicated across many of them. It is generally agreed that the great value of the *Grid-of-Tries* technique lies in its ability to efficiently handle filters classifying on address prefixes.

4.3.2 Extended Grid-of-Tries (EGT)

Baboescu, Singh, and Varghese proposed *Extended Grid-of-Tries* (EGT) that supports multiple fields searches without the need for many instances of the data structure [58]. EGT essentially alters the *switch pointers* to be *jump pointers* that direct the search to all possible matching filters,

[2]Note that this restriction can be prohibitive for fi lter sets specifying arbitrary ranges. While fi lters could be replicated, typical ranges cover thousands of port numbers which induces an unmanageable expansion in the size of the fi lter set.

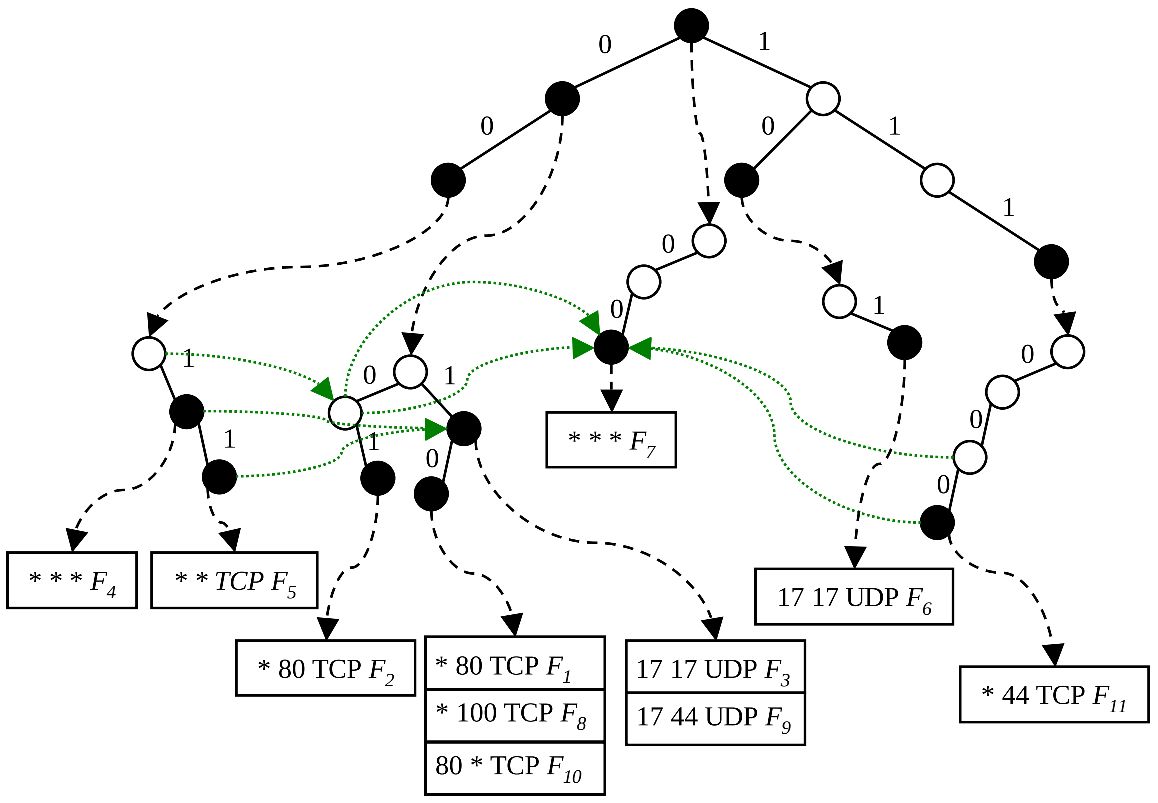


Figure 4.7: Example of 5-tuple packet classification using *Extended Grid-of-Tries* (EGT) for the example filter set in Table 4.1.

rather than the filters with the longest matching destination and source address prefixes. As shown in Figure 4.7, EGT begins by constructing a standard *Grid-of-Tries* using the destination and source address prefixes of all the filters in the filters set. Rather than storing matching filters at source address prefix nodes, EGT stores a pointer to a list of filters that specify the destination and source address prefixes, along with the remaining three fields of the filters. The authors observe that the size of these lists is small for typical *core router* filter sets[3], thus a linear search through the list of filters is a viable option. Note that the *jump pointers* between source tries direct the search to all possible matching filters. In the worst case, EGT requires $O(W^2)$ memory accesses where W is the address length. Simulated results with core router filter sets show that EGT requires 84 to 137 memory accesses per lookup for filter sets ranging in size from 85 to 2799 filters. Simulated results with synthetically generated filter sets resulted in 121 to 213 memory accesses for filter sets ranging in size from 5k to 10k filters. Memory requirements ranged from 33 bytes per filter to 57 bytes per filter.

[3]This property does not necessarily hold for filter sets in other application environments such as firewalls and edge routers.

Table 4.2: Example filter set; address field is 4-bits and port ranges cover 4-bit port numbers.

Filter	*Address*	*Port*
a	1010	2 : 2
b	1100	5 : 5
c	0101	8 : 8
d	$*$	6 : 6
e	111$*$	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 : 11

4.3.3 Hierarchical Intelligent Cuttings (HiCuts)

Gupta and McKeown introduced a seminal technique called *Hierarchical Intelligent Cuttings* (*HiCuts*) [51]. The concept of "cutting" comes from viewing the packet classification problem geometrically. Each filter in the filter set defines a d-dimensional rectangle in d-dimensional space, where d is the number of fields in the filter. Selecting a decision criteria is analogous to choosing a partitioning, or "cutting", of the space. Consider the example filter set in Table 4.2 consisting of filters with two fields: a 4-bit address prefix and a port range covering 4-bit port numbers. This filter set is shown geometrically in Figure 4.8.

HiCuts preprocesses 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.

A *HiCuts* data structure for the example filter set in Table 4.2 is shown in Figure 4.9. 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 4.9 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. A geometric representation of the partitions created by the search tree are shown in Figure 4.10. 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

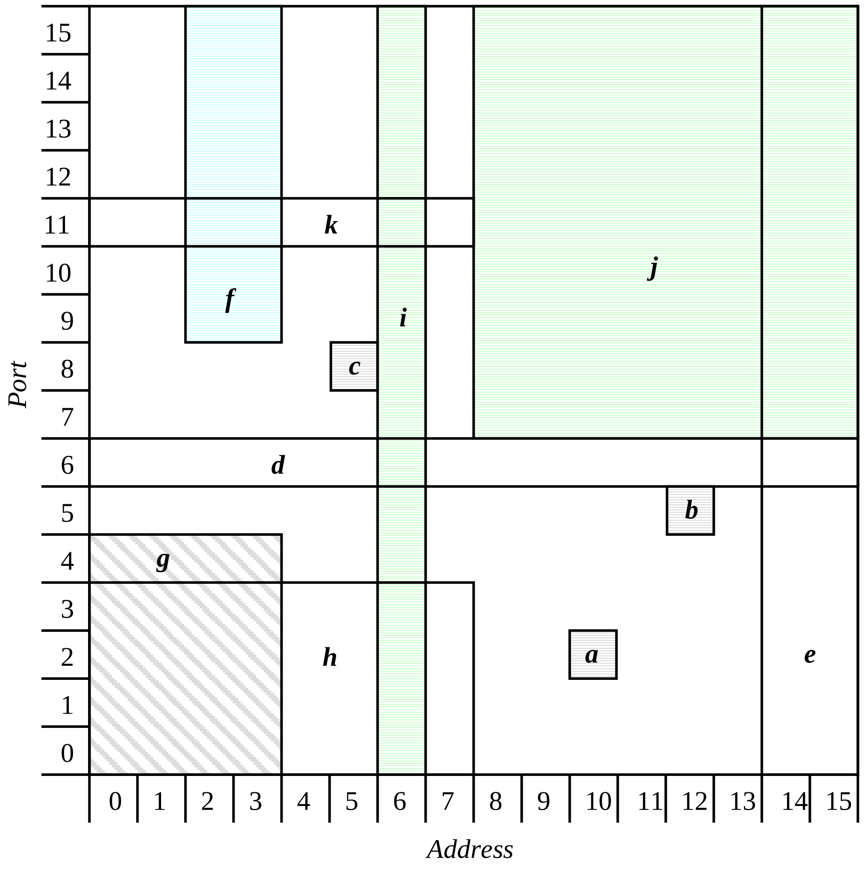


Figure 4.8: Geometric representation of the example filter set shown in Table 4.2.

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 preprocessing required, this scheme does not readily support incremental updates. Measured update times ranged from 1ms to 70ms.

4.3.4 Modular Packet Classification

Woo independently applied the same approach as *HiCuts* and introduced a flexible framework for packet classification based on a multi-stage search over ternary strings representing the filters [29]. The framework contains three stages: an index jump table, search trees, and filter buckets. An example data structure for the filter set in Table 4.2 is shown in Figure 4.11. A search begins by using selected bits of the input packet fields to address the index jump table. If the entry contains a valid pointer to a search tree, the search continues starting at the root of the search tree. Entries without a search tree pointer store the action to apply to matching packets. Each search tree node specifies the bits of the input packet fields to use in order to make a branching decision. When a filter bucket is reached, the matching filter is selected from the set in the bucket via linear search,

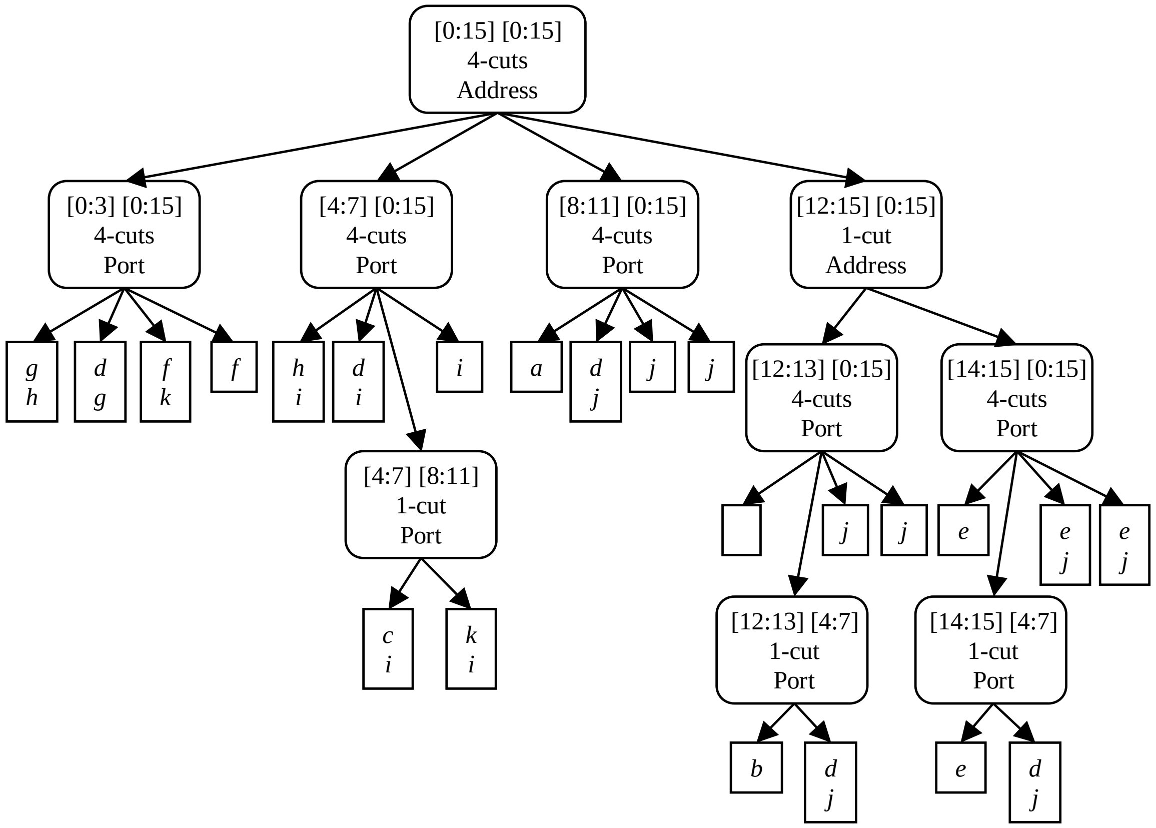


Figure 4.9: Example *HiCuts* data structure for example filter set in Table 4.2.

binary search, or CAM. A key assumption is that every filter can be expressed as a ternary string of 1's, 0's, and $*$'s which represent "don't care" bits. A filter containing prefix matches on each field is easily expressed as a ternary string by concatenating the fields of the filter; however, a filter containing arbitrary ranges may require replication. Recall that standard 5-tuple filters may contain arbitrary ranges for each of the two 16-bit transport port numbers; hence, a single filter may yield 900 filter strings in the worst case.

The first step in constructing the data structures is to convert the filters in the filter into ternary strings and organize them in an $n \times m$ array where the number of rows n is equal to the number of ternary strings and the number of columns m is equal to the number of bits in each string. Each string has an associated weight W_i which is proportional to its frequency of use relative to the other strings; more frequently matching filter strings will have a larger weight. Next, the bits used to address the index jump table are selected. For our example in Figure 4.11, we create a 3-bit index concatenate from bits 7, 3, and 2 of the ternary search strings. Typically, the bits used for the jump table address are selected such that every filter specifies those bits. When filters contain "don't cares" in jump table address bits, it must be stored in all search trees associated with the addresses covered by the jump index. For each entry in the index jump table that is addressed by at least one filter, a search tree is constructed. In the general framework, the search trees may examine

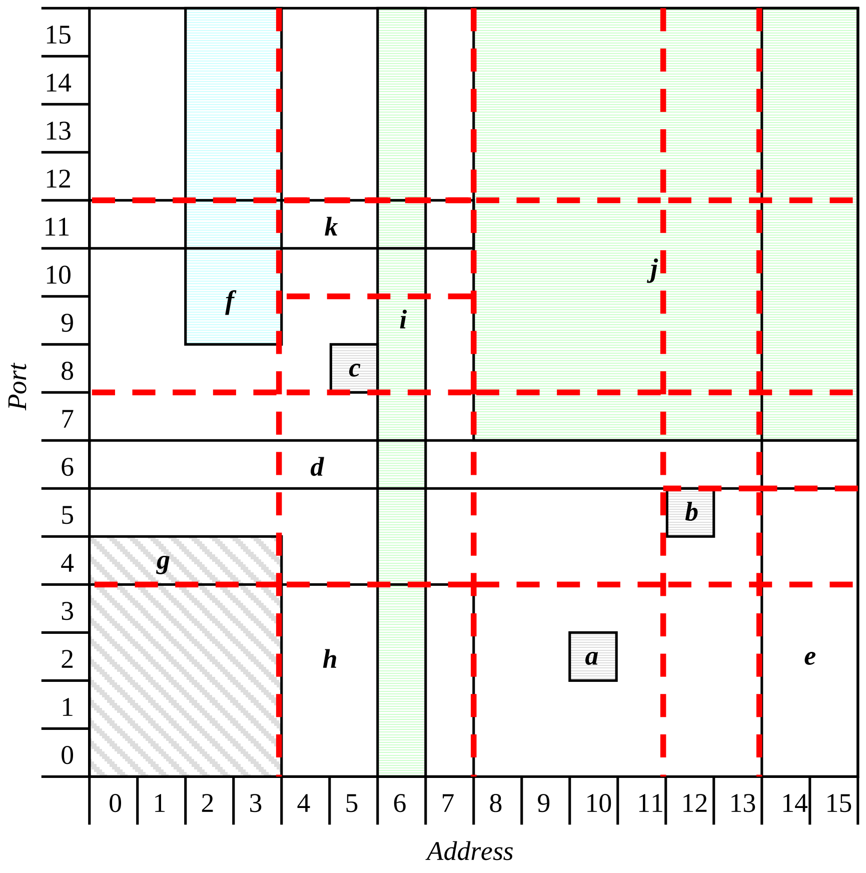


Figure 4.10: Geometric representation of partitioning created by *HiCuts* data structure shown in Figure 4.9.

any number of bits at each node in order to make a branching decision. Selection of bits is made based on a weighted average of the search path length where weights are derived from the filter weights W_i. This attempts to balance the search tree while placing more frequently accessed filter buckets nearer to the root of the search tree. Note that our example in Figure 4.11 does not reflect this weighting scheme. Search tree construction is performed recursively until the number of filters at each node falls below a threshold for filter bucket size, usually 128 filters or less. We set the threshold to two filters in our example. The construction algorithm is "greedy" in that it performs local optimizations.

Simulation results with synthetically generated filter sets show that memory scales linearly with the number of filters. For 512k filters and a filter bucket size of 16, the depth of the search tree ranged from 11 levels to 35 levels and the number of filter buckets ranged from 76k to 350k depending on the size of the index jump table. Note that larger index jump tables decrease tree depth at the cost of increasing the number of filter buckets due to filter replication.

Filter	B(7:0)
a	1010 0010
b	1100 0101
c	0101 1000
d	**** 0110
e	111* ****
*f*1	001* 1001
*f*2	001* 101*
*f*3	001* 11**
*g*1	00** 00**
*g*2	00** 0100
h	0*** 00**
i	0110 ****
*j*1	1*** 0111
*j*2	1*** 1***
k	0*** 1011

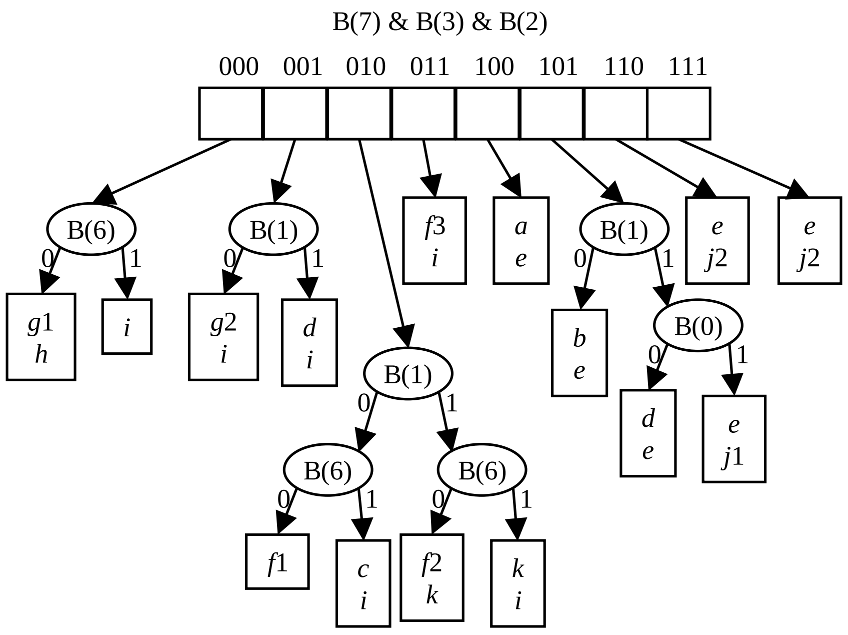


Figure 4.11: Modular packet classification using ternary strings and a three-stage search architecture.

4.3.5 HyperCuts

Introduced by Singh, Baboescu, Varghese, and Wang, the *HyperCuts* algorithm [59] improves upon the *HiCuts* algorithm developed by Gupta and McKeown [51] and also shares similarities with the *Modular Packet Classification* algorithms introduced by Woo [29]. In essence, *HyperCuts* is a decision tree algorithm that attempts to minimize the depth of the tree by selecting *multiple* "cuts" in multi-dimensional space that partition the filter set into lists of bounded size. By forcing cuts to create uniform regions, *HyperCuts* efficiently encodes pointers using indexing, which allows the data structure to make multiple cuts in multiple dimensions without a significant memory penalty.

According to reported simulation results, traversing the *HyperCuts* decision tree required between 8 and 32 memory accesses for real filter sets ranging in size from 85 to 4740 filters, respectively. Memory requirements for the decision tree ranged from 5.4 bytes per filter to 145.9 bytes per filter. For synthetic filter sets ranging in size from 5000 to 20000 filters, traversing the *HyperCuts* decision tree required between 8 and 35 memory accesses, while memory requirements for the decision tree ranged from 11.8 to 30.1 bytes per filter. The number of filters and encoding of filters in the final lists are not provided; hence, it is difficult to assess the additional time and space requirements for searching the lists at the leaves of the decision tree. *HyperCuts*'s support for incremental updates are not specifically addressed. While it is conceivable that the data structure can easily support a moderate rate of randomized updates, it appears that an adversarial worst-case stream of updates can either create an arbitrarily deep decision tree or force a significant restructuring of the tree.

4.3.6 Extended TCAM (E-TCAM)

Spitznagel, Taylor, and Turner recently introduced *Extended TCAM* (E-TCAM) to address two of the primary inefficiencies of Ternary Content-Addressable Memory (TCAM): power consumption and storage inefficiency. Recall that in standard TCAM, a single filter including two port ranges requires up to $2(w-1)^2$ entries where w is the number of bits required to specify a point in the range. Thus, a single filter with two fields specifying ranges on 16-bit port numbers requires 900 entries in the worst case. The authors found that storage efficiency of TCAMs for real filter sets ranges from 16% to 53%; thus, the average filter occupies between 1.8 and 6.2 TCAM entries. By implementing range matching directly in hardware, E-TCAM avoids this storage inefficiency at the cost of a small increase in hardware resources. When implemented in standard CMOS technology, a range matching circuit requires $44w$ transistors. This is considerably more than the $16w$ transistors required for prefix matching; however, the total hardware resources saved by eliminating the expansion factor for typical packet filter sets far outweighs the additional cost per bit for hardware range matching. Storing a filter for the standard IPv4 5-tuple requires approximately 18% more transistors per entry. This is a small increase relative to the 180% to 620% incurred by filter replication.

Given a query word, TCAMs compare the query word against every entry word in the device. This massively parallel operation results in high power consumption. E-TCAM reduces power consumption by limiting the number of active regions of the device during a search. The second architectural extension of E-TCAM is to partition the device into blocks that may be independently activated during a query. Realistic implementations would partition the device into blocks capable of storing hundreds of filters. In order to group filters into blocks, E-TCAM uses a multi-phase partitioning algorithm similar to the previously discussed "cutting" algorithms. The key differences in E-TCAM are that the depth of the "decision tree" used for the search is strictly limited by the hardware architecture and a query may search several "branches" of the decision tree in parallel. Figure 4.12 shows an example of an E-TCAM architecture and search using the example filter set in Table 4.2.

In this simple example, filter blocks may store up to four filters and the "decision tree" depth is limited to two levels. The first stage of the search queries the *index block* which contains one entry for each group created by the partitioning algorithm. For each phase of the partitioning algorithm except the last phase, a group is defined which completely contains at most b filters where b is the block size. Filters "overlapping" the group boundaries are not included in the group. The final phase of the algorithm includes such "overlapping" filters in the group. The number of phases determines the number of *index* entries that may match a query, and hence the number of filter blocks that need

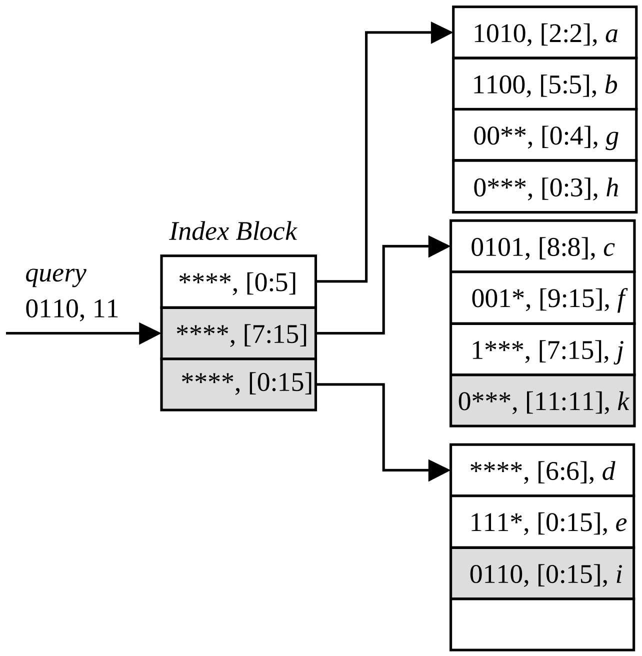


Figure 4.12: Example of searching the filter set in Table 4.2 using an *Extended TCAM* (E-TCAM) using a two-stage search and a filter block size of four.

to be searched. A geometric representation of the groupings created for our example is shown in Figure 4.13. Returning to our example in Figure 4.12, the matching entries in the *index block* activate the associated filter blocks for the next stage of the search. In this case, two filter blocks are active. Note that all active filter blocks are searched in parallel; thus, with a pipelined implementation E-TCAM can retain single-cycle lookups. Simulations show that E-TCAM requires less than five percent of the power required by regular TCAM. Also note that multi-stage *index blocks* can be used to further reduce power consumption and provide finer partitioning of the filter set.

4.3.7 Fat Inverted Segment (FIS) Trees

Feldman and Muthukrishnan introduced another framework for packet classification using independent field searches on *Fat Inverted Segment* (*FIS*) *Trees* [27]. Like the previously discussed "cutting" algorithms, *FIS Trees* utilize a geometric view of the filter set and map filters into d-dimensional space. As shown in Figure 4.14, projections from the "edges" of the d-dimensional rectangles specified by the filters define elementary intervals on the axes; in this case, we form elementary intervals on the *Address* axis. Note that we are using the example filter set shown in Table 4.2 where filters contain two fields: a 4-bit address prefix and a range covering 4-bit port numbers. N filters will define a maximum of $I = (2N + 1)$ elementary intervals on each axis. An *FIS Tree* is a balanced t-ary tree with l levels that stores a set of segments, or ranges. Note that $t = (2I + 1)^{1/l}$ is the maximum number of children a node may have. The leaf nodes of the tree correspond to the elementary

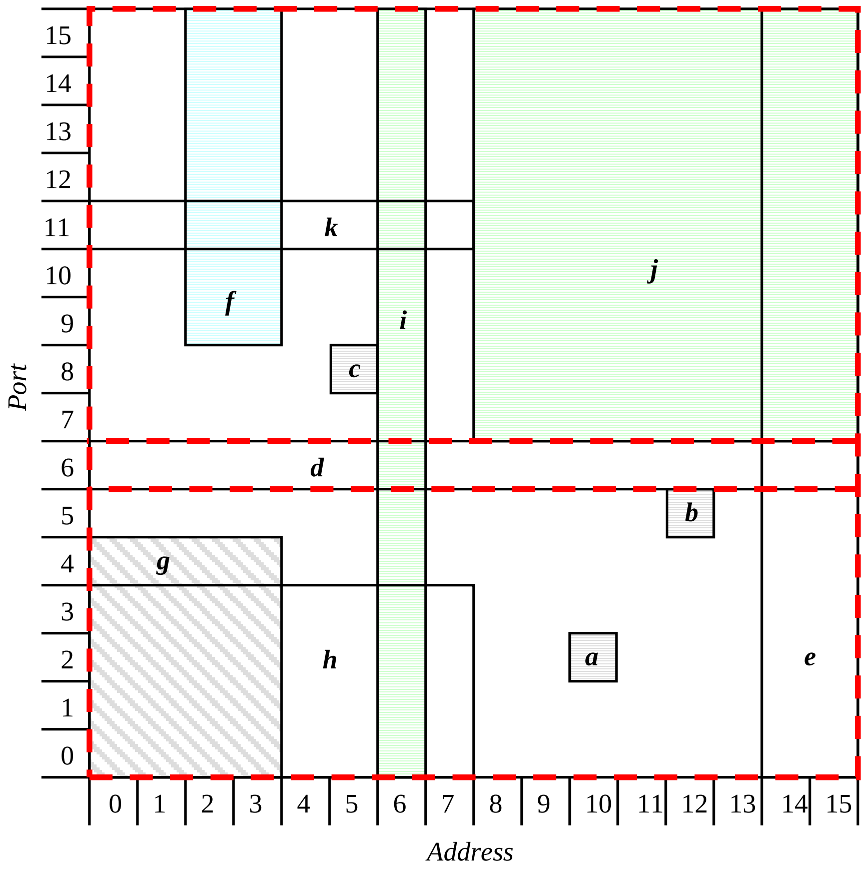


Figure 4.13: Example of partitioning the filter set in Table 4.2 for an *Extended TCAM* (E-TCAM) with a two-stage search and a filter block size of four.

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.

As shown in Figure 4.14, the framework 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. Note that an *FIS Tree* is not necessary for the last packet field. In this case, we only need to store the left-endpoints of the elementary intervals and the highest priority filter covering the elementary interval. The authors propose a method to use a *Longest Prefix Matching* technique to locate the elementary interval covering a given point. This method requires at most $2I$ prefixes.

Figure 4.14 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

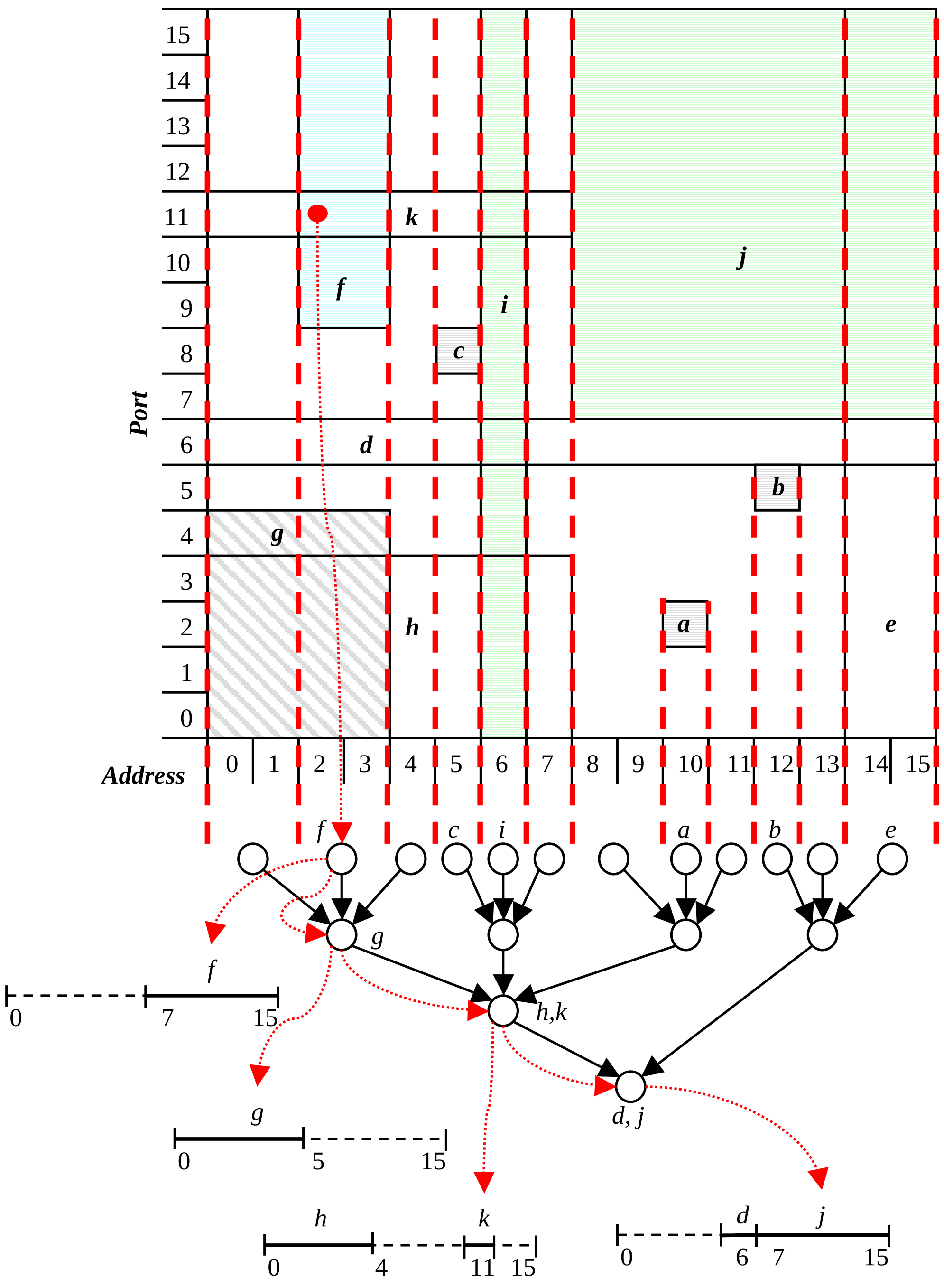


Figure 4.14: Example of *Fat Inverted Segment* (*FIS*) *Trees* for the filter set in Table 4.2.

remember the highest priority matching filter. Note that the basic framework requires a significant amount of precomputation due to its use of elementary intervals. This property does not readily support dynamic updates at high rates. The authors propose several data structure augmentations to allow dynamic updates. We do not discuss these sophisticated augmentations but do point out that they incur a performance penalty. The authors performed simulations with real and synthetic 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.

4.4 Decomposition

Given the wealth of efficient single field search techniques, decomposing a multiple field search problem into several instances of a single field search problem is a viable approach. Employing this high-level approach has several advantages. First, each single field search engine operates independently, thus we have the opportunity to leverage 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 fields. While these are compelling advantages, decomposing a multi-field search problem raises subtle issues.

The primary challenge in taking this high-level approach lies in efficiently aggregating the results of the single field searches. Many of the techniques discussed in this section use an encoding of the filters to facilitate result aggregation. Due to the freedom in choosing single field search techniques and filter encodings, the resource requirements and achievable performance vary drastically among the constituent techniques – even more so than with decision tree techniques. Limiting and managing the number of intermediate results returned by single field search engines is also a crucial design issue for decomposition techniques. Single field search engines often must return more than one result because packets may match more than one filter. As was highlighted by the previous discussion of using *Grid-of-Tries* for filters with additional port and protocol fields, it is not sufficient for single field search engines to simply return the longest matching prefix for a given filter field. 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 higher priority in other fields. As a result, techniques employing decomposition tend to leverage filter set characteristics that allow them to limit the number of intermediate results. In general, solutions using decomposition tend to provide the high throughput due to their amenability to parallel hardware implementations. The high level of lookup performance often comes at the cost of memory inefficiency and, hence, capacity constraints.

4.4.1 Parallel Bit-Vectors (BV)

Lakshman and Stiliadis introduced one of the first multiple field packet classification algorithms targeted to a hardware implementation. Their seminal technique is commonly referred to as the Lucent bit-vector scheme or *Parallel Bit-Vectors* (*BV*) [52]. The authors make the initial assumption that the filters may be sorted according to priority. Like the previously discussed "cutting" algorithms, *Parallel BV* utilizes a geometric view of the filter set and maps filters into d-dimensional space. As

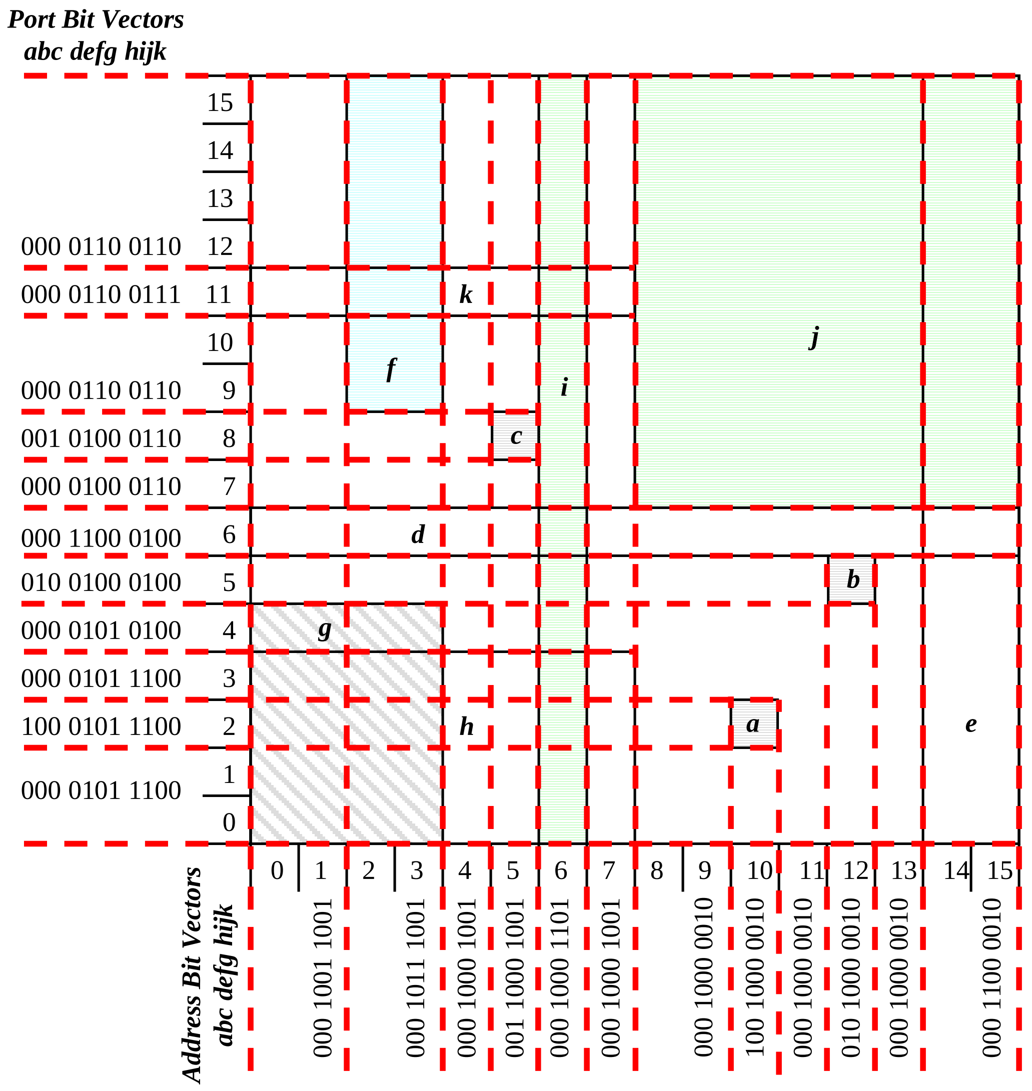


Figure 4.15: Example of bit-vector construction for the *Parallel Bit-Vectors* technique using the filter set shown in Table 4.2.

shown in Figure 4.15, 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 4.2 where filters contain two fields: a 4-bit address prefix and a range covering 4-bit port numbers. N filters will define a maximum of $(2N + 1)$ elementary intervals on each axis.

For each elementary interval on each axis, we define an N-bit bit-vector. 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 4.15. 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 returns 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 in an FPGA operating at 33MHz 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 $O(\lg N)$ search time and a rather unfavorable $O(N^2)$ memory requirement. The authors propose an algorithm to reduce the memory requirement to $O(N \log N)$ using incremental reads. The main idea behind this approach is to store a single bit vector for each dimension and a set of N pointers of size $\log N$ that record the bits that change between elementary intervals. This technique increases the number of memory accesses by $O(N \log N)$. The authors also propose a technique optimized for classification on source and destination address prefixes only, which we do not discuss here.

4.4.2 Aggregated Bit-Vector (ABV)

Baboescu and Varghese introduced the *Aggregated Bit-Vector* (ABV) algorithm which seeks to improve the performance of the *Parallel BV* technique by leveraging statistical observations of real filter sets [54]. *ABV* converts all filter fields to prefixes, hence it incurs the same replication penalty as TCAMs which we described in Section 4.2.2. 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 $\lceil \frac{N}{A} \rceil$ bits in size. Each chunk 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 4.16 provides an example using the filter set in Table 4.2.

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

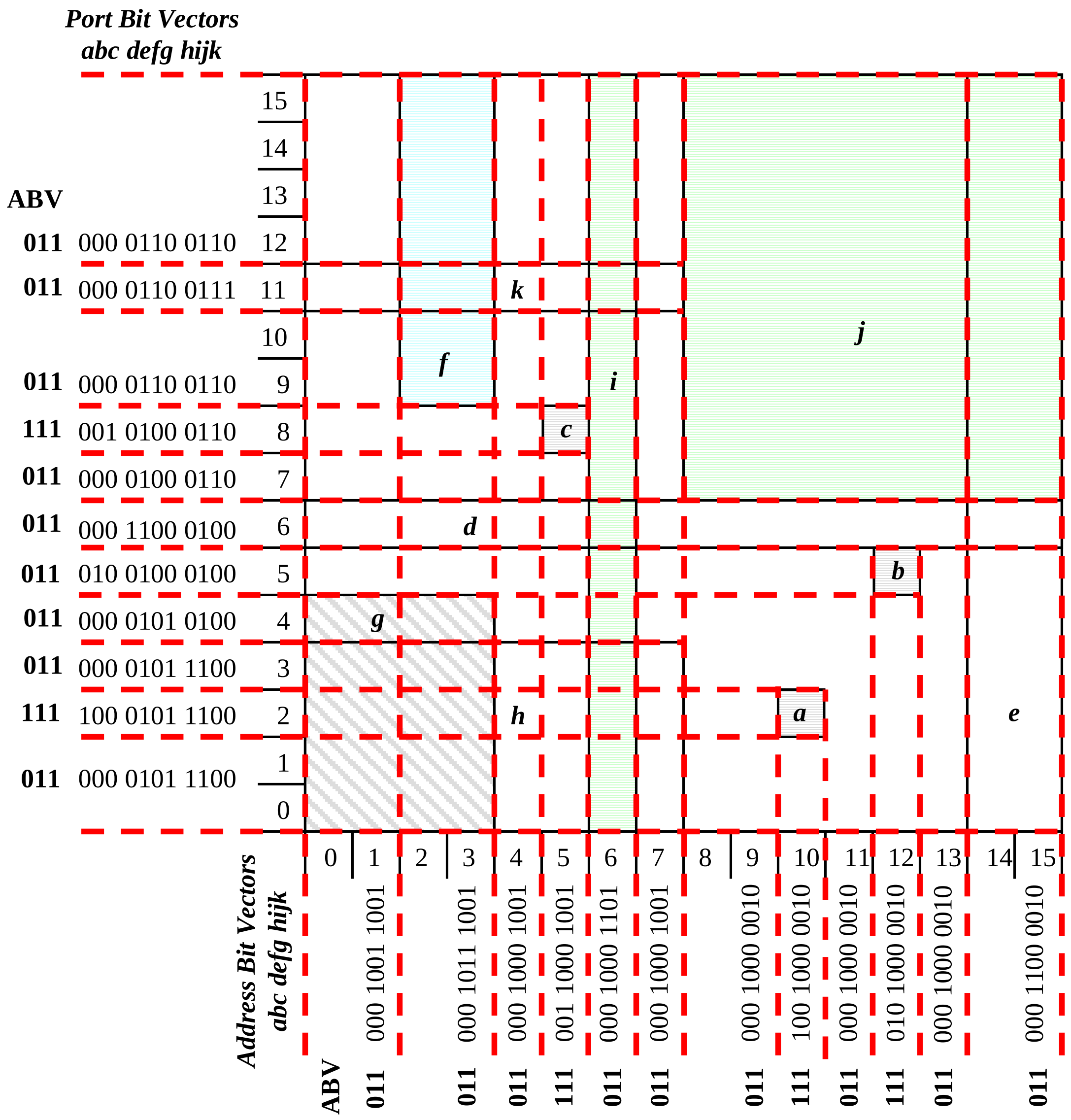


Figure 4.16: Example of bit-vector and aggregate bit-vector construction for the *Aggregated Bit-Vectors* technique using the filter set shown in Table 4.2.

reorder the filters 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 of a factor of 20 or more when the filter sets do not contain any wildcards. As wildcards increase, the reductions become much more modest.

4.4.3 Crossproducting

In addition to the previously described *Grid-of-Tries* algorithm, Srinivasan, Varghese, Suri, and Waldvogel also introduced the seminal *Crossproducting* technique [53]. Motivated by the observation that the number of unique field specifications is significantly less than the number of filters in the filter set, *Crossproducting* utilizes independent field searches then combines the results in a single step. For example, a filter set containing 100 filters may contain only 22 unique source address prefixes, 17 unique destination address prefixes, 11 unique source port ranges, etc. *Crossproducting* begins by constructing d sets of unique field specifications. For example, all of the destination address prefixes from all the filters in the filter set comprise a set, all the source address prefixes comprise a set, etc. Next, we construct independent data structures for each set that return a single best matching entry for a given packet field. In order to resolve the best matching filter for the given packet from the set of best matching entries for each field, we construct a table of crossproducts. In essence, we precompute the best matching filter for every possible combination of results from the d field searches. We locate the best matching filter for a given packet by using the concatenation of results from the independent lookups as a hash probe into the crossproduct table; thus, 5-tuple classification only requires five independent field searches and a single probe to a table of crossproducts. We provide a simple example for a filter set with three fields in Figure 4.17. Note that the full crossproduct table is not shown due to space constraints.

Given a parallel implementation, *Crossproducting* can provide high throughput, however it suffers from exponential memory requirements. For a set of N filters containing d fields each, the size of the crossproduct table can grow to $O(N^d)$. To keep a bound on the table size, the authors propose *On-demand Crossproducting* which places a limit on the size of the crossproduct table and treats it like a cache. If the field lookups produce a result without an entry in the crossproduct table of limited size, then we compute the crossproduct from the filter set and store it in the table[4]. The performance of this scheme largely depends upon locality of reference.

Finally the authors propose a combined scheme that seeks to leverage the strengths of both *Grid-of-Tries* and *Crossproducting*. The scheme utilizes *Grid-of-Tries* to perform the destination then source prefix matches and *Crossproducting* for ports and flags. The search terminates as soon as a matching filter is found. This assumes that the most specific filters are the highest priority and that a non-overlapping filter set can be constructed. Using optimistic assumptions regarding caching, the authors claim that a full filter match requires a worst case of 12 memory accesses.

4.4.4 Recursive Flow Classification (RFC)

Leveraging many of the same observations, Gupta and McKeown introduced *Recursive Flow Classification* (*RFC*) which provides high lookup rates at the cost of memory inefficiency [50]. The authors introduced a unique high-level view of the packet classification problem. Essentially, packet

[4]Cache entry replacement algorithms can be used to decide which entry to overwrite in the on-demand crossproduct table.

Filter Set

Filter	*Address*	*Port*	*Protocol*
a	000*	[0:1]	TCP
b	001*	[0:1]	TCP
c	1101	[1:1]	UDP
d	10*	[5:15]	UDP
e	001*	[5:15]	UDP
f	111*	[0:15]	UDP
g	000*	[5:15]	UDP
h	10*	[0:1]	TCP
i	001*	[1:1]	TCP
j	*	[0:15]	UDP
k	*	[0:15]	*

Field Sets

Address
000*
001*
1101
10*
111*
*

Port
[0:1]
[1:1]
[5:15]
[0:15]

Protocol
TCP
UDP
*

Table of Crossproducts

Address	*Port*	*Protocol*	*Best Match*
000*	[0:1]	TCP	*a*
000*	[0:1]	UDP	*j*
000*	[0:1]	*	*k*
000*	[1:1]	TCP	*a*
000*	[1:1]	UDP	*j*
000*	[1:1]	*	*k*
…	…	…	…
111*	[5:15]	TCP	*k*
111*	[5:15]	UDP	*f*
111*	[5:15]	*	*k*
…	…	…	…
*	[0:15]	*	*k*

Figure 4.17: Example of *Crossproducting* technique for filter set with three fields; full crossproduct table is not shown due to space constraints.

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 $O(n^d)$. 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}.

Similar to the *Crossproducting* technique, *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, rather than a single step as in *Crossproducting*. The result of each “chunk” lookup and aggregation step in *RFC* is an equivalence class identifier, *eqID*, that represents the set of potentially matching filters for the packet. The number of *eqIDs* in *RFC* depends upon the number of distinct sets of filters that can be matched by a packet. The number of *eqIDs* in an aggregation step scales with the number of unique overlapping regions formed by filter projections. An example of assigning *eqIDs* is shown in Figure 4.18. In this example, the rectangles $a, \ldots, k$ are defined by the two fields of the filters in our running example filter set in Table 4.2. In general, these could be rectangles defined by the projections of two “chunks” of the filters in the filter set. Note that the fields create nine equivalence classes in the *port* field and eight equivalence classes in the *address* field requiring 4-bit and 3-bit *eqIDs*, respectively.

RFC lookups in “chunk” and aggregation tables utilize indexing; the address for the table lookup is formed by concatenating the *eqIDs* from the previous stages. The resulting *eqID* is smaller (fewer number of bits) than the address; thus, *RFC* performs a multi-stage *reduction* to a final *eqID* 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. 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 discuss 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 precomputation in order to assign the proper *eqID* for the combination of the *eqID*s of the previous phases. Such extensive precomputation precludes dynamic updates at high rates.

4.4.5 Parallel Packet Classification (P^2C)

The *Parallel Packet Classification* (P^2C) scheme introduced by van Lunteren and Engbersen also falls into the class of techniques using decomposition [28]. The key novelties of P^2C are its encoding and aggregation of intermediate results. Similar to the *Parallel Bit-Vector* and *RFC* techniques, P^2C performs parallel searches in order to identify the elementary interval covering each packet field. The authors introduce three techniques for encoding the elementary intervals formed by the projections of filter fields. These techniques explore the design tradeoffs between update speed, space efficiency, and lookup complexity. For each field of each filter, P^2C computes the common

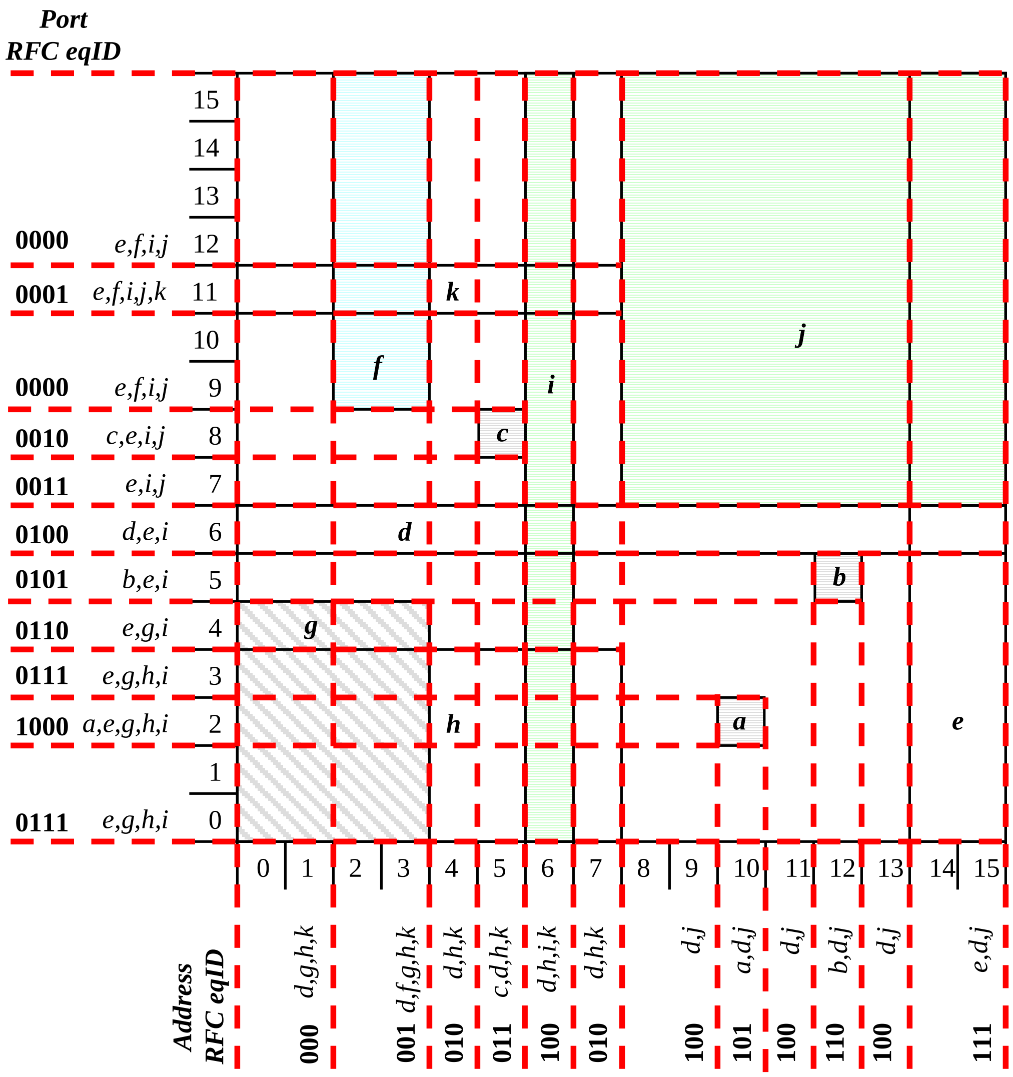
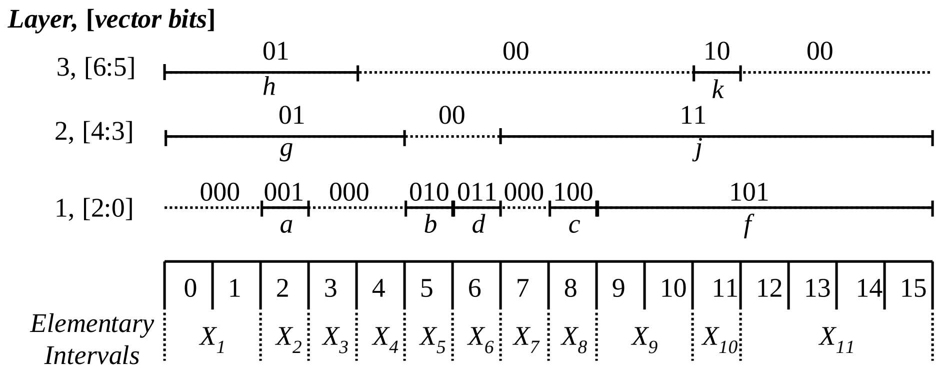


Figure 4.18: Example of *Recursive Flow Classification* (*RFC*) using the filter set in Table 4.2.

bits of all the encodings for the elementary intervals covered by the given filter field. This computation produces a ternary search string for each filter field. The ternary strings for each field are concatenated and stored in a TCAM according to the filter priority.

Figure 4.19 shows an example of the first, and most update efficient, P^2C encoding technique for the *port* fields of the filters in Table 4.2. In this encoding technique, we organize the ranges defined by the filters in the filter set into a multi-layer hierarchy such that the ranges at each layer are non-overlapping and the number of layers is minimized. Note that the number of layers is equal to the maximum number of overlapping ranges for any port number. At each layer, we assign a unique label to the ranges using the minimum number of bits. Within each layer, regions not covered by a range may share the same label. Next, we compute an intermediate bit-vector for each elementary interval $X_1, \ldots, X_{11}$ defined by the filter fields. We form an intermediate bit-vector



Elementary Interval	*Intermediate Vector* [6:0]
X_1	0101000
X_2	0101001
X_3	0101000
X_4	0001000
X_5	0000010
X_6	0000011
X_7	0011000
X_8	0011100
X_9	0011101
X_{10}	1011101
X_{11}	0011101

Filter	*Ternary Match Condition*
a	0101001
b	0000010
c	0000100
d	0000011
e	*******
f	0*11101
g	0*0100*
h	010100*
i	*******
j	0*11*0*
k	1011101

Figure 4.19: Example of *Parallel Packet Classification* (P^2C) using the most update-efficient encoding style for the port ranges defined in the filter set in Table 4.2.

by concatenating the labels for the covering ranges in each layer. Consider elementary interval X_2 which is covered by range $h(01)$ in layer 3, $g(01)$ in layer 2, and $a(001)$ in layer 1; its intermediate bit vector is 0101001. Finally, we compute the ternary match condition for each filter by computing the common bits of the intermediate bit-vectors for the set of elementary intervals covered by each filter. For each bit position in the intermediate bit-vectors, if all elementary intervals share the same bit value, then we assign that bit value to the corresponding bit position of the ternary match string; otherwise, we assign a "don't care", $*$, to the bit position in the ternary match string. Consider filter g which covers elementary intervals X_1, X_2, X_3, and X_4. For all bit-vectors, the most significant bit is '0' but they differ in the next bit position; thus, the ternary match string for g begins with $0*$.

Once we construct the table of ternary match strings for each filter field, we concatenate the field strings associated with each filter and store it in a TCAM. Strings are stored in order of filter priority. We also construct a data structure for each filter field which returns the intermediate bit-vector for the elementary interval covering the given packet field. A search locates the best-matching filter for a packet by searching these data structures in parallel, concatenating the intermediate bit-vectors to form a search key, and querying the TCAM with the search key. For the single field searches, the authors employ the *BARTs* technique which restricts independent field searches to be either prefix or exact match [65]. Arbitrary ranges must be converted to prefixes, increasing the number of unique field specifications. The primary deficiency of P^2C is its use of elementary intervals, as a single filter update may add or remove several elementary intervals for each field. When using the most space efficient encoding techniques, it is possible for one filter update to require updates to every primitive range encoding. Using the most update efficient encoding, the number and size of intermediate results grows super-linearly with the number of filters. For a sample filter set of 1733 filters, P^2C required 2k bytes of SRAM and 5.1k bytes of TCAM. The same filter set requires 24k bytes using a standard TCAM exclusively, thus P^2C reduced TCAM storage requirements by a factor of 4.7 and required only 1.2 bytes of SRAM per filter.

4.4.6 Distributed Crossproducting of Field Labels (DCFL)

Distributed Crossproducting of Field Labels (DCFL) leverages filter set characteristics, decomposition, and a novel labeling technique to construct a packet classification technique targeted to high-performance hardware implementation. We provide a complete description of DCFL in Chapter 7, but include a brief introduction to the algorithm here in order to place it in context with the body of related work. Two observations motivated the development of DCFL: the structure of real filter sets and advancements in integrated circuit technology. As we discuss in Chapter 5, we found that the number of unique filter field values matching a given packet are inherently limited in real filter sets. Likewise, the number of combinations of unique filter field values matching a given packet are also limited. As we discuss in Section 4.7, modern Application Specific Integrated Circuits (ASICs) and Field-Programmable Gate Arrays (FPGAs) provide millions of logic gates and hundreds of large multi-port embedded memory blocks in a single device. Using a high degree of parallelism, DCFL employs independent search engines for each filter field and aggregates the results of each field search in a distributed fashion; thus, DCFL avoids the exponential increase in time or space incurred when performing this operation in a single step as in the original *Crossproducting* technique discussed in Section 4.4.3.

The first key concept in DCFL is labeling unique field values with locally unique labels. In Figure 4.20, we show the labeling step for the same example filter set used in Figure 4.17. As in *Crossproducting*, DCFL begins by creating sets of unique filter field values. Note that DCFL assigns a locally unique label to each field value and records the number of filters specifying the field value in the “count” value. The count values are incremented and decremented as filters specifying the corresponding fields are added and removed from the filter set. A data structure in a field search engine or aggregation node only needs to be updated when the “count” value changes from 0 to

1 or 1 to 0. Given the sets of labels for each field, we can construct a unique label for each filter by simply concatenating the labels for each field value in the filter. For example, filter j may be uniquely labeled by $(5, 3, 1)$, where the order of field labels is (*Address, Port, Protocol*). The use of labels allows DCFL to aggregate the results from independent field searches using set membership data structures that only store labels corresponding to field values and combinations of field values present in the filter table. As shown in the *Port-Protocol Label Set* in the first aggregation node in Figure 4.21, we represent the unique combinations of port and protocol values specified by filters in the filter set by concatenating the labels for the individual field values[5].

We provide an example of a DCFL search in Figure 4.21 using the filter set and labeling shown in Figure 4.20 and a packet with the following header fields: address 0011, port 1, and protocol TCP. We begin by performing parallel searches on the individual packet fields and returning all matching results. In Figure 4.21 we employ a range tree for the port ranges, a direct lookup table for the protocol fields, and a binary trie for the address prefixes. Note that various options exist for each type of search and DCFL allows each search engine to apply local optimizations. DCFL allows intermediate result aggregation to occur in any order. In our example, we first aggregate the results from the port and protocol field searches. We form the set of all possible matching port-protocol pairs, $F_{query}(y, z)$, by computing the crossproduct of the results from the field searches. Since the field searches returned three port range labels and two protocol field labels, $F_{query}(y, z)$ contains six port-protocol labels. For each label in $F_{query}(y, z)$, we perform a set membership query in the *Port-Protocol Label Set*. Labels that are members of the set are passed on to the next aggregation node. *DCFL* utilizes several efficient data structures for performing set membership queries in aggregation nodes. Note that three port-protocol labels are passed on to the second aggregation node. We perform the same steps to form the set of possible matching filter labels, $F_{query}(x, y, z)$, and probe the *Filter Label Set*. In this example, three filters match the packet. The labels for the matching filters are passed on to a final priority resolution stage that selects the highest priority filter or set of filters.

In addition to the labeling concepts and efficient set membership data structures, we also introduce the concept of *Meta-Labeling* which reduces the memory requirements for aggregation nodes. They also provide techniques for minimizing the number of set membership queries at each aggregation node by computing the optimal ordering of aggregation nodes and limiting the number of labels returned by field search engines. The latter is achieved by a novel technique called *Field Splitting* which we do not discuss in this survey. Using a collection of 12 real filter sets and the *ClassBench* tools suite, we provide analyses of *DCFL* performance and resource requirements on filter sets of various sizes and compositions in Section 7.7. For the 12 real filter sets, we show that the worst-case number of sequential memory accesses is at most ten and memory requirements are at most 40 bytes per filter. Based on these results, an optimized implementation of *DCFL* can

[5]Count values are maintained for the sets of unique fi eld value combinations, like the sets of unique fi eld values shown in Figure 4.20. We do not show the count values in the example in Figure 4.21.

Filter Set

Filter	*Address*	*Port*	*Protocol*	*Label*
a	000*	[0:1]	TCP	(0,0,0)
b	001*	[0:1]	TCP	(1,0,0)
c	1101	[1:1]	UDP	(2,1,1)
d	10*	[5:15]	UDP	(3,2,1)
e	001*	[5:15]	UDP	(1,2,1)
f	111*	[0:15]	UDP	(4,3,1)
g	000*	[5:15]	UDP	(0,2,1)
h	10*	[0:1]	TCP	(3,0,0)
i	001*	[1:1]	TCP	(1,1,0)
j	*	[0:15]	UDP	(5,3,1)
k	*	[0:15]	*	(5,3,2)

Field Sets

Address	*Label*	*Count*
000*	0	2
001*	1	3
1101	2	1
10*	3	2
111*	4	1
*	5	2

Port	*Label*	*Count*
[0:1]	0	3
[1:1]	1	2
[5:15]	2	3
[0:15]	3	3

Protocol	*Label*	*Count*
TCP	0	4
UDP	1	6
*	2	1

Figure 4.20: Example of encoding filters with field labels in *Distributed Crossproducting of Field Labels* (DCFL) using same filter table as Figure 4.17; count values support dynamic updates.

provide over 100 million searches per second and storage for over 200 thousand filters with current generation hardware technology. Like several other packet classification techniques, DCFL provides the freedom to trade off memory for higher throughput. We also show that adding an additional aggregation node increases memory requirements by a modest 12.5 bytes per filter. Based on this observation, we assert that DCFL demonstrates scalability to additional filter fields.

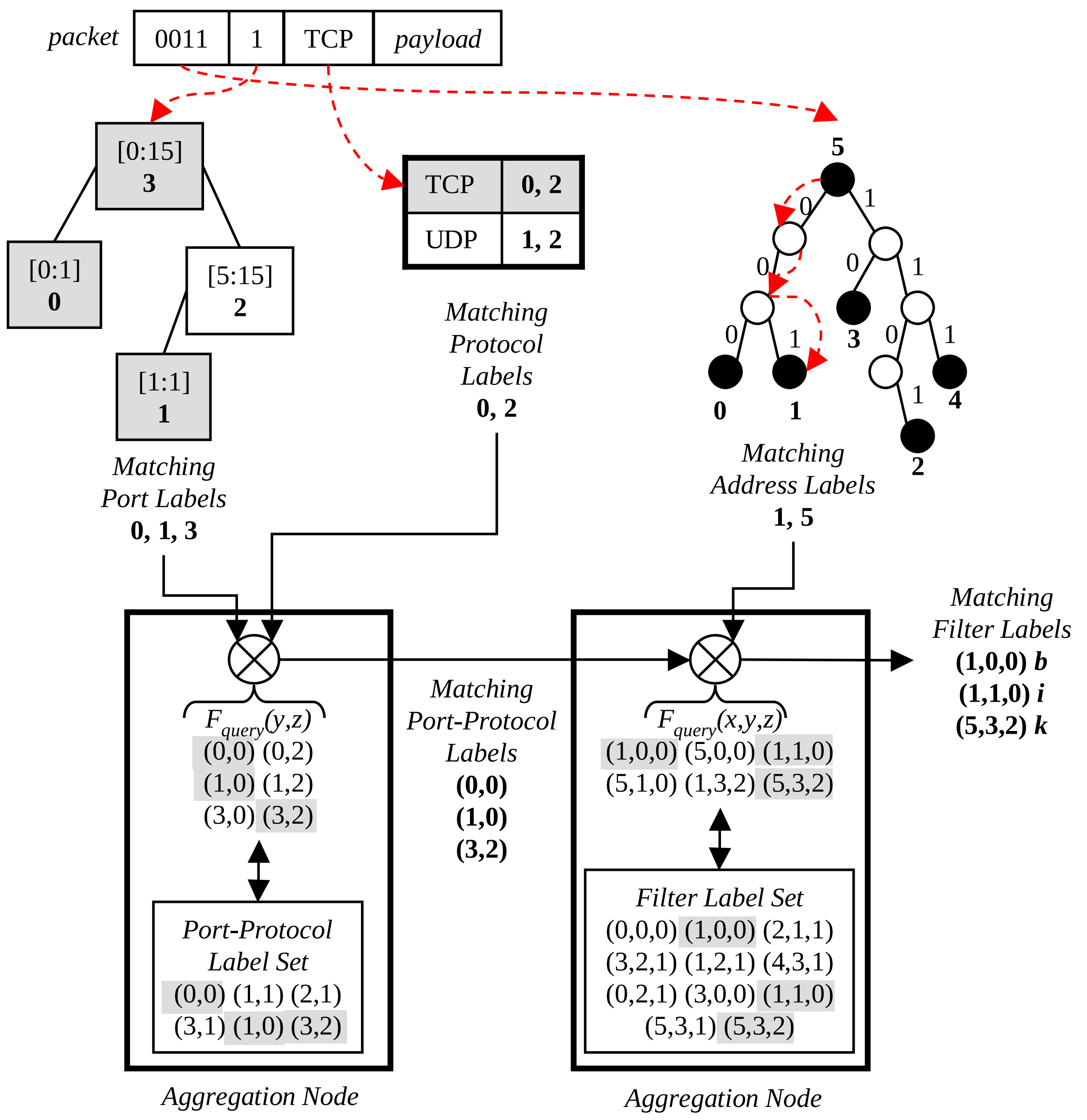


Figure 4.21: Example of search using *Distributed Crossproducting of Field Labels* (DCFL)

4.5 Tuple Space

We have discussed three high-level approaches to the packet classification problem thus far. The last high-level approach in our taxonomy attempts to quickly narrow the scope of a multiple field search by partitioning the filter set by "tuples". A tuple defines the number of specified bits in each field of the filter. Motivated by the observation that the number of distinct tuples is much less than the number of filters in the filter set, Srinivasan, Suri, and Varghese introduced the tuple space approach and a collection of *Tuple Space Search* algorithms in a seminal paper [66].

Table 4.3: Example filter set; address fields are 4-bits and port ranges cover 4-bit port numbers.

Filter	*SA*	*DA*	*SP*	*DP*	*Prot*	*Tuple*
a	$0*$	$001*$	$2:2$	$0:15$	TCP	$[1,3,2,0,1]$
b	$01*$	$0*$	$0:15$	$0:4$	UDP	$[2,1,0,1,1]$
c	0110	0011	$0:4$	$5:15$	TCP	$[4,4,1,1,1]$
d	1100	$*$	$5:15$	$2:2$	UDP	$[4,0,1,2,1]$
e	$1*$	$110*$	$2:2$	$0:15$	UDP	$[1,3,2,0,1]$
f	$10*$	$1*$	$0:15$	$0:4$	TCP	$[2,1,0,1,1]$
g	1001	1100	$0:4$	$5:15$	UDP	$[4,4,1,1,1]$
h	0011	$*$	$5:15$	$2:2$	TCP	$[4,0,1,2,1]$
i	$0*$	$110*$	$2:2$	$0:15$	UDP	$[1,3,2,0,1]$
j	$10*$	$0*$	$2:2$	$2:2$	TCP	$[2,1,2,2,1]$
k	0110	1100	$0:15$	$0:15$	ICMP	$[4,4,0,0,1]$
l	1110	$*$	$2:2$	$0:15$	$*$	$[4,0,2,0,0]$

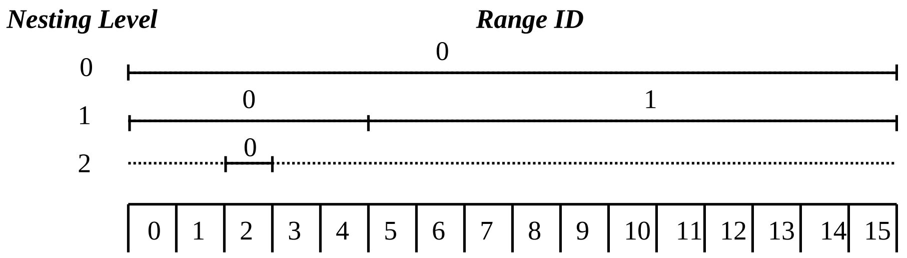


Figure 4.22: Example of assigning tuple values for ranges based on *Nesting Level* and *Range ID*.

In order to illustrate the concept of tuples, we provide an example filter set of filters classifying on five fields in Table 4.3. Address prefixes cover 4-bit addresses and port ranges cover 4-bit port numbers. For address prefix fields, the number of specified bits is simply the number of non-wildcard bits in the prefix. For the protocol field, the value is simply a Boolean: ‘1’ if a protocol is specified, ‘0’ if the wildcard is specified. The number of specified bits in a port range is not as straightforward to define. The authors introduce the concepts of *Nesting Level* and *Range ID* to define the tuple values for port ranges. Similar to the P^2C encoding technique discussed in Section 4.4.5, all ranges on a given port field are placed into a non-overlapping hierarchy. The *Nesting Level* specifies the “layer” of the hierarchy and the *Range ID* uniquely labels the range within its “layer”. In this way, we convert all port ranges to a (*Nesting Level, Range ID*) pair. The *Nesting Level* is used as the tuple value for the range, and the *Range ID* is used to identify the specific range within the tuple. We show an example of assigning *Nesting Level* and *Range ID* for the source port ranges of Table 4.3 in Figure 4.22. Given these definitions of tuple values, we list the tuple of each filter in Table 4.3 in the last column.

Since the tuple specifies the valid bits of its constituent filters, we can probe tuples for matching filters using a fast exact match technique like hashing. We probe a tuple for a matching filter by using the bits of the packet field specified by the tuple as the search key. For example, we construct a search key for the tuple $[1, 3, 2, 0, 1]$ by concatenating the first bit of the packet source address, the first three bits of the packet destination address, the *Range ID* of the source port range at *Nesting Level* 2 covering the packet source port number, the *Range ID* of the destination port range at *Nesting Level* 0 covering the packet destination port number, and the protocol field.

All algorithms using the tuple space approach involve a search of the tuple space or a subset of the tuples in the space. Probes to separate tuples may be performed independently, thus tuple space techniques can take advantage of parallelism. The challenge in designing a parallel implementation lies in the unpredictability of the size of the tuple space or subset to be searched. As a result the realizable lookup performance for tuple space techniques varies widely. Implementations of tuple space techniques can also be made memory efficient due to the effective compression of the filters. The masks or specification of valid bits for filters in the same tuple only needs to be stored once; likewise, only the valid bits of those filters need to be stored in memory. For filter sets with many fields and many wildcards within fields, tuple space techniques can be more space efficient than the $O(N)$ exhaustive techniques discussed in Section 4.2.

4.5.1 Tuple Space Search & Tuple Pruning

The basic *Tuple Space Search* technique introduced by Srinivasan, Suri, and Varghese performs an exhaustive search of the tuple space [66]. For our example filter set in Table 4.3, a search would have to probe seven tuples instead of searching all 12 filters. Using a modest set of real filter sets, the authors found that *Tuple Space Search* reduced the number of searches by a factor of four to seven relative to an exhaustive search over the set of filters[6]. The basic technique can provide adequate performance for large filter sets given favorable filter set properties and a massively parallel implementation.

Motivated by the observation that no address has more than six matching prefixes in backbone route tables, the authors introduced techniques to limit the number of tuples that need to be searched exhaustively. *Pruned Tuple Space Search* reduces the scope of the exhaustive search by performing searches on individual filter fields to find a subset of candidate tuples. While any field or combinations of fields may be used for pruning, the authors found that pruning on the source and destination address strikes a favorable balance between the reduction in candidate tuples and overhead for the pruning steps. We provide an example of pruning on the source and destination addresses in Figure 4.23. In this case, we begin by constructing tries for the source and destination address prefixes in the filter set in Table 4.3. Nodes representing valid prefixes store a list of tuples containing filters that specify the prefix[7]. We begin a *Pruned Tuple Space Search* by performing independent searches of the source and destination tries. The result of each search is a list of all

[6]We make a simplifying assumption that a probe to a tuple is equivalent to examining one filter in an exhaustive search.

[7]Note that the list of tuples may also be represented as a bitmap as in the *Parallel BV* technique.

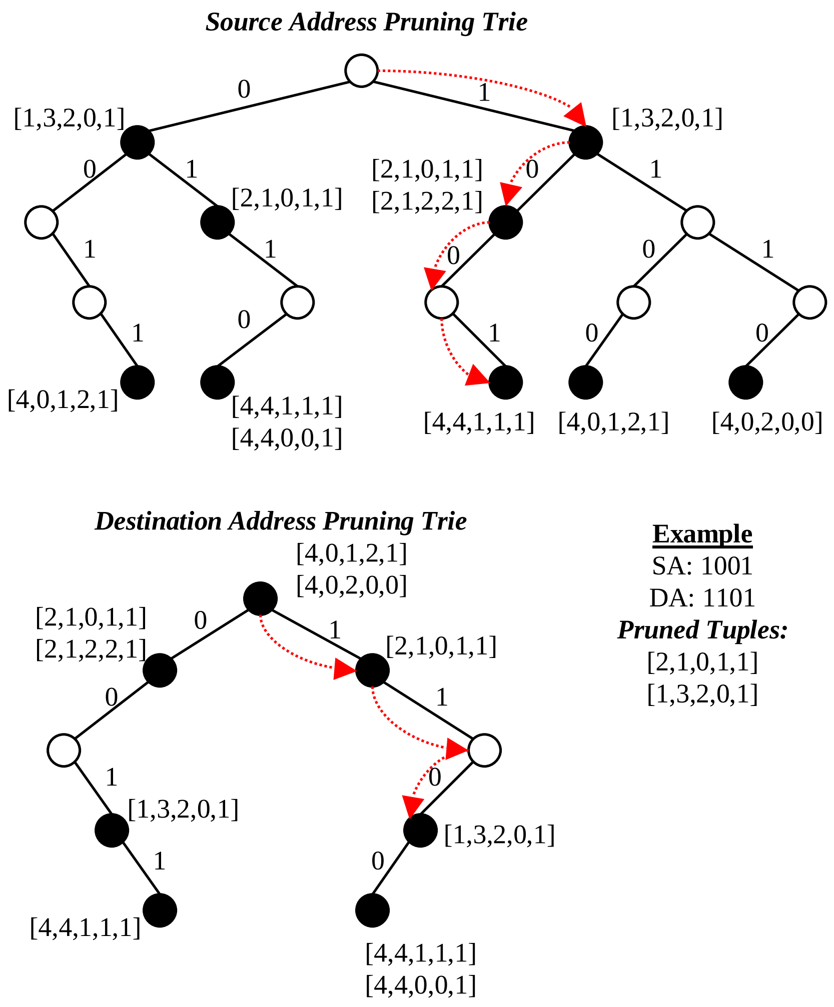


Figure 4.23: Example of *Tuple Pruning* to narrow the scope of the *Tuple Space Search*; the set of *pruned tuples* is the intersection of the sets of tuples found along the search paths for each field.

possible candidate tuples for each field. In order to construct the list of candidate tuples for the packet, we compute the intersection of the tuple lists returned by each search. Note that this is very similar to the *Parallel Bit-Vector* technique discussed in Section 4.4.1. The key difference is that *Pruned Tuple Space Search* computes the candidate *tuples* rather than the overlapping *filters*. In our example in Figure 4.23, we demonstrate pruning for a packet with source address 1001 and destination address 1101. In this case, we only have to probe two tuples instead of seven in the basic search. Using a modest set of real filter sets, the authors found that *Pruned Tuple Space Search* reduced the number of searches by a factor of three to five relative to the basic *Tuple Space Search*, and a factor of 13 to 26 relative to an exhaustive search over the set of filters.

Srinivasan expanded this set of algorithms with *Entry Pruned Tuple Search* (*EPTS*) [67]. This technique seeks to optimize the *Pruned Tuple Search* algorithm by eliminating the need to store a search data structure for each dimension by storing pruning information with *matches* in the

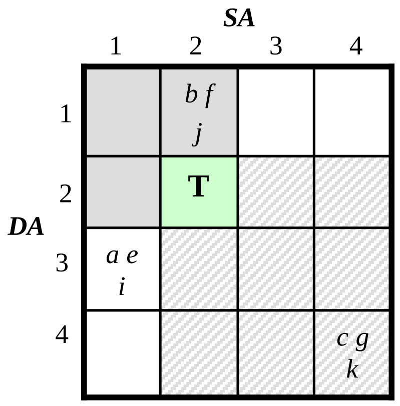


Figure 4.24: Example of *Rectangle Search* on source and destination prefixes of filters in Table 4.3.

tuples. The tuple pruning information is stored with each filter in the form of a bitmap of tuples containing non-conflicting filters. These bitmaps may be precomputed for each filter in the filter set. The author presents an algorithm to compute the tuple bitmaps in $O(TN)$, where T is the number of tuples and N is the number of filters.

4.5.2 Rectangle Search

In their seminal paper, Srinivasan, Suri, and Varghese also present the *Rectangle Search* algorithm that provides theoretically optimal performance for packet classification on two fields without making assumptions about the structure of the filter set. *Rectangle Search* employs the concepts of *markers* and *precomputation* introduced by the *Binary Search on Prefix Lengths* technique for longest prefix matching [24]. As shown in Figure 4.24, the tuple space for filters with two prefix fields may be viewed as a grid of rectangles where each rectangle is a tuple. For this example, we use the source and destination addresses of the filters in the example filter set shown in Table 4.3[8]. Implementing an exhaustive search over the grid of tuples requires W^2 probes in the worst case.

The strategy of *Rectangle Search* is to leverage precomputation and markers to limit the number of probes to at most $(2W - 1)$ where W is the address length. Each filter mapping to a tuple $[i, j]$ leaves a *marker* in each tuple to its left in its row. For example, a filter $(110*, 0111)$ stored in tuple $[3, 4]$ leaves *markers* $(11*, 0111)$ in $[2, 4]$ and $(1*, 0111)$ in $[1, 4]$. For all filters and markers in a tuple $[i, j]$, we can *precompute* the best matching filter from among the filters stored in less specific tuples. Consider tuple $[2, 2]$, labeled *T* in Figure 4.24. Graphically, less specific tuples are those in the shaded quadrant above and left of T in Figure 4.24. For example, if a marker $[01*, 00*]$ were stored in T, then we would *precompute* the best matching filter b and store it with the marker in T.

[8]Note that filters containing a wildcard are not included; these filters may be searched by maintaining separate search tries.

Rectangle Search begins at tuple $[1, W]$, the bottom-left tuple. If a matching filter is found, then we need not search any tuples above and left of this tuple due to precomputation. The search moves one tuple to the right to the next tuple in the row. If no matching filter is found in the tuple, then we need not search search any tuples below and right of this tuple due to markers. The search moves one tuple up to the next tuple in the column. Note that the worst-case search path follows a staircase pattern from tuple $[1, W]$ to tuple $[W, 1]$ probing a total of $(2W - 1)$ tuples. *Rectangle Search* requires $O(NW)$ memory as each filter may leave a marker in at most W tuples to its left in its row. The authors proved that $(2W - 1)$ probes is the theoretical lower bound for two fields and extend this bound to show that for d fields the lower bound is:

$$\frac{W^{(d-1)}}{d!} \tag{4.5}$$

4.5.3 Conflict-Free Rectangle Search

Warkhede, Suri, and Varghese provide an optimized version of *Rectangle Search* for the special case of packet classification on a *conflict-free* filter set [68]. A filter set is defined to be *conflict-free* if there is no pair of overlapping filters in the filter set such that one filter is more specific than the other in one field and less specific in another field. The authors observe that in real filter sets conflicts are rare; furthermore, techniques exist to resolve filter conflicts by inserting a small set of *resolving filters* that resolve filter conflicts [69].

Conflict-Free Rectangle Search begins by mapping the filter set to the $W \times W$ tuple space. Using precomputation and markers, the authors prove that a binary search can be performed on the columns of the grid due to the *conflict-free* nature of the filter set. This provides an $O(\log^2 w)$ bound on the number of tuple probes and an $O(n \log^2 w)$ bound on memory.

4.6 Caching

Finally, we briefly discuss caching, a general technique that can be combined with any search technique to improve average performance. A cache is a fast storage buffer for commonly referenced data. If data requests contain sufficient locality, the average time to access data is significantly reduced when the time to access the cache is significantly less than the time to access other storage media [70]. In the context of packet classification, the lookup time is significantly reduced if the time to perform a cache query is significantly less than the time to perform a full lookup. The efficacy of caching schemes largely depends on the data request patterns of the application.

Caching techniques have met with much skepticism from the research community due to the “wire-speed requirement” discussed in Section 1.3.1. In short, improving average case performance is irrelevant if we we evaluate packet classification techniques based on worst-case performance. Another argument against caching is the perception that packet flows on high-speed links lack locality of reference. As link speeds have increased, caching schemes have also met with increasing skepticism due to the question of sufficient temporal locality. This question arises due to the fact that the bandwidth requirement of the average packet flow has not increased at the same rate as link capacity. To put it simply: as link bandwidth increases, the number of flows sharing the link also increases. In order for a caching scheme to retain its effectiveness, we must scale the size of the cache with the link speed. Consider the example of a 10 Gb/s link supporting individual flows with peak rates of at most 1 Mb/s. The packet of a given flow will appear at most once in ten thousand packets, thus the cache must have a minimum capacity of ten thousand entries.

Despite the skepticism, a number of cache designs for packet classification have emerged [71, 72, 73]. One intriguing design utilizes Bloom filters and allows for a small probability of misclassification [71]. Holding the misclassification probability to approximately one in a billion, the authors measured an average cache hit-rate of approximately 95 percent using 4KB of memory and real packet traces collected from an OC-3 link; thus, only five percent of the traffic required a full classification lookup. While these results are compelling for low-speed links, the viability of caching for OC-192 (10 Gb/s) links remains an open question. It is a difficult one to answer due to the technical challenges of collecting packet traces from such high-speed links. If we simply scale the size of the cache with link speed, this Bloom filter approach would require 256k bytes of cache memory which may be prohibitively large in some applications.

4.7 Discussion

We have presented a survey of packet classification techniques. Using a taxonomy based on the high-level approach to the problem and a minimal set of running examples, we attempted to provide the reader with a deeper understanding of the seminal and recent algorithms and architectures, as well as a useful framework for discerning the relationships and distinctions. While we mentioned the simulation results reported by the authors of the literature introducing each technique, we consciously avoided a direct comparison of the techniques based on throughput, memory requirements, or update performance. Given the various implementation options and variability in simulation parameters, a fair comparison using those metrics is difficult. We believe that future high-performance packet classifiers will invariably be implementations of hybrid techniques that borrow ideas from a number of the previously described techniques. In closing, we would like to briefly highlight the implementation platforms for current and future packet classifiers.

Thanks to the endurance of Moore's Law, integrated circuits continue to provide better performance at lower cost. Application Specific Integrated Circuits (ASICs) and Field-Programmable Gate Arrays (FPGAs) provide millions of logic gates and millions of bits of memory distributed across many multi-port embedded memory blocks. For example, a current generation Xilinx FPGA operates at over 400 MHz and contains 556 dual-port embedded memory blocks, 18Kb each with 36-bit wide data paths for a total of over 10Mb of embedded memory [44]. Current ASIC standard cell libraries offer dual- and quad-port embedded SRAMs operating at 625MHz [74]. It is standard practice to utilize several embedded memories in parallel in order to achieve wide data paths. Dual Data Rate (DDR) and Quad Data Rate (QDR) SRAM technologies provide high bandwidth interfaces to several mega-bytes of off-chip memory [63, 75]. Network processors also provide a flexible platform for implementing packet classification techniques [76, 77, 78]. A number of current generation processors provide hardware assists for packet classification, interfaces to TCAM, and/or special instructions for search applications such as hash functions.

Chapter 5

Analysis of Real Filter Sets

There are three kinds of lies: lies, damned lies, and statistics.
Benjamin Disraeli, British Prime Minister (1868, 1874-1878)

Recent efforts to identify better packet classification techniques have focused on leveraging the characteristics of real filter sets for faster searches. While lower bounds for the general multi-field searching problem have been established, observations made in recent packet classification work offer enticing new possibilities to provide significantly better performance. In this chapter, we summarize the observations made in the literature and report the results of our additional analyses. We also seek to identify and understand the impetus for the observed structure of filter sets, to report other potentially useful characteristics for increasing the performance of packet classifiers, and to develop metrics and characterizations of filter set structure that aid in generating synthetic filter sets.

We performed a battery of analyses on 12 real filter sets provided by Internet Service Providers (ISPs), a network equipment vendor, and other researchers working in the field. The filter sets range in size from 68 to 4557 entries and utilize one of the following formats:

- Access Control List (ACL) - standard format for security, VPN, and NAT filters for firewalls and routers (enterprise, edge, and backbone)
- Firewall (FW) - proprietary format for specifying security filters for firewalls
- IP Chain (IPC) - decision tree format for security, VPN, and NAT filters for software-based systems

Due to confidentiality concerns, the filter sets were provided without supporting information regarding the types of systems and environments in which they are used. We are unable to comment on "where" in the network architecture the filter sets are used: enterprise core routers, ISP edge routers, backbone core routers, enterprise firewalls, etc. Nonetheless, the following analysis provide invaluable insight into the structure of real filter sets. We observe that various useful properties hold regardless of filter set size or format. The results of these analyses provide the foundation for the benchmarking tools described in Chapter 6 and the basis for the new packet classification technique, *Distributed Crossproducting of Field Labels* (DCFL), described in Chapter 7.

5.1 Understanding Filter Composition

Many of the observed characteristics of filter sets arise due to the administrative policies that drive their construction. The most complex packet filters typically appear in firewall and edge router filter sets due to the heterogeneous set of applications supported in these environments. Firewalls and edge routers typically implement security filters and network address translation (NAT), and they may support additional applications such as Virtual Private Networks (VPNs) and resource reservation. Typically, these filter sets are created manually by a system administrator using a standard management tool such as CiscoWorks VPN/Security Management Solution (VMS) [79] and Lucent Security Management Server (LSMS) [80]. Such tools conform to a model of filter construction which views a filter as specifying the communicating subnets and the application or set of applications. Hence, we can view each filter as having two major components: an address prefix pair and an application specification. The address prefix pair identifies the communicating subnets by specifying a source address prefix and a destination address prefix. The application specification identifies a specific application session by specifying the transport protocol, source port number, and destination port number. A set of applications may be identified by specifying ranges for the source and destination port numbers.

5.2 Previous Observations

Gupta and McKeown published a number of observations regarding the characteristics of real filter sets which have been widely cited [50]. Others have performed analyses on real filter sets and published their observations [58, 54, 28, 29, 77]. The following is a distillation of observations relevant to our discussion:

- Current filter set sizes are small, ranging from tens of filters to less than 5000 filters. It is unclear if the size limitation is “natural” or a result of the limited performance and high expense of existing packet classification solutions.
- The protocol field is restricted to a small set of values. In most filter sets, TCP, UDP, and the wildcard are the most common specifications; other specifications include ICMP, IGMP, (E)IGRP, GRE and IPINIP.
- Transport-layer specifications vary widely. Common range specifications for port numbers such as ‘gt 1023’ (greater than 1023) suggest that the use of range to prefix conversion techniques may be inefficient.

- The number of unique address prefixes matching a given address is typically five or less.
- Most prefixes have either a length of 0 or 32; there are some prefixes with lengths of 21, 23, 24 and 30.
- The number of filters matching a given packet is typically five or less.
- Different filters often share a number of the same field values.

The final observation has been a source of deeper insight and a springboard for several recent contributions in the area. We thoroughly explore the implications of this observation in Section 5.7.

Kounavis, et. al. performed a thorough analysis of four ACLs and proposed a general framework for packet classification in network processors [77]. The authors made a number of interesting observations and assertions. Specifically, they observed a dependency between the size of the ACL and the number of filters that have a wildcard in the source or destination IP address. The authors refer to filters that contain a wildcard in either the source or destination address as "partially specified". They found that partially specified filters comprise a smaller proportion of the filter set as the number of filters increases. Specifically, 83% of the filters in the smallest ACL were partially specified while only 10% of the filters in the largest ACL were partially specified. The authors also observed trends in the composition of partially specified filters. The smallest ACL from an enterprise firewall had large numbers of partially specified filters with destination address wildcards, while the largest ACL from an ISP had large numbers of partially specified filters with source address wildcards. The authors suggest that these characteristics are a result of the "location" of the ACL in the Internet. Small ACLs are "closer" to client subnets, therefore filters are used to block traffic flows from a number of specific subnets. Large ACLs are "closer" to the Internet backbone, thus filters are used to control access to a number of important servers or networks.

Kounavis, et. al. also found that the number of filters matching a packet is typically four with a maximum of seven. Inspired by the previously described model of filter construction, they also investigated the possibility of first classifying a packet on the address prefix pair. The authors performed an analysis of the overlap properties of address prefix pairs specified by the filter set. Address prefix pairs overlap if they cover a common address pair or set of address pairs. An example is shown in Figure 5.1. They found that a majority of the overlaps are caused by partially specified filters, but the number of overlaps is orders of magnitude less than the theoretical upper bound. Based on these results, the authors argue that such overlaps may be eliminated by inserting a small number of filters that cover the overlaps caused by partially specified filters. This is essentially independent verification of the findings of Hari, et. al. [69] and a similar approach to that employed by Warkhede, et. al. [68]. Finally, Kounavis, et. al. found that the number of unique application specifications (combination of transport protocol and port ranges) is small due to the limited number of popular applications in the Internet.

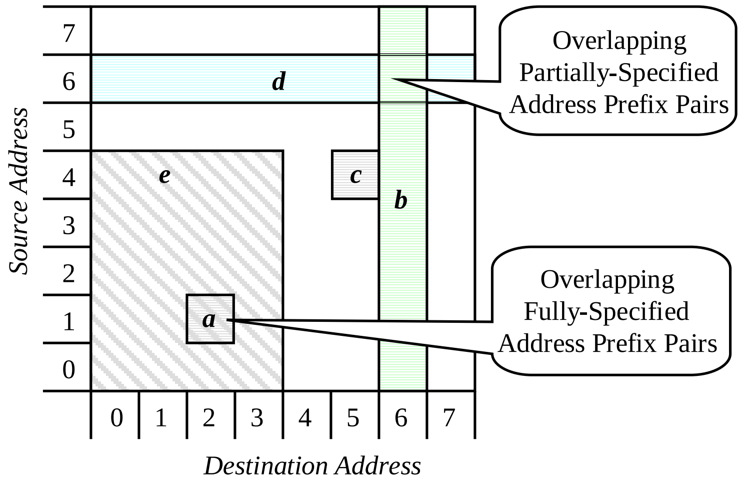

Overlapping Partially-Specified Address Prefix Pairs
Overlapping Fully-Specified Address Prefix Pairs
Source Address
Destination Address
a
b
c
d
e
0
1
2
3
4
5
6
7

Figure 5.1: Example of overlaps formed by fully-specified and partially-specified address prefix pairs.

5.3 Application Specifi cations

We analyzed the application specifications in the 12 filter sets in order to corroborate previous observations as well as extract new, potentially useful characteristics.

5.3.1 Protocol

For each of the filter sets, we examined the unique protocol specifications and the distribution of filters over the set of unique values. As shown in Table 5.1, filters specified one of nine protocols or the wildcard. Note that two filter sets, fw2 and fw4, contain anonymized protocol numbers; therefore, we did not include them in our analysis. We observed the following protocol specifications, listed in order of frequency of occurrence:

- Transmission Control Protocol (TCP), RFC793 [81]
- User Datagram Protocol (UDP), RFC768 [82]
- Wildcard
- Internet Control Message Protocol (ICMP), RFC792 [83]
- General Routing Encapsulation (GRE), RFC2784 [84]
- Open Shortest Path First (OSPF) Interior Gateway Protocol (IGP), RFC2178 [85]
- Enhanced Interior Gateway Routing Protocol (EIGRP), Cisco [86]
- IP Encapsulating Security Payload (ESP) for IPv6, RFC2406 [87]

Table 5.1: Observed protocols and filter distribution; values given as percentage (%) of filters in the filter set.

Set	$*$	*ICMP*	*IPE*	*TCP*	*UDP*	*GRE*	*ESP*	*AH*	*EIGRP*	*OSPF IGP*
acl1	8.46	3.14	0.00	87.31	1.09	0.00	0.00	0.00	0.00	0.00
acl2	46.39	0.96	0.00	44.94	6.74	0.00	0.00	0.00	0.96	0.00
acl3	4.92	4.17	0.00	65.00	25.87	0.00	0.00	0.00	0.00	0.04
acl4	4.08	3.99	0.10	65.76	25.87	0.16	0.00	0.00	0.00	0.03
acl5	0.00	28.59	0.00	28.22	41.78	0.00	0.00	0.00	0.00	1.40
fw1	1.06	3.89	0.00	57.24	32.16	5.65	0.00	0.00	0.00	0.00
fw3	1.63	5.98	0.00	55.98	36.41	0.00	0.00	0.00	0.00	0.00
fw5	1.88	6.87	0.00	51.88	39.38	0.00	0.00	0.00	0.00	0.00
ipc1	34.49	1.12	0.00	26.15	37.72	0.29	0.12	0.12	0.00	0.00
ipc2	27.08	36.46	0.00	10.42	26.04	0.00	0.00	0.00	0.00	0.00
AVG	**13.00**	**9.52**	**0.01**	**49.29**	**27.31**	**0.61**	**0.01**	**0.01**	**0.10**	**0.15**

- IP Authentication Header (AH) for IPv6, RFC2402 [88]
- IP Encapsulation within IP (IPE), RFC2003 [89]

Like previous studies, the most common protocol specification is TCP. On average, TCP is specified by twice as many filters as the next most common protocol, UDP. The wildcard is the third most common specification. All filter sets contain a small number of filters specifying ICMP. The remaining six protocols are only specified by a few filters in a few of the filter sets.

5.3.2 Port Ranges

Next, we examined the port ranges specified by filters in the filter sets and the distribution of filters over the unique values. In order to observe trends among the various filter sets, we define five classes of port ranges:

- WC, wildcard
- HI, ephemeral user port range $[1024 : 65535]$
- LO, well-known system port range $[0 : 1023]$
- AR, arbitrary range
- EM, exact match

Motivated by the allocation of port numbers, the first three classes represent common specifications for a port range. The last two classes may be viewed as partitioning the remaining specifications based on whether or not an exact port number is specified. Table 5.2 shows the distribution of

Table 5.2: Distribution of filters over the five port classes for source and destination port range specifications; values given as percentage (%) of filters in the filter set.

Set	*Source Port*					*Destination Port*				
	WC	*HI*	*LO*	*AR*	*EM*	*WC*	*HI*	*LO*	*AR*	*EM*
acl1	100.0	0.00	0.00	0.00	0.00	30.42	0.00	0.00	11.60	57.98
acl2	100.0	0.00	0.00	0.00	0.00	69.34	0.64	0.00	7.06	22.95
acl3	99.92	0.00	0.00	0.00	0.08	9.25	13.96	0.00	11.04	65.75
acl4	99.93	0.00	0.00	0.00	0.07	8.56	12.15	0.00	11.21	68.08
acl5	100.0	0.00	0.00	0.00	0.00	30.00	4.08	0.00	5.20	60.72
fw1	77.74	8.13	0.00	0.35	13.78	31.10	8.13	0.00	0.35	60.42
fw2	38.24	17.65	0.00	0.00	44.12	100.0	0.00	0.00	0.00	0.00
fw3	77.72	5.98	0.00	0.54	15.76	27.72	5.98	0.00	0.54	65.76
fw4	10.98	42.05	10.98	1.52	34.47	13.26	18.94	0.76	1.14	65.91
fw5	75.62	5.00	0.00	0.62	18.75	35.62	3.75	0.00	1.25	59.38
ipc1	82.84	0.35	0.00	2.00	14.81	55.46	6.52	0.00	2.53	35.49
ipc2	73.96	0.00	0.00	0.00	26.04	73.96	0.00	0.00	0.00	26.04
AVG	**78.08**	**6.60**	**0.92**	**0.42**	**13.99**	**40.39**	**6.18**	**0.06**	**4.33**	**49.04**

filters over the five port classes for both source and destination ports. We observe some interesting trends in the data. With rare exception, the filters in the ACL filter sets specify the wildcard for the source port. A majority of filters in the ACL filters specify an exact port number for the destination port. Source port specifications in the other filter sets are also dominated by the wildcard, but a considerable portion of the filters specify an exact port number. Destination port specifications in the other filter sets share the same trend, however the distribution between the wildcard and exact match is a bit more even. After the wildcard and exact match, the HI port class is the most common specification. A small portion of the filters specify an arbitrary range, 4% on average and at most 12%. Only one filter set contained filters specifying the LO port class for either the source or destination port range.

In the interest of designing efficient data structures, we now examine the number of unique specifications in the AR and EM classes. Checking for matches in the first three classes is trivial. As shown in Table 5.3, the number of unique specifications in the AR class is small relative to the size of the filter set. Consisting of 50 ranges, the largest set of arbitrary ranges may be efficiently searched using a simple interval tree. Likewise the number of specifications in the EM class is also small, thus a simple hash table would be sufficient to search this set of ranges.

5.3.3 Port Pair Class

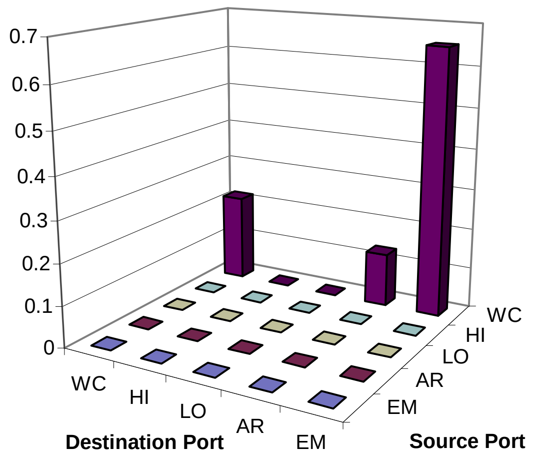
As previously discussed, the structure of source and destination port range pairs is a key point of interest for both modeling real filter sets and designing efficient search algorithms. We can characterize this structure by defining a *Port Pair Class* (PPC) for every combination of source and

Table 5.3: Number of unique specifications in the Arbitrary Range (AR) and Exact Match (EM) port classes for source and destination port ranges.

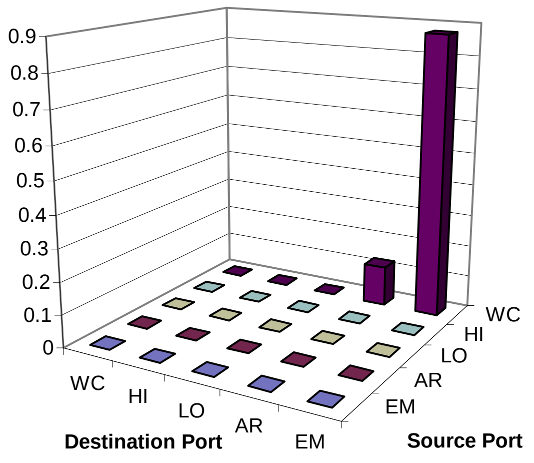
Set	*Size*	*Source Port*		*Destination Port*	
		AR	*EM*	*AR*	*EM*
acl1	733	0	0	34	73
acl2	623	0	0	1	24
acl3	2400	0	2	36	152
acl4	3061	0	2	50	183
acl5	4557	0	0	3	35
fw1	283	1	10	1	40
fw2	68	0	7	0	0
fw3	184	1	6	1	36
fw4	264	3	22	3	44
fw5	160	1	8	2	29
ipc1	1702	5	27	7	45
ipc2	192	0	2	0	2

destination port class. For example, WC-WC if both source and destination port ranges specify the wildcard, AR-LO if the source port range specifies an arbitrary range and the destination port range specifies the set of well-known system ports. As shown in Figure 5.2, a convenient way to visualize the structure of *Port Pair Classes* is to define a *Port Pair Class* matrix where rows share the same source port class and columns share the same destination port class. For each filter set, we examine the PPC defined by filters specifying the same protocol. For all protocols except TCP and UDP, the PPC is trivial – a single spike at WC/WC. Figure 5.2 highlights the uniqueness of PPC matrices among different protocols and filter sets.

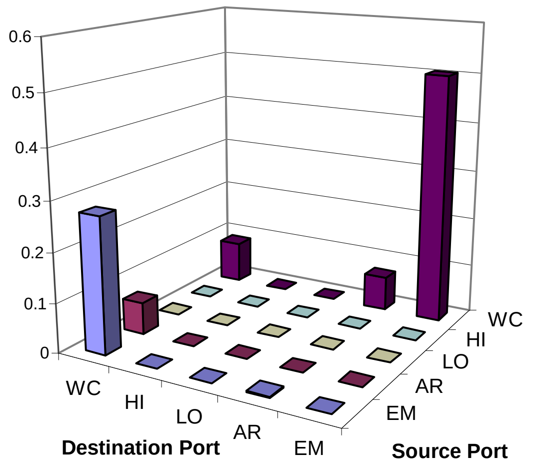
The combination of source and destination port range specifications has a significant effect on several packet classification techniques. This is especially true of TCAM due to the need to convert arbitrary range pairs into pairs of prefixes. See Section 4.2.2 for a discussion of TCAMs and the need for range conversion. In order to assess the effect of this conversion, we computed the number of TCAM entries required to store each filter set. We refer to the *Expansion Factor* as the ratio of TCAM entries to filter set size, which can be thought of as the average number of TCAM entries required by each filter in the filter set. As shown in Table 5.4, a filter set may require that a TCAM provide more than six entries for every filter. On average, the filter sets required 2.25 entries per filter. While this is considerably less than the worst case of 900 entries per filter, yet it remains a large source of inefficiency. The magnitude of the *Expansion Factor* is not the only challenge. Note the high variance in the *Expansion Factor* among the filter sets; this presents a challenge in designing systems, as the filter storage capacity varies widely with filter set composition.



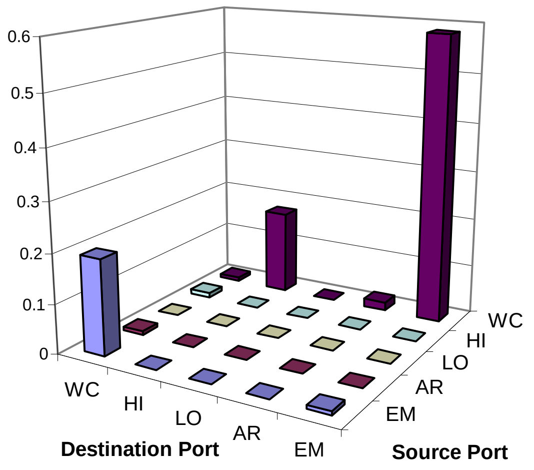
(a) acl1, TCP



(b) acl1, UDP



(c) fw4, TCP



(d) fw4, UDP

Figure 5.2: Port Pair Matrices for two filter sets.

5.4 Address Prefix Pairs

A filter identifies communicating hosts or subnets by specifying a source and destination address prefix, or address prefix pair. The speed and efficiency of several longest prefix matching and packet classification algorithms depend upon the number of unique prefix lengths and the distribution of filters across those unique values. We begin our analysis by examining the number of unique prefix lengths. In Table 5.5 we report the number of unique source address prefix lengths, destinations address prefix lengths, and source/destination prefix pair lengths for the 12 filter sets. A majority of the filter sets have more unique source address prefix lengths than unique destination prefix lengths. For all of the filter sets, the number of unique source/destination prefix pair lengths is small relative to the filter set size and the number of possible combinations, 1024.

Table 5.4: Number of entries required to store filter set in a standard TCAM.

Set	*Size*	*TCAM Entries*	*Expansion Factor*
acl1	733	997	1.3602
acl2	623	1259	2.0209
acl3	2400	4421	1.8421
acl4	3061	5368	1.7537
acl5	4557	5726	1.2565
fw1	283	998	3.5265
fw2	68	128	1.8824
fw3	184	554	3.0109
fw4	264	1638	6.2045
fw5	160	420	2.6250
ipc1	1702	2332	1.3702
ipc2	192	192	1.0000
Average			**2.3211**

Table 5.5: Number of unique address prefix lengths for source address (SA), destination address (DA), and source/destination address pairs (SA/DA).

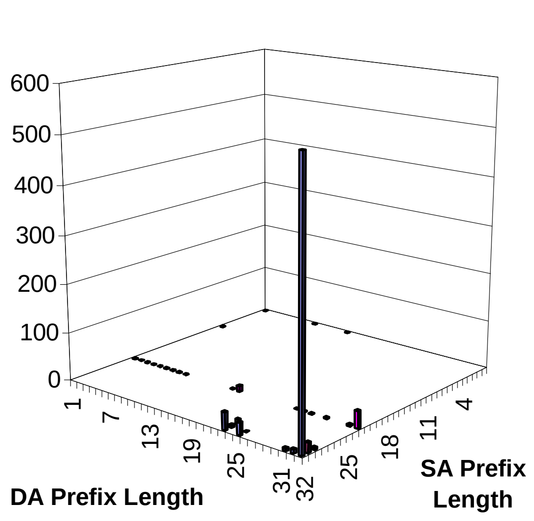
Set	*Size*	*SA*	*DA*	*SA/DA*
acl1	733	6	20	31
acl2	623	13	13	50
acl3	2400	22	12	89
acl4	3061	22	15	98
acl5	4557	11	3	31
fw1	283	12	6	22
fw2	68	4	3	8
fw3	184	9	3	13
fw4	264	5	6	12
fw5	160	10	4	17
ipc1	1702	15	13	93
ipc2	192	4	2	5

Next, we examine the distribution of filters over the unique address prefix pair lengths. Note that this study is unique in that previous studies and models of filter sets utilized independent distributions for source and destination address prefixes. When constructing synthetic filter sets to test new packet classification algorithms, researchers often randomly select address prefixes from backbone route tables which are dominated by class C address prefixes (24-bit network address) and aggregates of class A, B, and C address prefixes. As shown in Figure 5.3, real filter sets have unique prefix pair distributions that reflect the types of filters contained in the filter set. For example, fully specified source and destination addresses dominate the distribution for acl5 shown in Figure 5.3(a). There are very few filters specifying a 24-bit prefix for either the source or destination address. Also consider the distribution for fw1 shown in Figure 5.3(c). The most common prefix pair is a fully specified destination address and a wildcard for the source address. This is due to the nature of the filter set, a firewall limiting access to a key host. It is not possible to model the prefix pair distribution using independent prefix length distributions, even if those distributions are taken from real filter sets. Finally, we observe that while the distributions are sufficiently different from each other a majority of the filters in the filter sets specify prefix pair lengths around the "edges" of the distribution. Note that there are not large "spikes" in or around the centers of the distributions in Figure 5.3. This implies that, typically, one of the address prefixes is either fully specified or wildcarded.

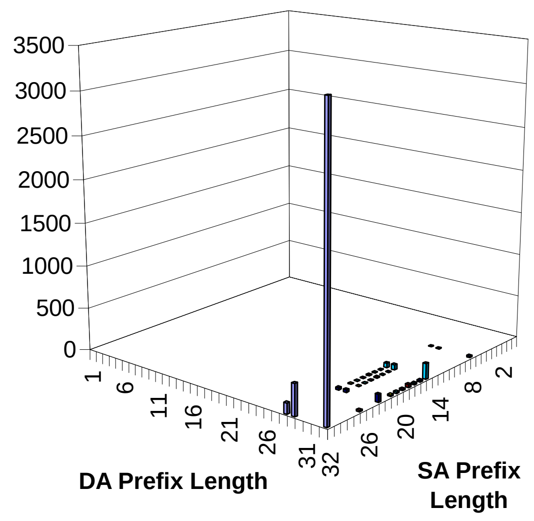
By considering the prefix pair distribution, we characterize the *size* of the communicating subnets specified by filters in the filter set. Next, we would like to characterize the relationships among address prefixes and the amount of address space covered by the prefixes in the filter set. Our primary motivation is to devise metrics to facilitate construction of synthetic filter sets that accurately model real filter sets. Consider a binary tree constructed from the IP source address prefixes of all filters in the filter set. From this tree, we could completely characterize the data structure by determining a branching probability for each node. For example, assume that an address prefix is generated by traversing the tree starting at the root node. At each node, the decision to take to the 0 path or the 1 path exiting the node depends upon the branching probability at the node. For a complete characterization of the tree, the branching probability at each node is unique. As shown in Figure 5.4, $p\{0|11\}$ is the probability that the 1 path is chosen at level 2 given that the 1 path was chosen at level 0 and the 1 path was chosen at level 1.

Such a characterization is infeasible, hence we employ suitable metrics that capture the important characteristics while providing a convenient abstraction. We begin by constructing two binary tries from the source and destination prefixes in the filter set. Note that there is one level in the tree for each possible prefix length 0 through 32 for a total of 33 levels. For each level in the tree, we compute the probability that a node has one child or two children. Nodes with no children are excluded from the calculation. We refer to this distribution as the *Branching Probability*.

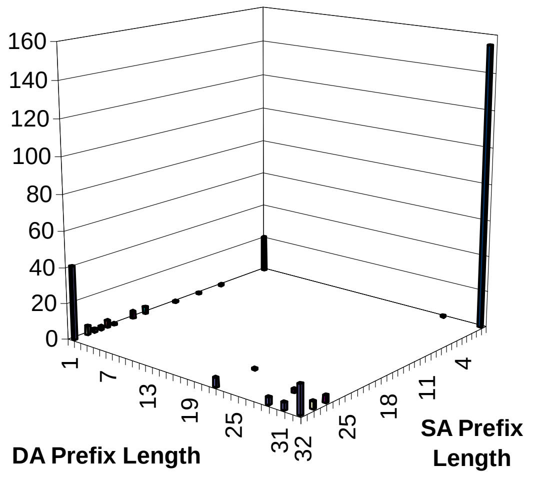
For nodes with two children, we compute *skew*, which is relative measure of the weights of the left and right subtrees of the node. Subtree weight is defined to be the number of filters



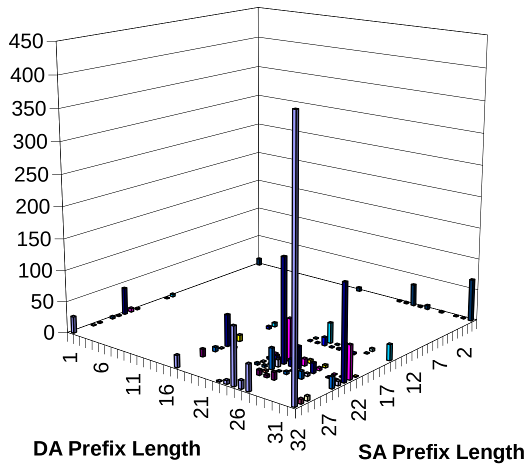
(a) acl1



(b) acl5



(c) fw1



(d) ipc1

Figure 5.3: Prefix length distribution for address prefix pairs.

specifying prefixes in the subtree, not the number of prefixes in the subtree. This definition of weight accounts for "popular" prefixes that occur in many filters. Let $heavy$ be the subtree with the largest weight and let $light$ be the subtree with equal or less weight. The following is a precise definition of skew:

$$skew = 1 - \frac{weight(light)}{weight(heavy)} \tag{5.1}$$

Note that this definition of skew provides an anonymous measure of address prefix structure, as it does not preserve address prefix values. Consider the following example: given a node k with two children at level m, assume that 10 filters specify prefixes in the 1-subtree of node k (the subtree

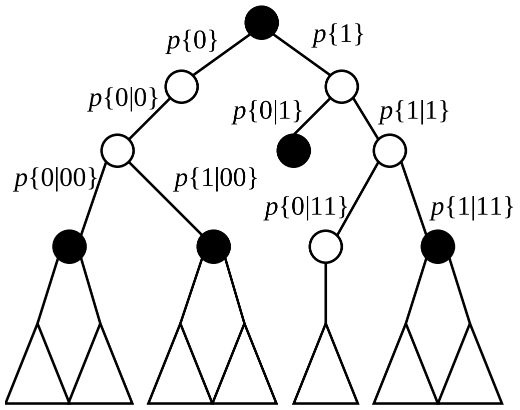


Figure 5.4: Example of complete statistical characterization of address prefixes.

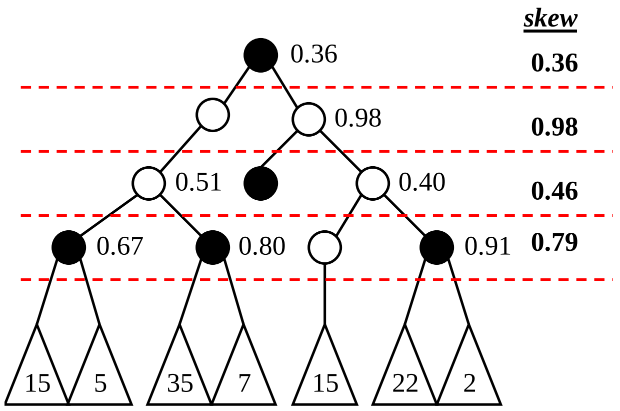
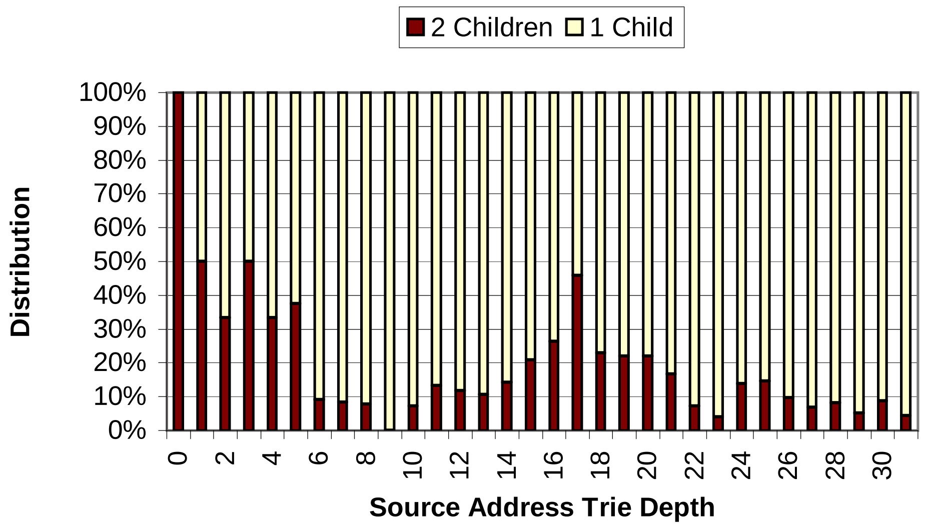


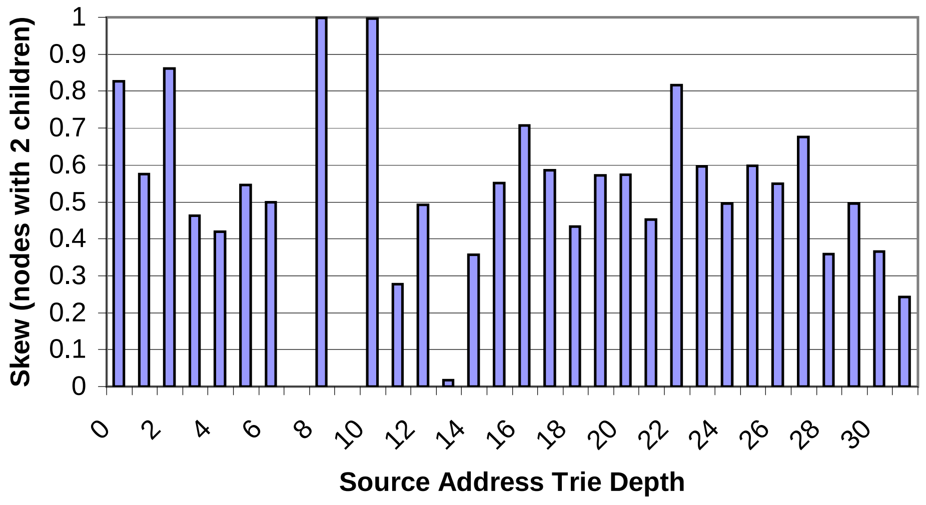
Figure 5.5: Example of skew computation for the first four levels of an address trie; shaded nodes denote a prefix specified by a single filter; subtrees denoted by triangles with associated weight.

visited if the next bit of the address is 1) and 25 filters specify prefixes in the 0-subtree of node k. The 1-subtree is the $light$ subtree, the 0-subtree is the $heavy$ subtree, and the skew at node k is 0.6. We compute the average skew for all nodes with two children at level m, record it in the distribution, and move on to level $(m + 1)$. We provide and example of computing skew for the first four levels of an address trie in Figure 5.5.

The result of this analysis is two distributions for each address trie, a *branching probability* distribution and a *skew* distribution. We plot these distributions for the source address prefixes in filter set acl5 in Figure 5.6. In Figure 5.6(a), note that a significant portion of the nodes in levels zero through five have two children, but the amount generally decreases as we move down the trie. The increase at level 16 and 17 is a notable exception. This implies that there is a considerable amount of branching near the “top” of the trie, but the paths generally remain contained as we move down the trie. In Figure 5.6(b), we observe that skew for nodes with two children hovers around 0.5, thus the one subtree tends to contain prefixes specified by twice as many filters as the other subtree. Note that skew is not defined at levels where all nodes have one child. Also note that levels containing



(a) Source address branching probability; average per level.

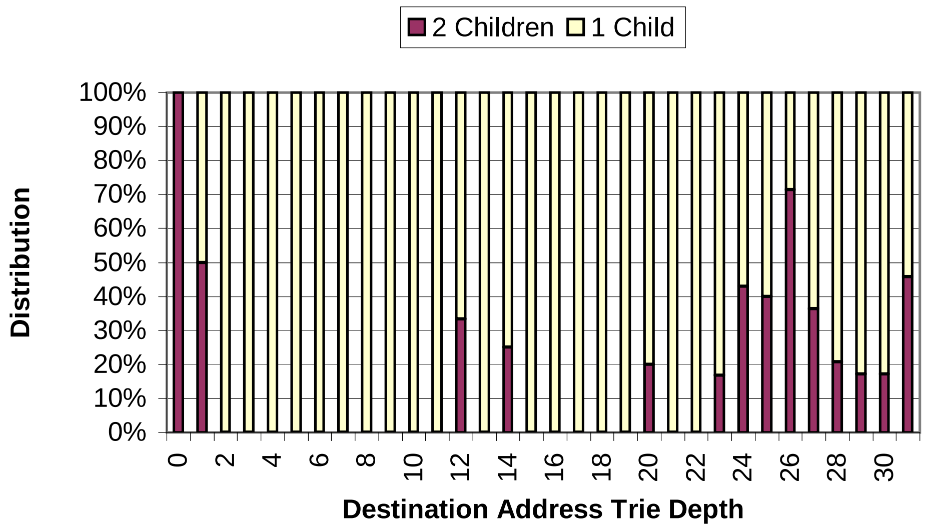


(b) Source address skew; average per level for nodes with two children.

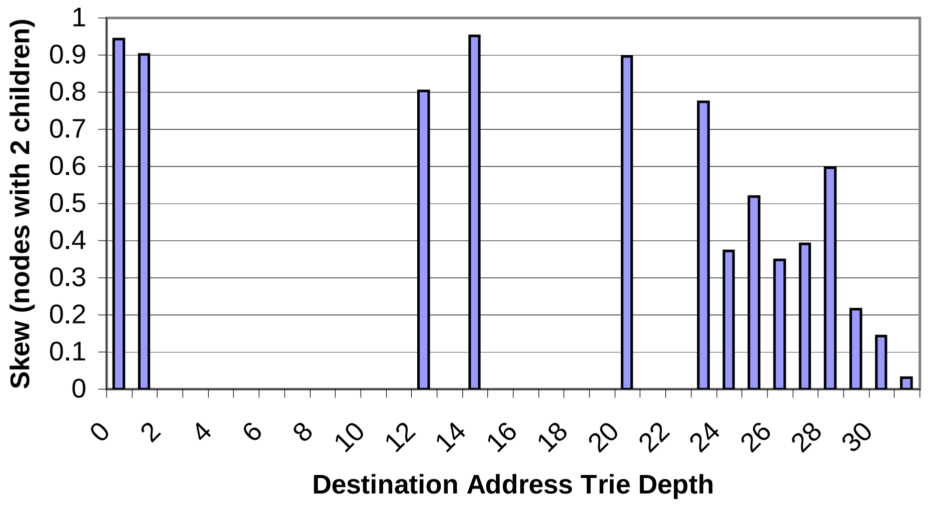
Figure 5.6: Source address branching probability and skew for filter set acl5.

nodes with two children may have an average skew of zero (completely balanced subtrees), but this is rare.

We plot the branching probability and skew for the destination address prefixes specified by filters in filter set acl5 in Figure 5.7. Note that there is considerably less branching at levels 2 through 11 in the destination address trie; however, there is considerably more branching at lower levels with a majority of the nodes at level 26 having two children. Likewise, the skew is high (when it is defined) at levels 0 through 23, but significantly decreases at levels 24 through 31. Thus



(a) Destination address branching probability; average per level.



(b) Destination address skew; average per level for nodes with two children.

Figure 5.7: Destination address branching probability and skew for filter set acl5.

destination address prefixes in acl5 will tend to be similar for the first 25 bits or so, then diverge. Plots of branching probability and skew for additional filter sets may be found in Appendix A.

Branching probability and skew characterize the structure of the individual source and destination address prefixes; however, it does not capture their interdependence. It is possible that some filters in a filter set match flows contained within a single subnet, while others match flows between different subnets. In order to capture this characteristic of a seed filter set, we measure the "correlation" of source and destination prefixes. In this context, we define correlation to be the probability that the source and destination address prefixes continue to be the same for a given prefix length.

This measure is only valid within the range of address bits specified by both address prefixes. While this measure is not particularly insightful for the purpose of crafting search algorithms, it does allow us to accurately model real filter sets.

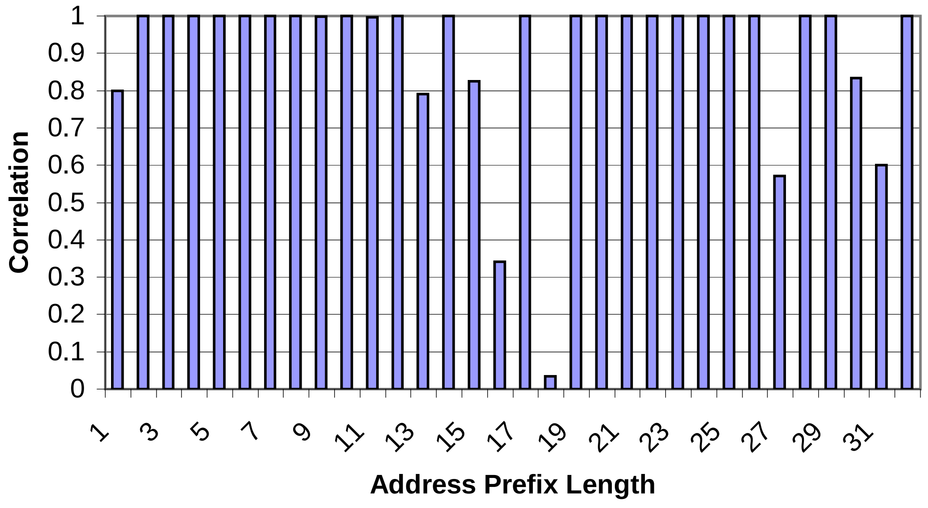
Consider the example of a filter set containing filters that all specify flows contained within the same class B network (16-bit network address); the correlation for levels 1 through 16 is 1.0, then falls off for levels 17 through 32 as the source and destination address prefixes diverge. From the seed filter set, we simply generate a probability distribution over the range of possible prefix lengths, $[1 \ldots 32]$. For the filter sets we studied, the address prefix correlation varies widely. The correlation for filter set acl5 is shown in Figure 5.8(a). Note that approximately 80% of the filters contain source and destination address prefixes with the same first bit. For those with the same first bit, they continue to be identical through the first 13 bits. Of those filters with source and destination address prefixes with the same initial 13 bits, approximately 80% of those continue to be correlated through bit 14, etc. Very few filters in acl5 contain address prefixes the remain correlated through bit 19. The correlation for filter set ipc1 is shown in Figure 5.8(b). Note that less than half of the filters contain source and destination address prefixes with the same first bit. Likewise, very few filters contain source and destination address prefixes that remain correlated through bit 26.

5.5 Scope

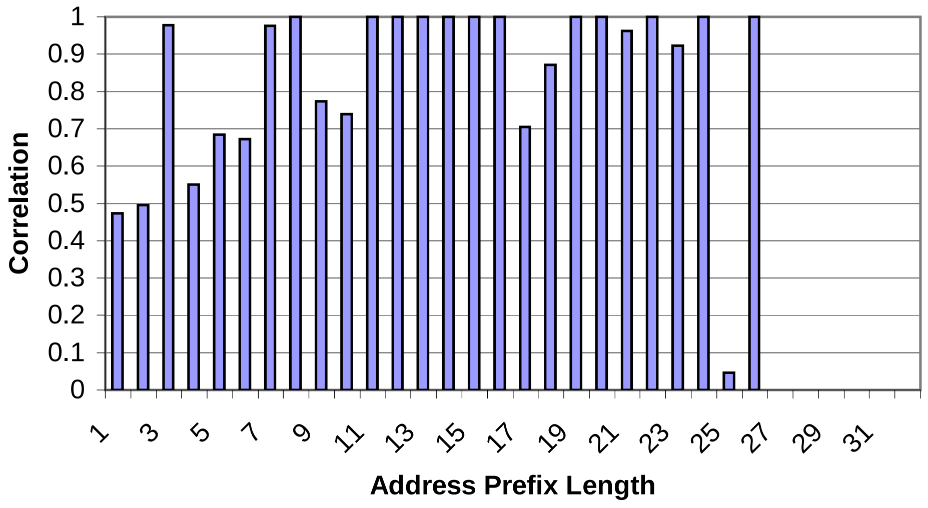
From a geometric perspective, a filter defines a region in d-dimensional space where d is the number of fields specified by the filter. The volume of that region is the product of the one-dimensional "lengths" specified by the filter. For example, length in the source address dimension corresponds to the number of addresses covered by the source address prefix of the filter. Likewise, length in the destination port dimension corresponds to the number of port numbers covered by the destination port range. Points in the d-dimensional space correspond to packet headers; hence, the geometric volume translates to the number of possible packet headers that match the filter. Instead of geometric lengths and volumes, we often refer to these filter properties in terms of a *tuple* specification. To be specific, we define the filter 5-tuple as a vector containing the following fields:

- $t[0]$, source address prefix length, $[0...32]$
- $t[1]$, destination address prefix length, $[0...32]$
- $t[2]$, source port range width, the number of port numbers covered by the range, $[0...2^{16}]$
- $t[3]$, destination port range width, the number of port numbers covered by the range, $[0...2^{16}]$
- $t[4]$, protocol specification, Boolean value denoting whether or not a protocol is specified, $[0, 1]$

The tuple essentially defines the *specificity* of the filter. Filters that are more specific cover a small set of possible packet headers while filters that are less specific cover a large set of possible packet



(a) acl5.



(b) ipc1

Figure 5.8: Address prefix correlation; probability that address prefixes of a filter continue to be the same at a given prefix length.

headers. To facilitate filter set measurement and modeling, we define a new metric, *scope*, to be the logarithmic measure of the number of possible packet headers covered by the filter. Using the 5-tuple definition above, we define *scope* for the 5-tuple as follows:

$$\begin{aligned} scope &= \lg\{(2^{32-t[0]}) \times (2^{32-t[1]}) \times t[2] \times t[3] \times (2^{8(1-t[4])})\} \\ &= (32 - t[0]) + (32 - t[1]) + (\lg t[2]) + (\lg t[2]) + 8(1 - t[4]) \end{aligned} \quad (5.2)$$

Table 5.6: 5-tuple scope measurements, average (μ_{scope}) and standard deviation (σ_{scope}).

Set	*Size*	μ_{scope}	σ_{scope}
acl1	733	25.0146	13.4585
acl2	623	51.6869	17.6880
acl3	2400	32.0168	15.6699
acl4	3061	30.9481	15.1367
acl5	4557	24.2274	8.0554
fw1	283	51.1686	15.6819
fw2	68	56.5842	23.0965
fw3	184	54.3004	14.8012
fw4	264	48.1127	27.9439
fw5	160	55.7881	16.9506
ipc1	1702	39.7172	19.4508
ipc2	192	47.0521	27.7966

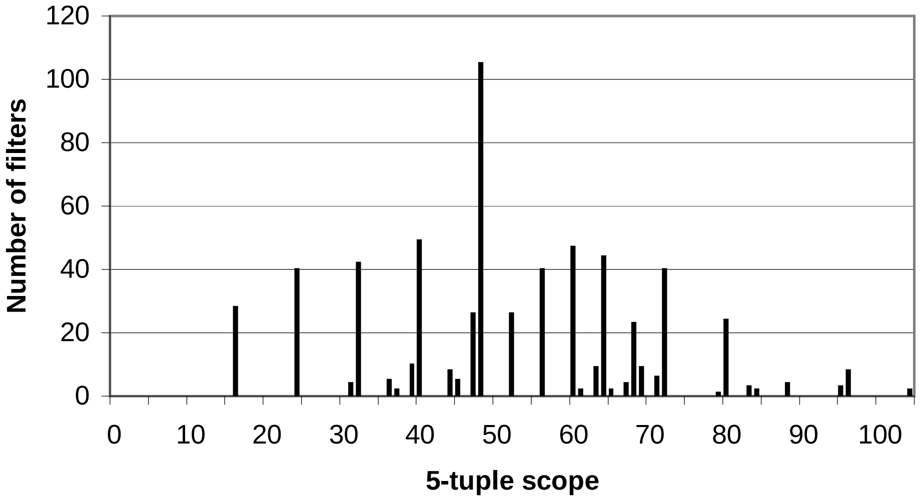
Scope translates the filter tuple into a measure of filter specificity on a scale from 0 to 104. Scope is isomorphic to the logarithm of the geometric volume specified by the filter.

The average 5-tuple scope and standard deviation for the 12 filter sets is shown in Table 5.6. The average 5-tuple scope ranges from 56 to 24. We note that filters in the ACL filter sets tend to have narrower scope, while filters in the FW filter sets tend to have wider scope. While the average scope of the filter sets does not vary drastically, the distributions of filter scope can exhibit drastically different characteristics. Figure A.6 shows the 5-tuple scope distribution of filter set acl2 and acl5. The filters in acl2 are distributed among scope values between 16 and 80 with the largest concentration at 48. The filters in acl5 are much more concentrated with most filter distributed among scope values between 16 and 32. The largest concentration is at scope 16. Additional 5-tuple scope distributions are provided in Appendix A.

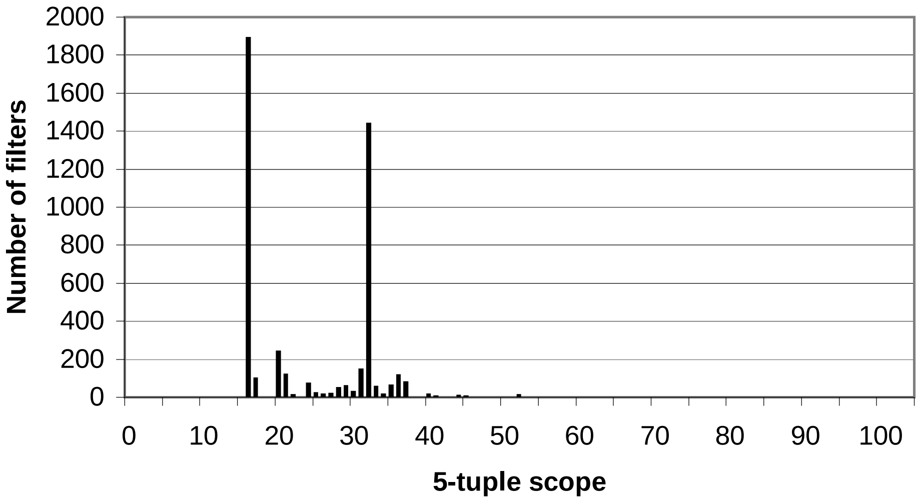
5.6 Filter Overlap

Many previous studies have shown that the maximum number of filters that match a packet is small for real filter sets, typically five to seven filters. Some recent studies have shown that the maximum number of filters that partially match a packet is also limited [77, 58]. For example, consider a filter set specifying 1000 filters on the standard 5-tuple. The number of filters that match the source and destination address of a packet may be 20 or less; thus, an effective way to narrow the scope of a search is to first perform a match on the address prefix pair. This is precisely the approach taken in *Extended Grid-of-Tries* (EGT) which we discussed in Section 4.3.2 [58].

The number of filters that match a packet for a partial or full match is often referred to as "filter overlap". This stems from the geometric view of filters where a packet defines a point in d-dimensional space and filters which match the packet define d-dimensional polygons which contain



(a) acl2, $\mu = 51.7$, $\sigma = 17.7$



(b) acl5, $\mu = 24.2$, $\sigma = 8.1$

Figure 5.9: Distribution of 5-tuple scope for filters in filter sets acl2 and acl5.

the point. Filters which cover a set of common points in the space are said to overlap. Algorithms such as EGT strongly rely on the filter overlap properties to hold. This is part of the reason that Baboescu and Varghese restrict their performance claims for EGT to filter sets *core routers*, as these filter sets tend to have few wildcards and limited prefix nesting. We performed filter overlap analyses in order to evaluate the efficacy of such approaches for our set of 12 real filter sets. In Table 5.7, we report the maximum number of filters matching a packet when matching on the individual 5-tuple fields (source address, destination address, etc.), the source/destination address prefix pair, the application specification, and all filter fields. The number of filters matching a packet when

Table 5.7: Maximum number of filters matching any packet; partial matches for each field in the 5-tuple, source/destination address prefix pair (SA-DA), and application specification (SP-DP-PR); full matches on all fields (All); matches; data from 12 real filter sets.

Set	*Size*	*SA*	*DA*	*SP*	*DP*	*PR*	*SA-DA*	*SP-DP-PR*	*All*
acl1	733	119	49	733	306	702	21	283	5
acl2	623	159	110	623	489	569	38	465	8
acl3	2400	323	235	2399	622	1678	44	412	7
acl4	3061	336	279	3060	743	2138	41	468	6
acl5	4557	309	354	4557	2344	1904	30	1303	2
fw1	283	192	107	245	117	165	43	62	5
fw2	68	19	43	38	68	41	9	26	4
fw3	184	140	92	156	66	106	50	26	4
fw4	264	172	116	169	89	184	61	43	5
fw5	160	113	84	131	72	86	42	36	5
ipc1	1702	257	398	1472	1105	1229	45	815	17
ipc2	192	121	36	172	172	122	10	122	3

classifying on a single field is high. Likewise, we find that the number of filters matching a packet when classifying on the address prefix pair is also high, up to 61 filters for filter set fw4. Clearly, techniques like *Extended Grid-of-Tries* (EGT) that depend on the limited filter overlap observed in core router filter sets do not perform well for other types of filter sets. We also conclude that performing a partial classification on the application specification is not helpful, as the number of matching filters is even higher than for the address prefix pair. Previous studies reported that the maximum number of filters matching a packet when classifying on all fields is typically less than five or six. Our results provide a few notable exceptions, as we observe three filter sets where the maximum numbers of matching filters is 7, 8, and 17.

5.7 Field Value Overlap

The observations made in previous studies and the model of filter construction discussed in Section 5.1 suggest that the number of unique filter field values and combinations of unique field values that match a packet may be inherently limited. Previous studies observed that filters share common field values; thus, the number of unique field values for a given filter field may be significantly less than the number of filters. Likewise, the number of unique field values that a packet matches must be less than or equal to the number of filters that match the packet when using the field in the match. For example, consider matching on the source address only. Assume there are 100 filters in the filter set; half of the filters specify the wildcard in the source address and half of the filters have a fully specified source address. The number of unique field values for the source address field is 51, the wildcard and 50 fully specified addresses. The number of overlapping filters for a packet will be 50 or 51 depending on whether or not the packet's source address is specified by a filter. The number of unique field values that match a packet will be 1 or 2 depending on whether or not the packet's source address is specified by a filter.

Table 5.8 shows the number of unique filter field values and combinations of field fields for the 12 real filter sets. Note that the number of unique fields are significantly less than the number of filters in the filter set. In several of the ACL filter sets, all source port fields are wildcarded, thus there is only one unique field value. We performed an exhaustive analysis of the maximum number of unique field values and unique combinations of field values which match any packet. A summary of the results for unique single fields, address prefix pairs, and application specifications are given in Table 5.9. Note that field overlap for address fields is more commonly referred to as prefix nesting. Another way to think about this measurement is that it specifies the maximum number of prefixes along any path from root to leaf in a trie defined by the unique address prefixes in the filter set. Also note that the number of unique field values is significantly less than the number of filters and the maximum number of unique field values matching any packet remains relatively constant for various filter set sizes. We also performed the same analysis for every possible combination of fields (every possible combination of two fields, three fields, etc.). There are

$$\sum_{i=1}^{d} \binom{d}{i} \tag{5.3}$$

unique combinations of d fields. For the standard 5-tuple, there are 31 unique combinations of fields. We observed that the maximum number of unique combinations of field values which match any packet is typically bounded by twice the maximum number of matching single field values, and also remains relatively constant for various filter set sizes. Finally, Table 5.9 also reports the maximum number of unique field combinations that match a packet when classifying on *All* fields. This is identical to the measurement of the maximum number of filters that match a packet.

5.8 Additional Fields

An examination of real filter sets reveals that additional fields beyond the standard 5-tuple are relevant. In 10 of the 12 filter sets that we studied, filters contain matches on TCP flags or ICMP type numbers. In most filter sets, a small percentage of the filters specify a non-wildcard value for the flags, typically less then two percent. There are notable exceptions, as approximately half the filters in filter set *ipc1* contain non-wildcard flags. We argue that new services and administrative policies will demand that packet classification techniques scale to support additional fields (i.e. more "dimensions") beyond the standard 5-tuple. It is not difficult to identify applications that could benefit from packet classification on fields in higher level protocol headers. Consider the following example: an ISP wants to deploy Voice over IP (VoIP) service running over an IPv6/UDP/RTP stack for new IP-enabled handsets and mobile appliances. The ISP also wants to make efficient use of expensive wireless links connecting Base Station Controllers (BSCs) to multiple Base Station

Table 5.8: Number of unique field values and combinations of field values specified by filters in 12 real filter sets.

Set	*Size*	*SA*	*DA*	*SP*	*DP*	*PR*	*Flag*	*SA-DA*	*SP-DP-PR*
acl1	733	97	205	1	108	4	3	426	112
acl2	623	182	207	1	27	5	6	527	37
acl3	2400	431	516	3	190	5	3	1588	202
acl4	3061	574	557	3	235	7	3	2065	250
acl5	4557	169	80	1	40	4	2	1873	42
fw1	283	57	66	13	43	5	11	128	612
fw2	68	31	21	9	1	5		50	14
fw3	184	31	28	9	39	4	11	61	52
fw4	264	30	43	28	49	9		79	82
fw5	160	38	35	11	33	4	11	72	46
ipc1	1702	152	128	34	54	7	11	941	96
ipc2	192	29	32	3	3	4	8	122	5

Table 5.9: Maximum number of unique field values and combinations of field values matching a packet; data from 12 real filter sets.

Set	*Size*	*SA*	*DA*	*SP*	*DP*	*PR*	*Flag*	*SA-DA*	*SP-DP-PR*	*All*
acl1	733	4	4	1	5	2	2	5	6	5
acl2	623	5	5	1	4	2	2	7	5	8
acl3	2400	6	4	2	6	2	2	7	7	7
acl4	3061	7	5	2	7	2	2	8	8	6
acl5	4557	3	2	1	4	1	2	3	3	2
fw1	283	4	4	3	3	2	2	7	4	5
fw2	68	3	3	2	1	2		4	3	4
fw3	184	4	3	3	3	2	2	7	4	4
fw4	264	3	4	4	3	2		6	4	5
fw5	160	5	4	3	3	2	2	7	4	5
ipc1	1702	4	5	4	5	2	2	10	8	17
ipc2	192	3	2	2	2	2	2	3	2	3

Transceivers (BSTs); hence, the ISP would like to use a header compression protocol like Robust Header Compression (ROHC). ROHC is a robust protocol that compresses packet headers for efficient use of wireless links with high loss rates [90]. In order to support this, the BSC must maintain a dynamic filter set which binds packets to ROHC contexts based on fields in the IPv6, UDP, and RTP headers. A total of seven header fields (352 bits) must be examined in order to classify such packets.

Matches on ICMP type number, RTP Synchronization Source Identifier (SSRC), and other higher-level header fields are likely to be exact matches; therefore, the number of unique field values matching any packet are at most two, an exact value and the wildcard if present. There may be other types of matches that more naturally suit the application, such as arbitrary bit masks on TCP flags; however, we do not foresee any reasons why the structure of filters with these additional fields will significantly deviate from the observed structure in current filter tables.

5.9 Impact of IPv6 Migration

While the current deployment of Internet Protocol Version 6 (IPv6) is extremely limited, most observers expect that migration to IPv6 from the current IPv4 protocol will eventually happen [9]. Currently, no filter sets containing rules with IPv6 addresses are available for study. A sense of how IPv6 addresses will be managed and what impact these practices may have on the statistical structure of filter sets may be garnered by examining IPv6 forwarding tables containing destination address prefixes. In order to assess the current state of IPv6 forwarding tables, five IPv6 BGP table snapshots were collected from several sites [91]. Figure 5.10 shows the combined distribution for a total of 1,550 entries. The individual tables are sufficiently small, so the combined distribution was chosen to reflect the overall trend. A significant feature is that the total number of unique prefix lengths in the combined distribution is 14. We now investigate IPv6 address architecture and deployment policies to gain a sense of whether or not the number of unique prefix lengths in forwarding tables and filter sets is expected to grow significantly. The number of unique prefix lengths is of greater interest than the distribution of prefixes among the prefix lengths for prefix matching techniques which are well-suited for IPv6 [24, 25]. Such techniques provide a strong foundation for developing packet classification techniques that perform well with IPv6 address fields.

5.9.1 Address Architecture

The addressing architecture for IPv6 is detailed in RFC 3513 [92]. In terms of the number of prefix lengths in forwarding tables, the important address type is the global unicast address which may be aggregated [93]. RFC 3513 states that IPv6 unicast addresses may be aggregated with arbitrary prefix lengths like IPv4 addresses under CIDR. While this provides extensive flexibility, we do not foresee that this flexibility necessarily results in an explosion of unique prefix lengths. The global unicast address format has three fields: a global routing prefix, a subnet ID, and an interface ID.

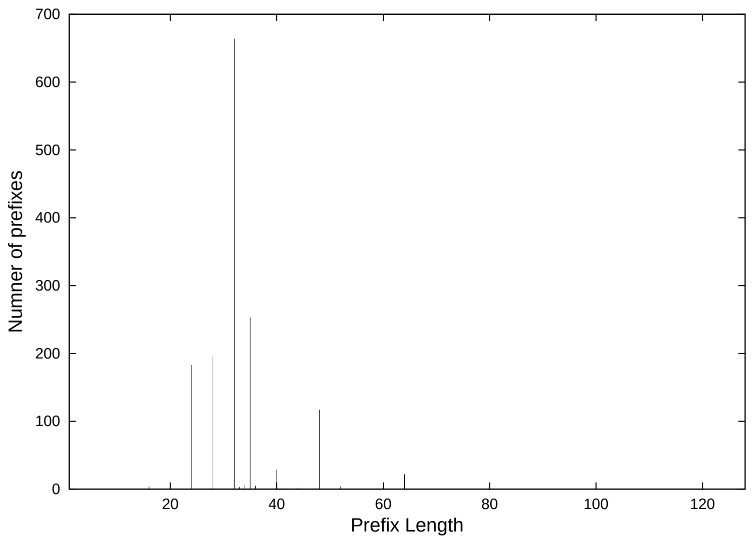


Figure 5.10: Combined prefix length distribution for IPv6 BGP route table snapshots.

All global unicast addresses, other than those that begin with 000, must have a 64-bit interface ID in the Modified EUI-64 format. These identifiers may be of global or local scope; however, we are only interested in the structure they impose on filter sets. In such cases, the global routing prefix and subnet ID fields must consume a total of 64 bits.

Global unicast addresses that begin with 000 do not have any restrictions on interface ID size; however, these addresses are intended for special purposes such as embedded IPv4 addresses. Embedded IPv4 addresses provide a mechanism for tunneling IPv6 packets over IPv4 routing infrastructure. We anticipate that this special class of global unicast addresses will not contribute many unique prefix lengths to IPv6 routing tables and will not affect the structure of current IPv4 filter sets.

5.9.2 Address Allocation & Assignment

[94] In a June 26, 2002 memo entitled, “IPv6 Address Allocation and Assignment Policy” the Internet Assigned Numbers Authority (IANA) announced initial policies governing the distribution or “licensing” of IPv6 address space [94]. One of its stated goals is to distribute address space in a hierarchical manner so as to “permit the aggregation of routing information by ISPs, and to limit the expansion of Internet routing tables”. To that end, the distribution process is also hierarchical. IANA has made initial distributions of /16 address blocks to existing Regional Internet Registries (RIRs). The RIRs are responsible for allocating address blocks to National Internet Registries (NIRs) and Local Internet Registries (LIRs). The LIRs and NIRs are responsible for assigning addresses and address blocks to end users and Internet Service Providers (ISPs).

The minimum allocation of address space to Internet Registries is in units of /32 blocks. LIRs must meet several criteria in order to receive an address allocation, including a plan to provide

IPv6 connectivity by assigning /48 address blocks. During the assignment process /64 blocks are assigned when only one subnet ID is required and /128 addresses when only one device interface is required. While it is not clear how much aggregation will occur due to ISPs assigning multiple /48 blocks, the allocation and assignment policy does provide significant structure. If these policies are followed, we anticipate that IPv6 routing tables and filter sets will not contain significantly more unique prefix lengths than current IPv4 tables. It is also likely that the number of prefixes matching a given IPv6 address will be equal or less than the number of prefixes matching a given IPv4 address.

Chapter 6

ClassBench: A Packet Classification Benchmark

The engineer is the key figure in the material progress of the world.
Sir Eric Ashby, Vice Chancellor of Cambridge University (1967-1969)

Due to the importance and complexity of the packet classification problem, a myriad of algorithms and resulting implementations exist. The performance and capacity of many algorithms and classification devices, including TCAMs, depend upon properties of the filter set and query patterns. Unlike microprocessors in the field of computer architecture, there are no standard performance evaluation tools or techniques available to evaluate packet classification algorithms and products. Network service providers are reluctant to distribute copies of real filter sets for security and confidentiality reasons, hence realistic test vectors are a scarce commodity. The small subset of the research community who obtain real filter sets either limit performance evaluation to the small sample space or employ ad hoc methods of modifying those filter sets. In response to this problem, we present *ClassBench*, a suite of tools for benchmarking packet classification algorithms and devices.

6.1 Motivation

Deployment of next generation network services hinges on the ability of Internet infrastructure to provide flow identification at physical link speeds. A packet classifier must compare header fields of every incoming packet against a set of filters in order to identify a flow. The resulting flow identifier is used to apply security policies, application processing, and quality-of-service guarantees to packets belonging to the specified flow. Typical packet classification filter sets have fewer than a thousand filters and reside in enterprise firewalls or edge routers. As network services and packet classifiers continue to migrate into the network core, it is anticipated that filter sets could swell to tens of thousands of filters or more. A more complete introduction to packet classification is provided in Chapter 1. The most common type of multiple field packet classification examines only the packet header fields comprising the 5-tuple, possibly due to the lack of fast and efficient solutions that scale with the number of search fields. As we discuss in Section 5.8, packet filters often examine fields beyond the standard IP 5-tuple and we anticipate that filter sets will continue to scale to larger numbers of fields. For this reason, we designed *ClassBench* with the capability of generating additional filter fields such as TCP flags and ICMP type numbers. While this is an important feature, the primary contribution of our work is the accurate modeling of the structure of the filter fields comprising the standard IP 5-tuple.

As reported in Chapter 5, it has been observed that real filter sets exhibit a considerable amount of structure. In response, several algorithmic techniques have been developed which exploit filter set structure to accelerate search time or reduce storage requirements [50, 51, 54, 58]. Consequently, the performance of these approaches are subject to the structure or statistical characteristics of the filter set. A more complete survey of multi-dimensional search algorithms and devices is provided in Chapter 4. As discussed in Section 4.2.2 and Section 5.3.3, the capacity and efficiency of the most prominent packet classification solution, Ternary Content Addressable Memories (TCAMs), is also subject to the characteristics of the filter set.

Despite the influence of filter set composition on the performance of packet classification search techniques and devices, no publicly available benchmarking tools, filter sets, or formal methodology exists for standardized performance evaluation. Due to security and confidentiality issues, access to large, real filter sets for analysis and performance measurements of new classification techniques has been limited to a small subset of the research community. Some researchers in academia have gained access to filter sets through confidentiality agreements, but are unable to distribute those filter sets. Furthermore, performance evaluations using real filter sets are restricted by the size and structure of the sample filter sets. Some researchers have proposed ad hoc methods, such as independently selecting address prefixes from backbone route tables, to construct synthetic filter sets or modify their composition. A survey of related work is provided in Section 6.2.

In order to facilitate future research and provide a foundation for a meaningful benchmark, we present *ClassBench*, a publicly available suite of tools for benchmarking packet classification algorithms and devices. As shown in Figure 6.1, *ClassBench* consists of three tools: a *Filter Set Analyzer*, *Filter Set Generator*, and *Trace Generator*. The general approach of *ClassBench* is to construct a set of benchmark *parameter files* that specify the relevant characteristics of real filter sets, generate a synthetic filter set from a chosen *parameter file* and a small set of high-level inputs, and also provide the option to generate a sequence of packet headers to probe the synthetic filter set using the *Trace Generator*. *Parameter files* contain various statistics and probability distributions that guide the generation of synthetic filter sets. The *Filter Set* analyzer tool extracts relevant statistics from a seed filter set, constructs probability distributions to guide the generation of synthetic filter sets, and generates a *parameter file*. This provides the capability to generate large synthetic filter sets which model the structure of a seed filter set. In Section 6.3 we discuss the statistics and probability distributions contained in the *parameter files* that drive the synthetic filter generation

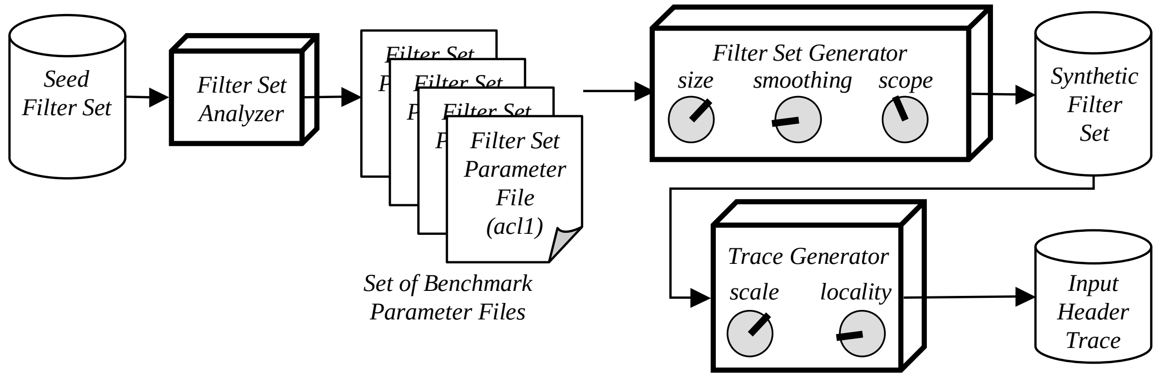


Figure 6.1: Block diagram of the *ClassBench* tools suite. The synthetic *Filter Set Generator* has size, smoothing, and scope adjustments which provide high-level, systematic mechanisms for altering the size and composition of synthetic filter sets. The set of benchmark *parameter files* model real filter sets and may be refined over time. The *Trace Generator* provides adjustments for trace size and locality of reference.

process. Selection of the relevant statistics and distributions is based on our study of 12 real filter sets presented in Chapter 5, and several iterations of the *Filter Set Generator* design. Note that *parameter files* may also be hand-constructed from qualitative characterizations of a specific filter set or class of filter sets such as backbone routers, edge routers, etc. We envision a set of benchmark *parameter files* that may be refined or expanded over time as the tools enjoy broader use.

The *Filter Set Generator* takes as input a *parameter file* and a few high-level parameters. Along with specifying filter set size, the tool provides mechanisms for systematically altering the composition of filters. Two adjustments, *smoothing* and *scope*, provide high-level control over filter set generation and an abstraction from the low-level statistics and distributions contained in the *parameter files*. The *smoothing* adjustment provides a structured mechanism for introducing new address aggregates which is useful for modeling filter sets significantly larger than the filter set used to generate the *parameter file*. The *scope* adjustment provides a biasing mechanism to favor more or less specific filters during the generation process. These adjustments and their affects on the resulting filter sets are discussed in Section 6.4.1 and Section 6.4.2. Finally, the *Trace Generator* tool examines the synthetic filter set, then generates a sequence of packet headers to exercise the filter set. Like the *Filter Set Generator*, the trace generator provides adjustments for scaling the size of the trace as well as the locality of reference of headers in the trace. These adjustments are described in detail in Section 6.5.

We highlight previous performance evaluation efforts by the research community as well as related benchmarking activity of the IETF in Section 6.2. It is our hope that this work initiates a broader discussion which will lead to refinement of the tools, compilation of a standard set of *parameter files*, and codification of a meaningful benchmark. Its value will depend on its perceived clarity and usefulness to the interested community. In the case of packet classification, this community is comprised of at least the following groups:

- *Researchers* seeking to evaluate new classification algorithms relative to alternative approaches and commercial products.
- *Classification product vendors* seeking to market their products with convincing performance claims over competing products.
- *Classification product customers* seeking to verify and compare classification product performance on a uniform scale. This group can be sub-divided into two major sub-groups: router vendors seeking to compare competing classification products during the design process and prior to selecting components, and router customers seeking to independently verify performance claims of router vendors based on the components used in the router.

6.2 Related Work

Extensive work has been done in developing benchmarks for many applications and data processing devices. Benchmarks are used extensively in the field of computer architecture to evaluate microprocessor performance. The effectiveness of these benchmarks to accurately distinguish the effects of architectural improvements, fabrication advances, and compiler optimizations is debatable; yet, there exists inherent value in providing a uniform scale for comparison.

In the field of computer communications, the Internet Engineering Task Force (IETF) has several working groups exploring network performance measurement. Specifically, the IP Performance Metrics (IPPM) working group was formed with the purpose of developing standard metrics for Internet data delivery [95]. The Benchmarking Methodology Working Group (BMWG) seeks to make measurement recommendations for various internetworking technologies [96][97]. These recommendations address metrics and performance characteristics as well as collection methodologies.

The BMWG specifically attacked the problem of measuring the performance of Forwarding Information Base (FIB) routers [98][99]. Realizing that router throughput, latency, and frame loss rate depend on the structure of the Forwarding Information Base (FIB) or route table, the BMWG prescribes a testing methodology with accompanying terminology. The recommendations describe testing at the router level, compounding the effects of system interfaces, control, and switching fabric. While the suggested tests take into consideration table size and prefix distribution, they lack specificity in how prefix distributions should be varied. The recommendations do introduce a methodology for determining the maximum FIB size and evaluating throughput relative to the table size. The BMWG also produced a methodology for benchmarking firewalls [100][101]. The methodology contains broad specifications such as: the firewall should contain at least one rule for each host, tests should be run with various filter set sizes, and test traffic should correspond to rules at the “end” of the filter set. This final specification provides for more accurate performance assessment of firewalls employing simple linear search algorithms. We assert that *ClassBench* complements efforts by the IETF by providing the necessary tools for generating test vectors with high-level control over filter set and input trace composition. The Network Processor Forum (NPF) has also initiated a benchmarking effort [102]. Currently, the NPF has produced benchmarks for switch fabrics and route lookup engines. To our knowledge, there are no current efforts by the IETF or the NPF to provide a benchmark for multiple field filter matching.

In the absence of publicly available packet filter sets, researchers have exerted much effort in order to generate realistic performance tests for new algorithms. Several research groups obtained access to real filter sets through confidentiality agreements. Gupta and McKeown obtained access to 40 real filter sets and extracted a number of useful statistics which have been widely cited [50]. Gupta and McKeown also generated synthetic two-dimensional filter sets consisting of source-destination address prefix pairs by randomly selecting address prefixes from publicly available route tables [51]. This technique was also employed by Feldman and Muthukrishnan [27]. Warkhede, Suri, and Varghese used this technique in a study of packet classification for two-dimensional "conflict-free" filters [68]. Baboescu and Varghese also generated synthetic two-dimensional filter sets by randomly selecting prefixes from publicly available route tables, but added refinements for controlling the number of zero-length prefixes (wildcards) and prefix nesting [54, 103]. A simple technique for appending randomly selected port ranges and protocols from real filter sets in order to generate synthetic five-dimensional filter sets is also described [54]. Baboescu and Varghese also introduced a simple scheme for using a sample filter set to generate a larger synthetic five-dimensional filter set [58]. This technique replicates filters by changing the IP prefixes while keeping the other fields unchanged. While these techniques address some aspects of scaling filter sets in size, they lack high-level mechanisms for adjusting filter set composition which is crucial for evaluating algorithms that exploit filter set characteristics.

Woo provided strong motivation for a packet classification benchmark and initiated the effort by providing an overview of filter characteristics for different environments (ISP Peering Router, ISP Core Router, Enterprise Edge Router, etc.) [29]. Based on high-level characteristics, Woo generated large synthetic filter sets, but provided few details about how the filter sets were constructed. The technique also does not provide controls for varying the composition of filters within the filter set. Nonetheless, his efforts provide a good starting point for constructing a benchmark capable of modeling various application environments for packet classification. Sahasranaman and Buddhikot used the characteristics compiled by Woo in a comparative evaluation of a few packet classification techniques [104].

6.3 Parameter Files

Our technique for generating synthetic filter sets with five or more fields addresses the issue of providing high-level control over the composition of synthetic filter sets and provides a more flexible foundation for a packet classification benchmark. Our technique uses real filter sets to generate *parameter files* which guide the *Filter Set Generator* and provide sufficient anonymity of addresses in the original filter set. We have generated a set of 12 *parameter files* which are publicly available along with the *ClassBench* tools suite. There still exists a need for a large sample space of real filter sets from various application environments in order to refine the *parameter files*. By reducing confidentiality concerns, we seek to remove the significant access barriers to realistic test vectors for researchers and promote the development of a meaningful benchmark.

Given a real filter set, the *Filter Set Analyzer* generates a *parameter file* that contains statistics and probability distributions that allow the *Filter Set Generator* to produce a synthetic filter set that retains the relevant characteristics of the real filter set. We chose the statistics and distributions to include in the *parameter file* based on thorough analysis of 12 real filter sets and several iterations of the *Filter Set Generator* design. Results of this analysis and a description of our metrics are provided in Chapter 5. We discuss the entries in the parameter file below. Where possible, we avoid discussing format details; interested readers and potential users of *ClassBench* may find a discussion of parameter file format in the documentation provided with the tools.

Protocols The *Filter Set Analyzer* generates a list of the unique protocol specifications and the distribution of filters over those values. We report the protocol distributions from 12 real filter sets and discuss observed trends in Section 5.3.1.

Port Pair Classes As we discussed in Section 5.3.3, we characterize the structure of source and destination port range pairs by defining a *Port Pair Class* (PPC). The *Filter Set Analyzer* generates a PPC distribution for each unique protocol specification in the filter set. This process can be thought of as follows: sort the filters into sets by protocol specification; for each set compute the PPC distribution and record it in the *parameter file*.

Flags For each unique protocol specification in the filter set, the *Filter Set Analyzer* generates a list of unique flag specifications and a distribution of filters over those values. As discussed in Section 5.8, 10 out of the 12 filter sets that we studied contain matches on TCP flags or ICMP type numbers.

Arbitrary Ranges As reported in Section 5.3.2, filter sets typically contain a small number of unique arbitrary range specifications. The *Filter Set Analyzer* generates a list of unique arbitrary range specifications and a distribution of filters over those values for both the source and destination port fields. Both distributions are recorded in the *parameter file*.

Exact Port Numbers As reported in Section 5.3.2, a significant number of filters specify exact port numbers in the source and destination port fields. Like the arbitrary range distributions, the

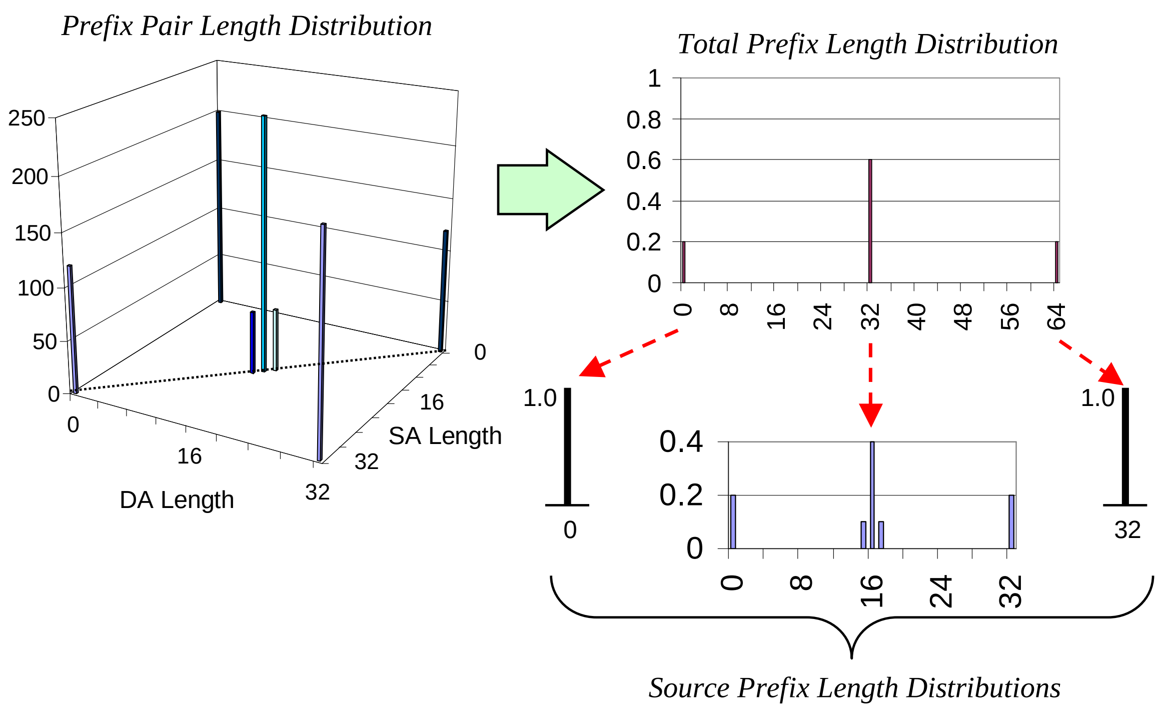


Figure 6.2: *Parameter files* represent prefix pair length distributions using a combination of a total prefix length distribution and source prefix length distributions for each non-zero total length.

Filter Set Analyzer generates a list of unique exact port specifications and a distribution of filters over those values for both the source and destination port fields. Both distributions are recorded in the *parameter file*.

Address Prefix Pair Lengths In Section 5.4 we demonstrated the importance of considering the prefix pair length as opposed to independent distributions for the source and destination address prefix lengths. *Parameter files* represent prefix pair length distributions using a combination of a total prefix length distribution and source prefix length distributions for each specified total length[1] as shown in Figure 6.2. The total prefix length is simply the sum of the prefix lengths for the source and destination address prefixes. As we will demonstrate in Section 6.4.2, modeling the total prefix length distribution allows us to easily bias the generation of more or less specific filters based on the *scope* input parameter. The source prefix length distributions associated with each specified total length allow us to model the prefix pair length distribution, as the destination prefix length is simply the difference of the total length and the source length.

Address Prefix Branching and Skew The branching probability and skew distributions defined in Section 5.4 allow us to model the address space coverage and relationships between address

[1]We do not need to store a source prefix distribution for total prefix lengths that are not specified by filters in the filter set.

prefixes specified in the filter set. The *Filter Set Analyzer* computes branching probability and skew distributions for both source and destination address prefixes. Both distributions are recorded in the *parameter file*.

Address Prefix Correlation The address correlation distribution defined in Section 5.4 specifies the relationship between source and destination address prefixes in each filter. The *Filter Set Analyzer* computes the address prefix correlation distribution and records it in the *parameter file*.

Prefix Nesting Thresholds As discussed in Section 5.7, the number of unique address prefixes that match a given packet is an important property of real filter sets and is often referred to as *prefix nesting*. We found that if the *Filter Set Generator* is ignorant of this property, it is likely to create filter sets with significantly higher prefix nesting, especially when the synthetic filter set is larger than the filter set used to generate the *parameter file*. Given that prefix nesting remains relatively constant for filter sets of various sizes, we place a limit on the prefix nesting during the filter generation process. The *Filter Set Analyzer* computes the maximum prefix nesting for both the source and destination address prefixes in the filter set and records these statistics in the *parameter file*. The *Filter Set Generator* retains these prefix nesting properties in the synthetic filter set, regardless of size. We discuss the process of generating address prefixes and retaining prefix nesting properties in Section 6.4.

Scale The *Filter Set Analyzer* also records the size of the real filter set in the generated *parameter file*. This statistic primarily serves as a reference point to users when selecting parameter files to use to test a particular device or algorithm. It is also used when the user chooses to scale the source and destination address branching probability and skew distributions with filter set size. This option is provided via a high-level command switch to the *Filter Set Generator*. For example, if a parameter file from a firewall filter set of 100 filters is used to generate a synthetic filter set of 10000 filters the user may want to allow the source and destination addresses to cover more of the IP address space while retaining the prefix nesting and prefix pair length distributions.

6.4 Synthetic Filter Set Generation

The *Filter Set Generator* is the cornerstone of the *ClassBench* tools suite. Perhaps the most succinct way to describe the synthetic filter set generation process is to walk through the pseudocode shown in Figure 6.3. The first step in the filter generation process is to read the statistics and distributions from the *parameter file*. Rather then list all of the distributions here, we will discuss them when they are used in the process. Next, we get the four high-level input parameters:

- *size*: target size for the synthetic filter set
- *smoothing*: controls the number of new address aggregates (prefix lengths)

- *port scope*: biases the tool to generate more or less specific port range pairs
- *address scope*: biases the tool to generate more or less specific address prefix pairs

We refer to the *size* parameter as a "target" size because the generated filter set may have fewer filters. This is due to the fact that it is possible for the *Filter Set Generator* to produce a filter set containing redundant filters, thus the final step in the process removes the redundant filters. The generation of redundant filters stems from the way the tool assigns source and destination address prefixes that preserve the properties specified in the *parameter file*. This process will be described in more detail in a moment.

Before we begin the generation process, we apply the *smoothing* adjustment to the prefix pair length distributions[2](lines 6 through 10). This adjustment provides a systematic, high-level mechanism for injecting new prefix lengths into the filter set while retaining the general characteristics specified in the *parameter file*. We discuss this adjustment and its effects on the generated filter set in Section 6.4.1. The *parameter file* specifies a prefix pair length distribution for each Port Pair Class. As described in Section 6.3, the *parameter file* represents each prefix pair length distribution as a total prefix length distribution with a source prefix length distribution for each specified total length. In order to apply the *smoothing* adjustment, we must iterate over all Port Pair Classes (line 7), apply the adjustment to each total prefix length distribution (line 8) and iterate over all total prefix lengths (line 9), and apply the adjustment to each source prefix length distribution associated with the total prefix length (line 10).

Prior to generating filters, we allocate a temporary array (line 11). The next set of steps (lines 12 through 27) generate a *partial* filter for each entry in the `Filters` array. Basically, we assign all filter fields except the address prefix values. Note that the prefix lengths for both source and destination address *are* assigned. The reason for this approach will become clear when we discuss the assignment of address prefix values in a moment. The first step in generating a *partial* filter is to select a protocol from the `Protocols` distribution specified by the *parameter file* (line 14). Note that this selection is performed with a uniform random variable, `rv` (line 13). We chose to select the protocol first because we found that the protocol specification dictates the structure of the other filter fields. Next, we select the protocol flags from the `Flags` distribution associated with the chosen protocol (line 16). The `Flags` distributions for all protocol specifications are given by the *parameter file*. Note that the protocol flags field is typically the wildcard unless the chosen protocol is TCP or ICMP. This selection is also performed with a uniform random variable (line 15).

After choosing the protocol and flags, we select a Port Pair Class, `PPC`, from the Port Pair Class matrix, `PPCMatrix`, associated with the chosen protocol (line 18). As discussed in Section 5.3.3, Port Pair Classes specify the type of port range specified by the source and destination port fields (wildcard, arbitrary range, etc.). Note that the selection of the `PPC` is performed with a random variable that is biased by the *port scope* parameter (line 17). This adjustment allows the user

[2]Note that the *scope* adjustments do not add any new prefi x lengths to the distributions. It only changes the likelihood that longer or shorter prefi x lengths in the distribution are chosen.

```
FilterSetGenerator()
// Read input file and parameters
read(parameter file)
get(size)
get(smoothing)
get(port scope)
get(address scope)
// Apply smoothing to prefix pair length distributions
If $smoothing > 0$
  For $i : 1$ to MaxPortPairClass
    TotalLengths[i]→smooth(smoothing)
    For $j : 0$ to 64
      SALengths[i][j]→smooth(smoothing)
// Allocate temporary filter array
FilterType Filters[size]
// Generate filters
For $i : 1$ to size
   // Choose an application specification
   rv = Random()
   Filters[i].Prot = Protocols→choose(rv)
   rv = Random()
   Filters[i].Flags = Flags[Filters[i].Prot]→choose(rv)
   rv = RandomBias(port scope)
   PPC = PPCMatrix[Filters[i].Prot]→choose(rv)
   rv = Random()
   Filters[i].SP = SrcPorts[PPC.SPClass]→choose(rv)
   rv = Random()
   Filters[i].DP = DstPorts[PPC.DPClass]→choose(rv)
   // Choose an address prefix length pair
   rv = RandomBias(address scope)
   TotalLength = TotalLengths[PPC]→choose(rv)
   rv = Random()
   Filters[i].SALength = SrcLengths[PPC][TotalLength]→choose(rv)
   Filters[i].DALength = TotalLength - Filters[i].SALength
// Assign address prefix pairs
AssignSA(Filters)
AssignDA(Filters)
// Remove redundant filters and prevent filter nesting
RemoveRedundantFilters(Filters)
OrderNestedFilters(Filters)
PrintFilters(Filters)
```

Figure 6.3: Pseudocode for *Filter Set Generator*.

to bias the *Filter Set Generator* to produce a filter set with more or less specific Port Pair Classes where WC-WC (both port ranges wildcarded) is the least specific and EM-EM (both port ranges specify an exact match port number) is the most specific. We discuss this adjustment and its effects on the generated filter set in Section 6.4.2. Given the Port Pair Class, we can select the source and destination port ranges from their respective port range distributions associated with each Port Class (lines 20 and 22). Note that the distributions for Port Classes WC, HI, and LO are trivial as they define single ranges; therefore, the *parameter file* only needs to specify arbitrary range (AR) and exact match (EM) port number distributions for both the source and destination ports. The selection of port ranges from a Port Class distribution is performed using a uniform random variable (lines 19 and 21).

Selecting the address prefix pair lengths is the last step in generating a *partial* filter. We select a total prefix pair length from the distribution associated with the chosen Port Pair Class (line 24) using a random variable biased by the *address scope* parameter (line 23). We discuss this adjustment and its effects on the generated filter set in Section 6.4.2. We select a source prefix length from the distribution associated with the chosen Port Pair Class and total length (line 26) using a uniform random variable (line 25). Note that we use an unbiased, uniform random variable for choosing the source address length. This allows us to retain the relationships between source and destination address prefix lengths. Finally, we calculate the destination address prefix length using the chosen total length and source address prefix length (line 27).

After we generate all the *partial* filters, we must assign the source and destination address prefix values. We begin by assigning the source address prefix values (line 28). The `AssignSA` routine constructs a binary trie using the set of source address prefix lengths in `Filters` and the source address branching probability and skew distributions specified by the *parameter file*. We start by allocating a root node, constructing a list of filters `FilterList` containing all the partial filters in `Filters`, and passing `FilterList` and a node pointer to a recursive process, `VisitNode`. This process first examines all of the entries in `FilterList`. If an entry has a source prefix length equal to the level of the node[3], it assigns the node's address to the entry and removes the entry from `FilterList`. Once completed, `VisitNode` recursively distributes the remaining filters to child nodes according to the branching probability and skew for the node's level. Note that we also keep track of the number of prefixes that have been assigned along a path by passing a `Nest` variable to the recursive process. If `Nest` $\geq$ `SANestThresh - 1`, where `SANestThresh` is the source prefix nesting threshold specified by the *parameter file*, then `VisitNode` ignores the branching probability and skew distributions. In this case, `VisitNode` partitions `FilterList` into two lists, one containing filters with source address prefix lengths equal to the next tree level, and one containing all the remaining filters. `VisitNode` then recursively passes the lists to two child nodes. In doing so, we ensure that the nesting threshold is not exceeded.

[3]Node level is synonymous with tree depth.

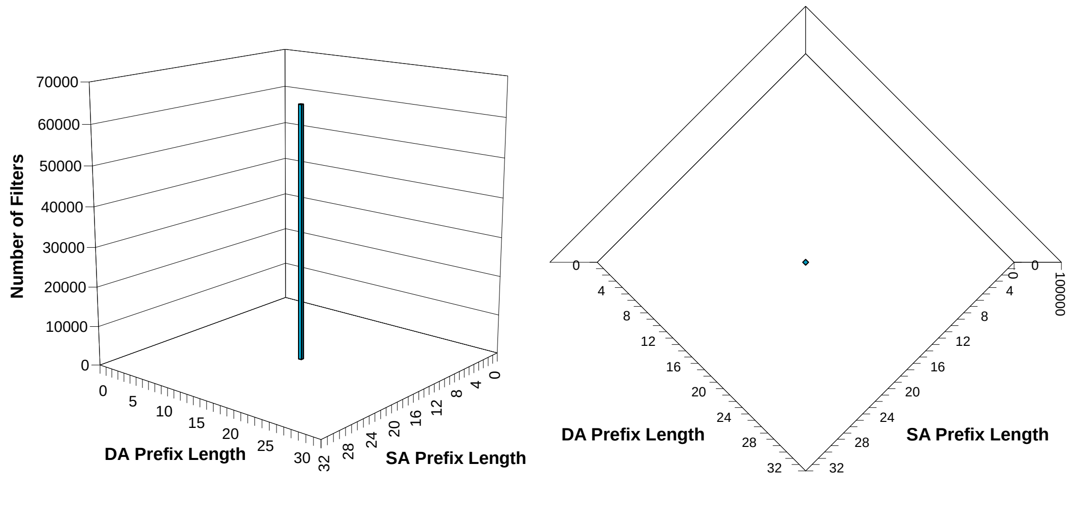
Assigning destination address prefix values is symmetric to the process for source address prefixes with one extension. In order to preserve the relationship between source and destination address prefixes in each filter, the `AssignDA` process (line 29) also considers the correlation distribution specified in the *parameter file*. In order to preserve the correlation, `AssignDA` employs a two-phase process of constructing the destination address trie. The first phase recursively distributes filters according to the correlation distribution. When the address prefixes of a particular filter cease to be correlated, it stores the filter in a temporary `StubList` associated with the current tree node. The second phase recursively walks down the tree and completes the assignment process in the same manner as the `AssignSA` process, with the exception that the `StubList` is appended to the `FilterList` passed to the `AssignDA` process prior to processing.

Note that we do not explicitly prevent the *Filter Set Generator* from generating redundant filters. Identical *partial* filters may be assigned the same source and destination address prefix values by the `AssignSA` and `AssignDA` functions. In essence, this preserves the characteristics specified by the *parameter file* because the number of unique filter field values allowed by the various distributions is inherently limited. Consider the example of attempting to generate a large filter set using a *parameter file* from a small filter set. If we are forced to generate the number of filters specified by the *size* parameter, we face two unfavorable results: (1) the resulting filter set may not model the *parameter file* because we are repeatedly forced to choose values from the tails of the distributions in order to create unique filters, or (2) the *Filter Set Generator* never terminates because it has exhausted the distributions and cannot create any more unique filters. With the current design of the *Filter Set Generator*, a user can produce a larger filter set by simply increasing the *size* target beyond the desired size. While this does introduce some variability in the size of the synthetic filter set, we believe this is a tolerable trade-off to make for maintaining the characteristics in the *parameter file* and achieving reasonable execution times for the *Filter Set Generator*.

Thus, after generating a list of *size* synthetic filters, we remove any redundant filters from the list via the `RemoveRedundantFilters` function (line 30). A naïve implementation of this function would require $O(N^2)$ time, where N is equal to $size$. We discuss an efficient mechanism for removing redundant filters from the set in Section 6.4.3. After removing redundant filters from the filter set, we sort the filters in order of increasing scope (line 31). This allows the filter set to be searched using a simple linear search technique, as nested filters will be searched in order of decreasing specificity. An efficient technique for performing this sorting step is also discussed in Section 6.4.3. Finally, we print the filter set to an output file (line 32). The following subsections provide detailed descriptions and analyses of the smoothing and scope adjustments, as well as efficient techniques for removing redundant filters and sorting the filters to prevent nesting.

6.4.1 Smoothing Adjustment

As filter sets scale in size, we anticipate that new address prefix pair lengths will emerge due to network address aggregation and segregation. In order to model this behavior, we provide for the



(a) $r = 0$ (b) $r = 0$, top view

Figure 6.4: Prefix pair length distribution for a synthetic filter set of 64000 filters generated with a *parameter file* specifying 16-bit prefix lengths for all addresses.

introduction of new prefix lengths in a structured manner. Injecting purely random address prefix pair lengths during the generation process neglects the structure of the filter set used to generate the *parameter file*. Using scope as a measure of distance, we expect that new address aggregates will emerge “near” an existing address aggregate. Consider the address prefix pair length distribution shown in Figure 6.4. In this example, all filters in the filter set have 16-bit source and destination address prefixes; thus, the distribution is a single “spike”. When injecting new address prefix pair lengths into the distribution, we would like them to be clustered around the existing spike in the distribution. This structured approach translates “spikes” in the distribution into smoother “hills”; hence, we refer to the process as smoothing.

In order to control the injection of new prefix lengths, we define a *smoothing* parameter which limits the maximum radius of deviation from the original prefix pair length, where radius is measured in the number of bits specified by the prefix pair. Geometrically, this measurement may be viewed as the Manhattan distance from one prefix pair length to another. For convenience, let the *smoothing* parameter be equal to r. We chose to model the clustering using a symmetric binomial distribution. Given the parameter r, a symmetric binomial distribution is defined on the range $[0 : 2r]$, and the probability at each point i in the range is given by:

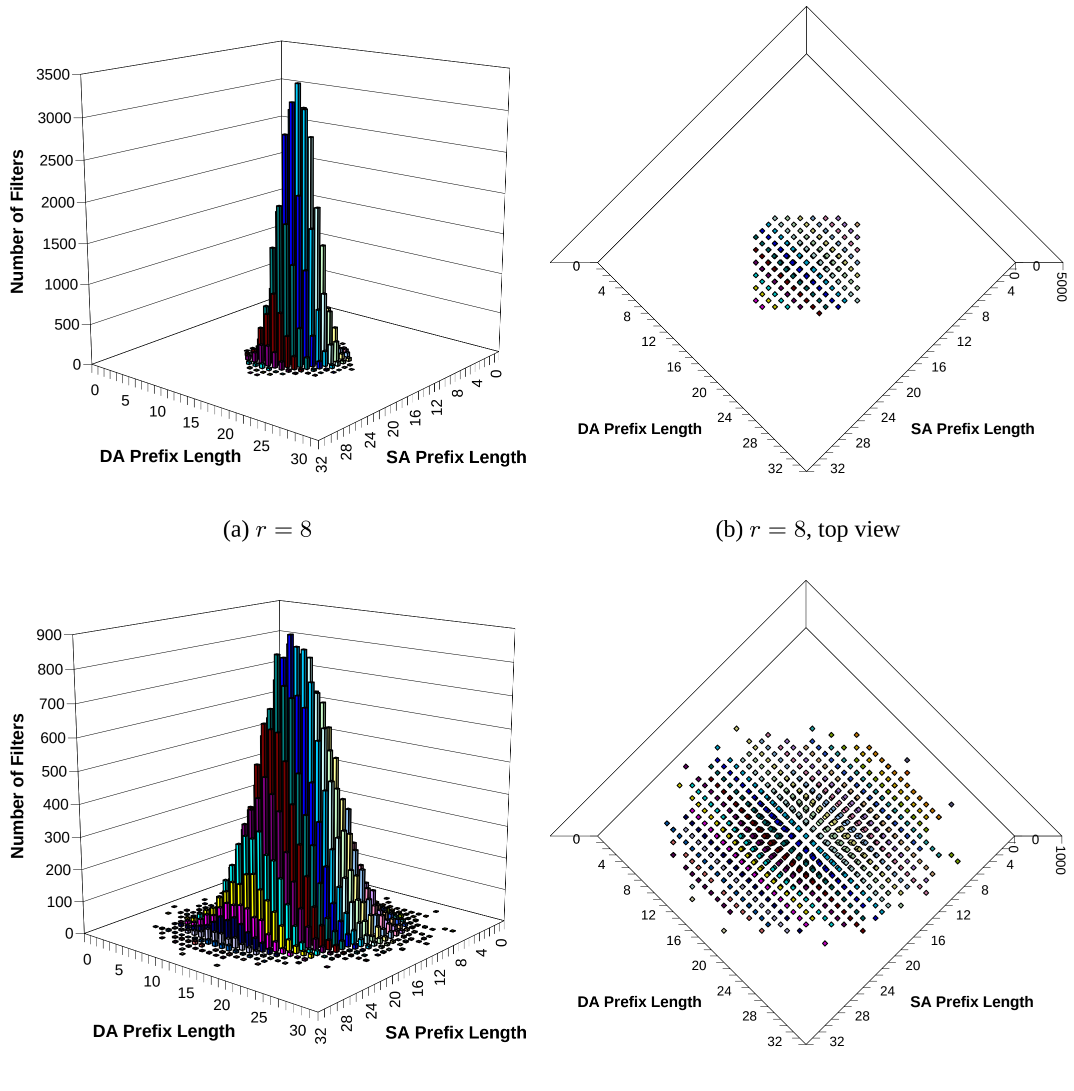
$$p_i = \binom{2r}{i} \left(\frac{1}{2}\right)^{2r} \tag{6.1}$$

Note that r is the median point in the range with probability p_r, and r may assume values in the range $[0 : 64]$.

Once we generate the symmetric binomial distribution from the *smoothing* parameter, we apply this distribution to each specified prefix pair length. The smoothing process involves scaling each “spike” in the distribution according to the median probability p_r, and binomially distributing the residue to the prefix pair lengths within the r-bit radius. When prefix lengths are at the “edges” of the distribution, we simply truncate the binomial distribution. This requires us to normalize the prefix pair length distribution as the last step in the smoothing process. Note that we must apply the smoothing adjustment to each prefix pair length distribution associated with each Port Pair Class in the *parameter file*. In order to demonstrate this process, we provide an example of smoothing the prefix pair length distribution in Figure 6.4 using two different values of r. Figure 6.5(a) and Figure 6.5(b) show the prefix pair length distributions for a synthetic filter set generated with a *parameter file* specifying 16-bit prefix lengths for all addresses and a smoothing parameter $r = 8$. With the exception of the fringe effects due to random number generation, the single spike at 16-16 is binomially distributed to the prefix pair lengths within a Manhattan distance of 8. The same effect is shown in Figure 6.5(a) and Figure 6.5(b) for a smoothing parameter $r = 32$.

In practice, we expect that the *smoothing* parameter will be limited to at most 8. In order to demonstrate the effect of smoothing in a realistic context, we generated a synthetic filter set using a *smoothing* parameter of 4. Figure 6.6(a) and Figure 6.6(b) show the prefix pair length distribution for a synthetic filter set of 64000 filters generated using the ipc1 *parameter file* and smoothing parameter $r = 0$. Figure 6.6(c) and Figure 6.6(d) show the prefix pair length distribution for a synthetic filter set of 64000 filters generated using the ipc1 *parameter file* and smoothing parameter $r = 4$. Note that this synthetic filter set retains the structure of the original filter set while modeling a realistic amount of address aggregation and segregation.

Recall that we choose to truncate and normalize to deal with the edge cases. As evident in Figure 6.6, many of the most common address prefix pair lengths occur at the edges of the distribution. As a result, applying the smoothing adjustment may affect the average scope of the generated filter set. Consider the case of the spike at 32-32 (fully specified source and destination addresses). Applying the smoothing adjustment to this point distributes some of the residue to less specific prefix pair lengths, but the residue allocated to more specific prefix pair lengths is truncated as there are not any more specific prefix pair lengths. In order to assess the effects of truncation and normalization on the resulting filter sets, we generated several filter sets of the same size using three different *parameter files* and various values of the smoothing parameter. The results are shown in Figure 6.4.1. Note that as we increase the amount of smoothing applied to the prefix pair length distributions, the effect on the 5-tuple scope and address pair scope is minimal. We observe a slight drift toward the median scope value due to the aforementioned truncation of the distributions at the edges.



(a) $r = 8$ (b) $r = 8$, top view

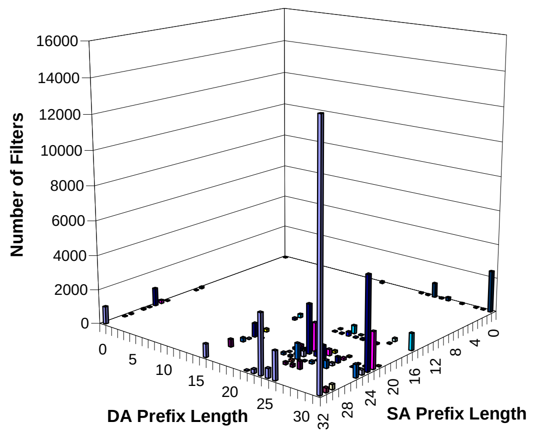
(c) $r = 32$ (d) $r = 32$, top view

Figure 6.5: Prefix pair length distributions for a synthetic filter set of 64000 filters generated with a *parameter file* specifying 16-bit prefix lengths for all addresses and various values of smoothing parameter r.

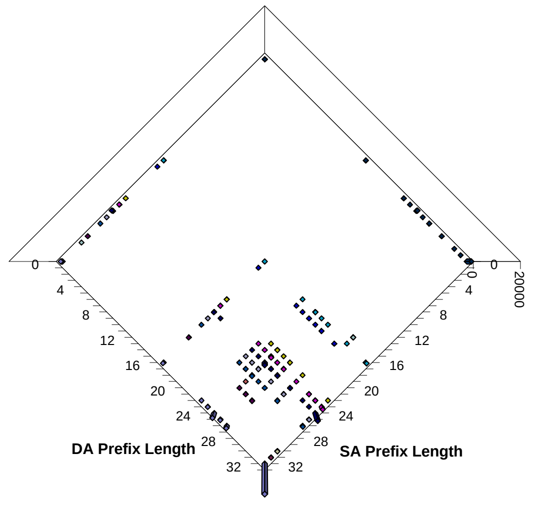
6.4.2 Scope Adjustment

As filter sets scale in size and new applications emerge, it is likely that the average scope of the filter set will change. As the number of flow-specific filters in a filter sets increases, the specificity of the filter set increases and the average scope decreases. If the number of explicitly blocked ports for all packets in a firewall filter set increases, then the specificity of the filter set may decrease and the average scope may increase[4]. In order to explore the effect of filter scope on the performance

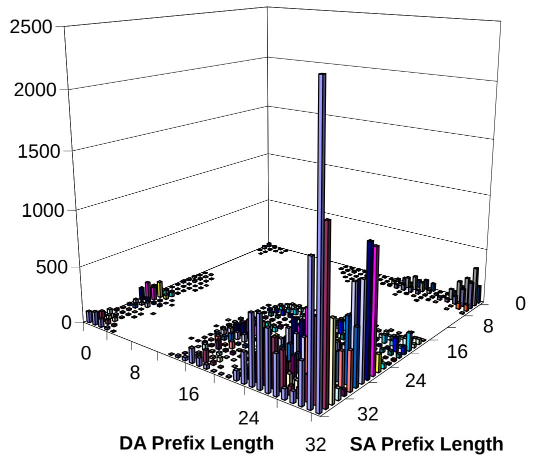
[4]We are assuming a common practice of specifying an exact match on the blocked port number and wildcards for all other fi lter fi elds



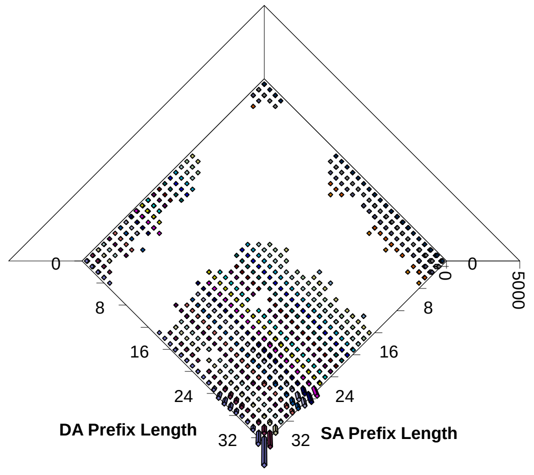
(a) $r = 0$



(b) $r = 0$, top view



(c) $r = 4$

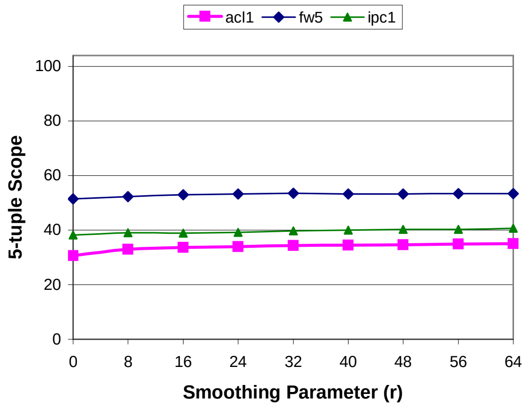


(d) $r = 4$, top view

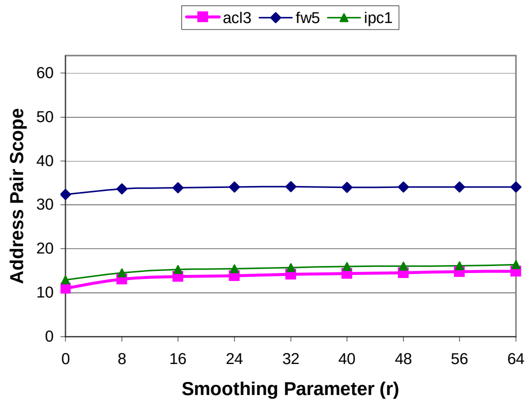
Figure 6.6: Prefix pair length distribution for a synthetic filter set of 64000 filters generated with the ipc1 *parameter file* with smoothing parameters $r = 0$ and $r = 4$.

of algorithms and packet classification devices, we provide high-level adjustments of the average scope of the synthetic filter set. Two input parameters, *address scope* and *port scope*, allow the user to bias the *Filter Set Generator* to create more or less specific address prefix pairs and port pairs, respectively.

In order to illustrate the effects of scope adjustments, consider the standard method of sampling from a distribution using a uniformly distributed random variable. In Figure 6.8, we show the cumulative distribution for the total prefix pair length associated with the WC-WC port pair class of the acl2 filter set. In order to sample from this distribution, the *Filter Set Generator* selects



(a) 5-tuple Scope



(b) Address Prefi x Pair Scope

Figure 6.7: Average scope of synthetic filter sets consisting of 16000 filters generated with parameter files extracted from filter sets *acl3*, *fw5*, and *ipc1*, and various values of the smoothing parameter r.

a random number between zero and one using a uniform random number generator, then chooses the total prefix pair length covering that number in the cumulative distribution. Graphically, this amounts to projecting a horizontal line from the random number on the y-axis. The x-coordinate of the “step” which it intersects is the sampled total prefix pair length. In Figure 6.8, we shown an example of sampling with a random variable equal to 0.5 to choose the total prefix pair length of 44.

The *address scope* adjustment essentially biases the sampling process to select more or less specific total prefix pair lengths. We can realize this in two ways: (1) apply the adjustment to the cumulative distribution, or (2) bias the random variable used to sample from the cumulative distribution. The first option requires that we recompute the cumulative density distribution to make longer or shorter prefix lengths more or less probable, as dictated by the *address scope* parameter. The second option provides a conceptually simpler alternative. Returning to the example in Figure 6.8, if we want to bias the *Filter Set Generator* to produce more specific address prefix pairs, then we want the random variable used to sample from the distribution to be biased to values closer to 1. The reverse is true if we want less specific address prefix pairs. Thus, in order to apply the scope adjustment we simply use a random number generator to choose a uniformly distributed random variable, rv_{uni}, apply a biasing function to generate a biased random variable, rv_{bias}, and sample from the cumulative distribution using rv_{bias}.

While there are many possible biasing functions, we limit ourselves to a particularly simple class of functions. Our chosen biasing function may be viewed as applying a slope, s, to the uniform distribution as shown in Figure 6.9(a). When the slope $s = 0$, the distribution is uniform. The biased random variable corresponding to a uniform random variable on the x-axis is equal to the area of the

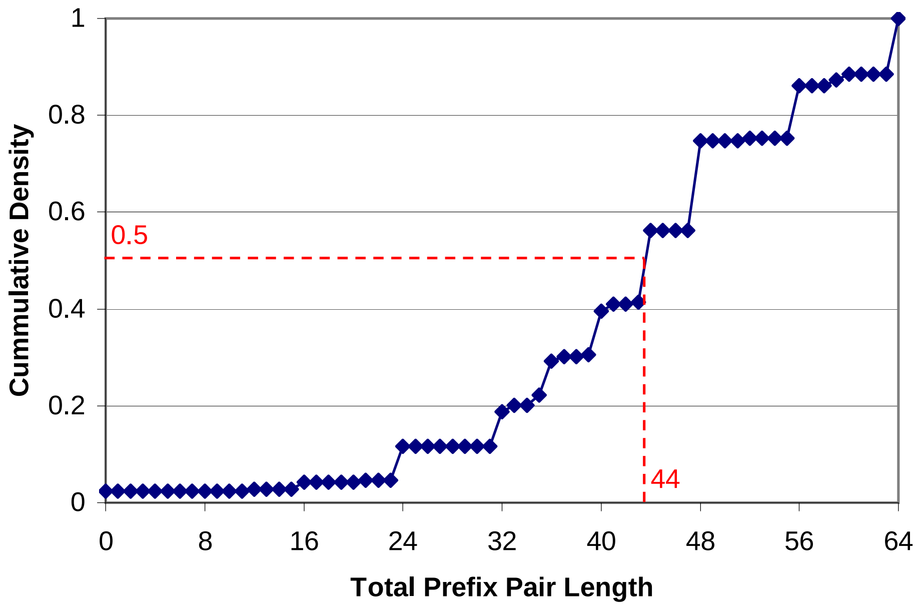


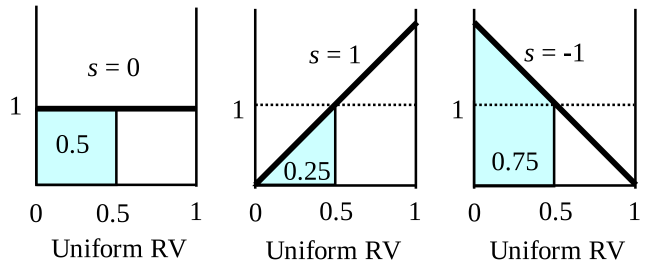
Figure 6.8: Example of sampling from a cumulative distribution using a random variable. Distribution is for the total prefix pair length associated with the WC-WC port pair class of the acl2 filter set. A random variable equal to 0.5 chooses 44 as the total prefix pair length.

rectangle defined by the value and a line intersecting the y-axis at one with a slope of zero. Thus, the biased random variable is equal to the uniform random variable. As shown in Figure 6.9(a), we can bias the random variable by altering the slope of the line. Note that in order for the biasing function to be defined for random variables in the range $[0 : 1]$ and have a cumulative probability of 1 for a random variable equal to 1, the slope adjustment must be in the range $[-2 : 2]$. Graphically, this results in the line pivoting about the point $(0.5, 1)$. For convenience, we define the scope adjustments to be in the range $[-1 : 1]$, thus the slope is equal to two times the scope adjustment. For non-zero slope values, the biased random variable corresponding to a uniform random variable on the x-axis is equal to the area of the trapezoid[5] defined by the value and a line intersecting the point $(0.5, 1)$ with a slope of s. The expression for the biased random variable, rv_{bias}, given a uniform random variable, rv_{uni}, and a *scope* parameter in the range $[-1 : 1]$ is:

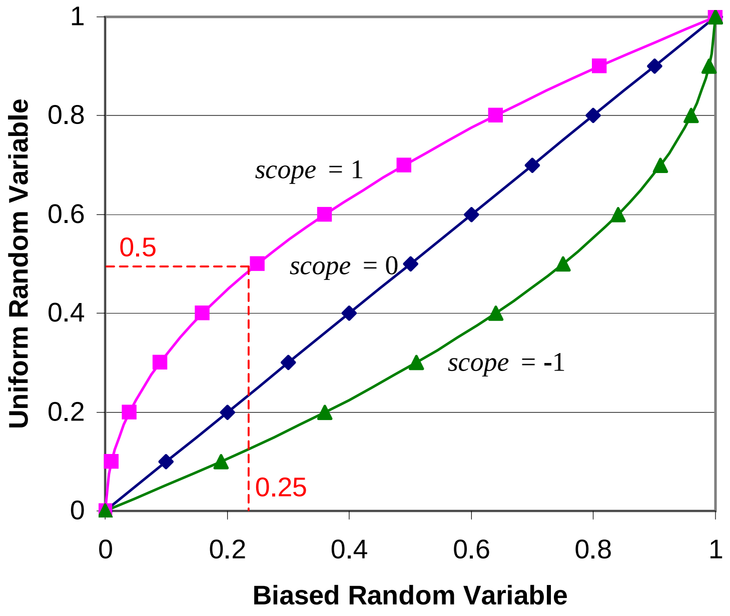
$$rv_{bias} = rv_{uni}(scope \times rv_{uni} - scope + 1) \tag{6.2}$$

Figure 6.9(b) shows a plot of the biasing function for *scope* values of 0, -1, and 1. We also provide a graphical example of computing the biased random variable given a uniform random variable of 0.5 and a *scope* parameter of 1. In this case the rv_{bias} is 0.25. Let us return to the example of choosing the total address prefix length from the cumulative distribution. In Figure 6.10, we show examples of sampling the distribution using the unbiased uniform random variable, $rv_{uni} = 0.5$,

[5]Recall that the area of a trapezoid is one half the product of the height and the sum of the lengths of the parallel edges, $A = \frac{1}{2} \times h \times (l_1 + l_2)$.



(a) Biased random variable is defined by area under line with slope $s = 2 \times scope$.



(b) Plot of scope biasing function.

Figure 6.9: Scope applies a biasing function to a uniform random variable.

and the biased random variable, $rv_{bias} = 0.25$, resulting from applying the biasing function with $scope = 1$. Note that the biasing results in the selection of a less specific address prefix pair, a total length of 35 as opposed to 44.

Positive values of *address scope* bias the *Filter Set Generator* to choose less specific address prefix pairs, thus increasing the average scope of the filter set. Likewise, negative values of *address scope* bias the *Filter Set Generator* to choose more specific address prefix pairs, thus decreasing the average scope of the filter set. The same effects are realized by the *port scope* adjustment by biasing the *Filter Set Generator* to select more or less specific port range pairs. Note that the cumulative distribution must be constructed in such a way that the distribution is computed over values sorted from least specific to most specific.

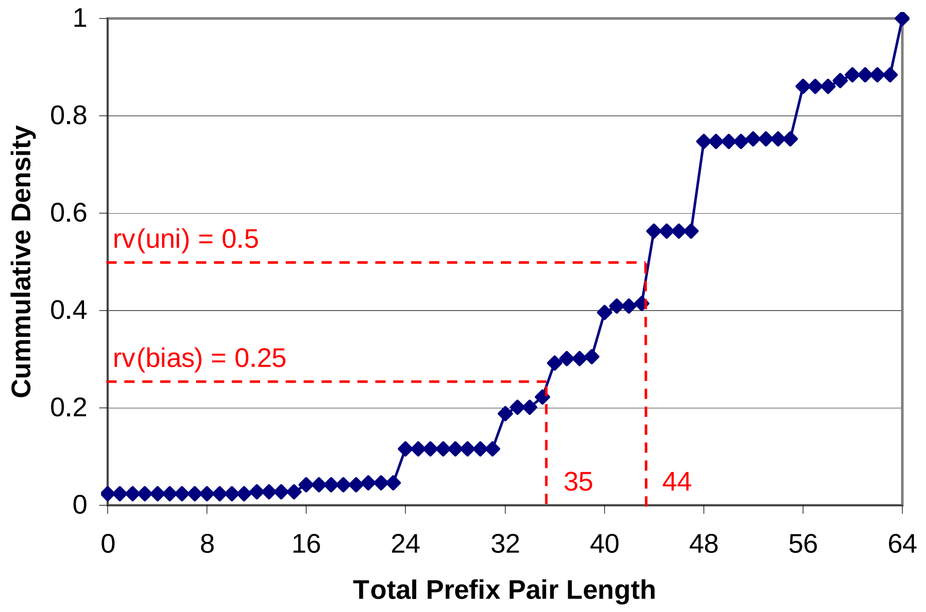
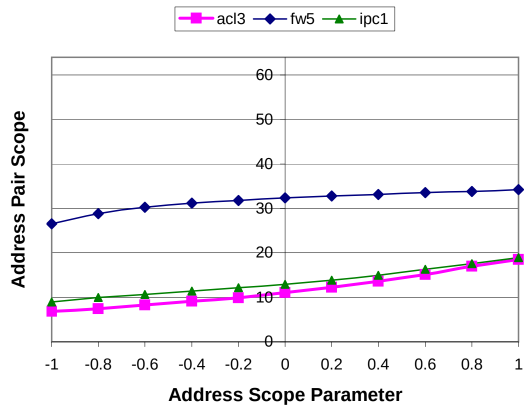


Figure 6.10: Example of sampling from a cumulative distribution using a random variable. Distribution is for the total prefix pair length associated with the WC-WC port pair class of the acl2 filter set. A random variable equal to 0.5 chooses 44 as the total prefix pair length.

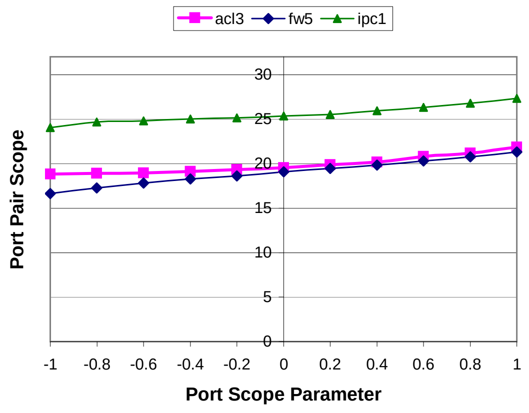
Finally, we report the results of tests assessing the effects of the *address scope* and *port scope* parameters on the synthetic filter sets generated by the *Filter Set Generator*. Each data point in the plots in Figure 6.4.2 is from a synthetic filter set containing 16000 filters generated from a *parameter file* from filter sets acl3, fw5, or ipc1. Figure 6.11(a) shows the effect of the *address scope* parameter on the average scope of the address prefix pairs in the resulting filter set. Over its range of values, the *address scope* alters the average address pair scope by ± 4 to ± 6. Figure 6.11(b) shows the effect of the *port scope* parameter on the average scope of the port range pairs in the resulting filter set. Over its range of values, the *port scope* alters the average port pair scope by ± 1.5 to ± 2.5. Note that the magnitude of change in average scope for both parameters is approximately the same relative to the range of possible scope values. Figure 6.11(c) shows the effect of both scope parameters on the average scope of the filters in the resulting filter set. For these tests, both scope parameters were set to the same value. Over their range of values, the scope parameters alter the average filter scope by ± 6 to ± 7.5. We assert that these scope adjustments provide a convenient high-level mechanism for exploring the effects of filter specificity on the performance of packet classification algorithms and devices.

6.4.3 Filter Redundancy & Priority

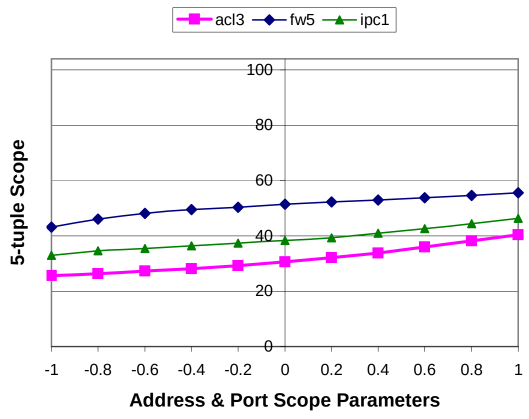
The final steps in synthetic filter set generation are removing redundant filters and ordering the remaining filters in order of increasing scope. The removal of redundant filters may be realized by simply comparing each filter against all other filters in the set; however, this naïve implementation



(a) Effect of *address scope* adjustment on the address prefi x pair scope



(b) Effect of *port scope* adjustment on the port pair scope



(c) 5-d Scope

Figure 6.11: Average scope of synthetic filter sets consisting of 16000 filters generated with parameter files extracted from filter sets *acl3*, *fw5*, and *ipc1*, and various values of the scope parameters.

requires $O(N^2)$ time, where N is equal to $size$. Such an approach makes execution times of the *Filter Set Generator* prohibitively long for filter sets in excess of a few thousand filters. In order to accelerate this process, we first sort the filters into sets according to their tuple specification. Sorting filters into tuple sets was introduced by Srinivasan, et. al. in the context of the *Tuple Space Search* packet classification algorithm discussed in Section 4.5 [66].

We perform this sorting efficiently by constructing a binary search tree of tuple set pointers, using the scope of the tuple as the key for the node. When adding a filter to a tuple set, we search the set for redundant filters. If no redundant filters exist in the set, then we add the filter to the set. If a redundant filter exists in the set, we discard the filter. The time complexity of this search technique depends on the number of tuples created by filters in the filter set and the distribution of filters across the tuples. In practice, we find that this technique provides acceptable performance.

Generating a synthetic filter set of 10k filters requires approximately five seconds, while a filter set of 100k filters requires approximately five minutes with a Sun Ultra 10 workstation.

In order to support the traditional linear search technique, filter priority is often inferred by placement in an ordered list. In such cases, the first matching filter is the best matching filter. This arrangement could obviate a filter f_i if a less specific filter $f_j \supset f_i$ occupies a higher position in the list. To prevent this, we order the filters in the synthetic filter set according to scope, where filters with minimum scope occur first. The binary search tree of tuple set pointers makes this ordering task simple. Recall that we use scope as the node key. Thus, we simply perform an in-order walk of the binary search tree, appending the filters in each tuple set to the output list of filters.

6.5 Trace Generation

When benchmarking a particular packet classification algorithm or device, many of the metrics of interest such as storage efficiency and maximum decision tree depth may be garnered using the synthetic filter sets generated by the *Filter Set Generator*. In order to evaluate the throughput of techniques employing caching or the power consumption of various devices under load, we must exercise the algorithm or device using a sequence of synthetic packet headers. The *Trace Generator* produces a list of synthetic packet headers that probe filters in a given filter set. Note that we do not want to generate random packet headers. Rather, we want to ensure that a packet header is covered by at least one filter in the *FilterSet* in order to exercise the packet classifier and avoid default filter matches. We experimented with a number of techniques to generate synthetic headers. One possibility is to compute all the d-dimensional polyhedra defined by the intersections of the filters in the filter set, then choose a point in the d-dimensional space covered by the polyhedra. The point defines a packet header. The best-matching filter for the packet header is simply the highest priority filter associated with the polyhedra. If we generate at least one header corresponding to each polyhedra, we fully exercise the filter set. The number of polyhedra defined by filter intersections grows exponentially, and thus fully exercising the filter set quickly becomes intractable. As a result, we chose a method that partially exercises the filter set and allows the user to vary the size and composition of the headers in the trace using high-level input parameters. These parameters control the scale of the header trace relative to the filter set, as well as the locality of reference in the sequence of headers. As we did with the *Filter Set Generator*, we discuss the *Trace Generator* using the pseudocode shown in Figure 6.12.

We begin by reading the *FilterSet* from an input file (line 1). Next, we get the input parameters *scale*, *ParetoA*, and *ParetoB* (lines 2 through 4). The *scale* parameter is used to set a threshold for the size of the list of headers relative to the size of the *FilterSet* (line 5). In this context, *scale* specifies the ratio of the number of headers in the trace to the number of filters in the filter set. After computing the `Threshold`, we allocate a list of headers, `Headers` (line 6). The next set

```
TraceGenerator()
   // Generate list of synthetic packet headers
1  read(FilterSet)
2  get(scale)
3  get(ParetoA)
4  get(ParetoB)
5  Threshold = $scale \times$ size(FilterSet)
6  HeaderList Headers()
7  While size(Headers) < Threshold
8        RandFilt = randint(0,size(FilterSet))
9        NewHeader = RandomCorner(RandFilt,FilterSet)
10       Copies = Pareto(ParetoA,ParetoB)
11       For $i : 1$ to Copies
12           Headers→append(NewHeader)
13 Headers→print
```

Figure 6.12: Pseudocode for *Trace Generator*.

of steps continue to generate synthetic headers as long as the size of `Headers` does not exceed the `Threshold`.

Each iteration of the header generation loop begins by selecting a random filter in the *FilterSet* (line 8). Next, we must choose a packet header covered by the filter. In the interest of exercising priority resolution mechanisms and providing conservative performance estimates for algorithms relying on filter overlap properties, we would like to choose headers matching a large number of filters. In the course of our analyses, we found the number of overlapping filters is large for packet headers representing the “corners” of filters. When we view a filter as defining a d-dimensional rectangle, the corners of this rectangle represent points in the d-dimensional space which correspond to packet headers. Each field of a filter covers a range of values. Choosing a packet header corresponding to a “corner” translates to choosing a value for each header field from one of the extrema of the range specified by each filter field. The `RandomCorner` function chooses a random “corner” of the filter identified by `RandFilt` and stores the header in `NewHeader`.

The last steps in the header generation loop append a variable number of copies of `NewHeader` to the trace. The number of copies, `Copies`, is chosen by sampling from a Pareto distribution controlled by the input parameters, *ParetoA* and *ParetoB* (line 10). In doing so, we provide a simple control point for the locality of reference in the header trace. The Pareto distribution[6] is one of the heavy-tailed distributions commonly used to model the burst size of Internet traffic flows as well as the file size distribution for traffic using the TCP protocol [105]. For convenience, let $a = ParetoA$ and $b = ParetoB$. The probability density function for the Pareto distribution may be expressed

[6]The Pareto distribution, a power law distribution named after the Italian economist Vilfredo Pareto, is also known as the Bradford distribution.

as:

$$P(x) = \frac{ab^a}{x^{a+1}} \tag{6.3}$$

where the cumulative distribution is:

$$D(x) = 1 - \left(\frac{b}{x}\right)^a \tag{6.4}$$

The Pareto distribution has a mean of:

$$\mu = \frac{ab}{a-1} \tag{6.5}$$

Expressed in this way, a is typically called the shape parameter and b is typically called the scale parameter, as the distribution is defined on values in the interval (b, ∞). The following are some examples of how the Pareto parameters are used to control locality of reference:

- Low locality of reference, short tail: ($a = 10$, $b = 1$) most headers will be inserted once
- Low locality of reference, long tail: ($a = 1$, $b = 1$) many headers will be inserted once, but some could be inserted over 20 times
- High locality of reference, short tail: ($a = 10$, $b = 4$) most headers will be inserted four times

Once the size of the trace exceeds the threshold, the header generation loop terminates. Note that a large burst near the end of the process will cause the trace to be larger than `Threshold`. After generating the list of headers, we write the trace to an output file (line 13).

6.6 Benchmarking with ClassBench

In order to provide value to the interested community, a packet classification benchmark must provide meaningful measurements that cover the broad spectrum of application environments. It is with this in mind that we designed the suite of *ClassBench* tools to be flexible while hiding the low-level details of filter set structure. While it is unclear if real filter sets will vary as specified by the smoothing and scope parameters, we assert that the tool provides a useful mechanism for measuring the effects of filter set composition on classifier performance. It is our hope that *ClassBench* will enjoy broader use by researchers in need of realistic test vectors; it is also our intention to initiate and frame a broader discussion within the community that results in a larger set of *parameter files* that model real filter sets as well as the formulation of a standard benchmarking methodology.

Packet classification algorithms and devices range from purely conceptual, to software implementations targeted to a variety of platforms, to state-of-the-art ASICs (Application Specific Integrated Circuits). For the purpose of our discussion, we present a generic packet classifier model as shown in Figure 6.13. In this model, the classifier consists of a search engine connected to memory which stores the filter set and any other data structures required for the search. For each packet header passed to the classifier, the search engine queries the filter set and returns an associated flow

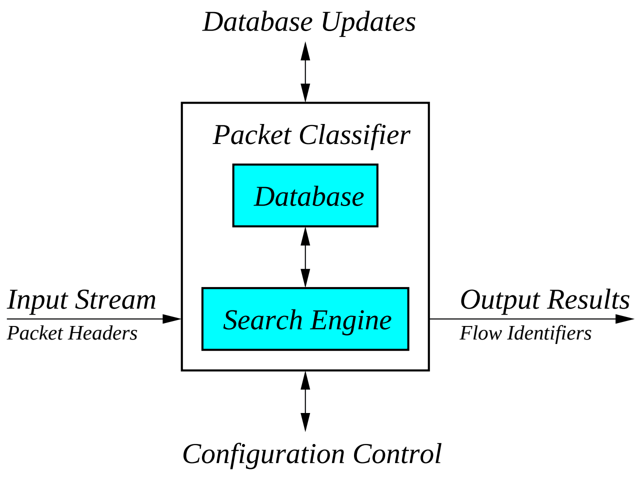


Figure 6.13: Generic model of a packet classifier.

identifier or set of flow identifiers. Note that the set of possible flow identifiers is application dependent. Firewalls may only specify two types of flows, admissible and inadmissible, whereas routers implementing per-flow queuing may specify thousands of unique flow identifiers. The configuration control is used to specify parameters such as the number of matching flow identifiers to return and the format of incoming packet headers. In order to model application environments where per-flow filters are dynamically created and deleted, the model includes a mechanism for dynamic filter set updates.

There are three primary metrics of interest for packet classification algorithms and devices: lookup throughput, memory requirements, and power consumption. Update performance is also a consideration, but secondary to the other three metrics. For packet classification devices or fixed implementations of algorithms, throughput can be directly measured using a synthetic filter set and associated header trace. Throughput measurements for software implementations of algorithms are not as straight-forward. In this case, the metric most directly influencing throughput is the required number of *sequential* memory accesses. Using parallel and pipelined design techniques, non-sequential memory accesses can be masked. A suitable benchmarking methodology should report both the total and sequential memory accesses in terms of average, worst observed, and best observed. The second metric of vital interest is the amount of memory required to store the filter set and supplemental data structures. For classification techniques employing random access memory, garnering memory usage metrics is straight-forward using a synthetic filter set. For TCAM-based devices, memory usage can be measured in terms of storage efficiency, which is defined to be the ratio of the number of required TCAM slots and the number of filters in the filter set. The *Filter Set Generator* allows us to analyze the effect of filter set size, scope, and smoothness on throughput and memory usage can be measured.

In the past, power consumption has not been a primary concern for those developing new packet classification techniques. As discussed in Section 4.2.2, TCAM-based classifiers have become the most popular solution for high performance routers, but they suffer from high power consumption. A typical TCAM consumes more than 100 times the power of state-of-the-art SRAMs and can account for a large fraction of the power budget on a router interface card. Recent developments in TCAM technology provide for partitioning the device such that only a subset of the available slots are activated at one time. IP lookup and packet classification techniques can take advantage of this capability to lower power consumption [106, 32]. The effect of filter set size, scope, and smoothness on standard TCAMs and algorithms employing partitioning in order to lower power consumption can be measured using the *Filter Set Generator*.

The *Trace Generator* is useful for evaluating algorithms and devices under realistic operating conditions. By providing control over the locality of reference in the sequence of packet header queries, we also provide a convenient tool for measuring the performance of packet classifiers employing caching.

With the desire to refine the *ClassBench* tools suite and formalize a benchmarking methodology, we seek to initiate a broader discussion and solicit input from the community to help guide the remainder of this work. To facilitate this discussion, we make the tools publicly available at the following site: `http://www.arl.wustl.edu/˜det3/ClassBench/`. Input garnered from the community will be used to refine the tools suite, assemble a standard set of *parameter files*, and formally specify a benchmarking methodology. While we have already found *ClassBench* to be very useful in our own research, it is our hope to promote its broader use in the research community.

Chapter 7

Scalable Packet Classification using Distributed Crossproducting of Field Labels

Follow the path of the unsafe, independent thinker. Expose your ideas to the dangers of controversy.
Thomas J. Watson, IBM

Due to the complexity of the search, packet classification is often a performance bottleneck in network infrastructure; therefore, it has received much attention in the research community and a wide variety of algorithms and devices exist in the research literature and commercial market. The existing solutions explore various design tradeoffs to provide high search rates, power and space efficiency, fast incremental updates, and the ability to scale to large numbers of filters. There remains a need for techniques that achieve a favorable balance among these tradeoffs and scale to support classification on additional fields beyond the standard 5-tuple. We introduce *Distributed Crossproducting of Field Labels* (*DCFL*), a novel combination of new and existing packet classification techniques that leverages key observations of the structure of real filter sets and takes advantage of the capabilities of modern hardware technology. Using a collection of 12 real filter sets and the *ClassBench* tools suite, we provide analyses of *DCFL* performance and resource requirements on filter sets of various sizes and compositions in Section 7.7. Based on these results, we show that an optimized implementation of *DCFL* can provide over 100 million searches per second and storage for over 200 thousand filters with current generation hardware technology. In Section 7.8, we discuss algorithms related to our approach and highlight the distinctions and advantages of *DCFL* relative to the state-of-the-art.

7.1 Description of DCFL

Distributed Crossproducting of Field Labels (*DCFL*) is a novel combination of new and existing packet classification techniques that leverages key observations of filter set structure and takes advantage of the capabilities of modern hardware technology. We discuss the observed structure of real filter sets in detail and provide motivation for packet classification on larger numbers of fields in Chapter 5. Two key observations motivate our approach: the number of unique field values for a given field in the filter set is small relative to the number of filters in the filter set, and the number of unique field values matched by any packet is very small relative to the number of filters in the filter set. We also draw from the encoding ideas highlighted in Section 4.2 in order to efficiently store the filter set and intermediate search results.

Using a high degree of parallelism, *DCFL* employs optimized search engines for each filter field and an efficient technique for aggregating the results of each field search. By performing this aggregation in a distributed fashion, we avoid the exponential increase in the time or space incurred when performing this operation in a single step. Given that search techniques for single packet fields are well-studied, the primary focus of this chapter is the development and analysis of an aggregation mechanism that can make use of the embedded multi-port memory blocks in the current generation of ASICs and FPGAs. We introduce several new concepts including field labeling, *Meta-Labeling* unique field combinations, *Field Splitting*, and optimized data structures such as *Bloom Filter Arrays* that minimize the number of memory accesses to perform set membership queries. As a result, our technique provides fast lookup performance, efficient use of memory, support for dynamic updates at high rates, and scalability to filters with additional fields.

DCFL may be described at a high-level using the following notation:

- Partition the filters in the filter set into fields
- Partition each packet header into corresponding fields
- Let F_i be the set of unique field values for filter field i that appear in one or more filters in the filter set
- Let $F_i(x) \subseteq F_i$ be the subset of filter field values in F_i matched by a packet with the value x in header field i
- Let $F_{i,j}$ be the set of unique filter field value pairs for fields i and j in the filter set; i.e. if $(u, v) \in F_{i,j}$ there is some filter or filters in the set with u in field i and v in field j
- Let $F_{i,j}(x, y) \subseteq F_{i,j}$ be the subset of filter field value pairs in $F_{i,j}$ matched by a packet with the value x in header field i and y in header field j
- This can be extended to higher-order combinations, such as set $F_{i,j,k}$ and subset $F_{i,j,k}(x, y, z)$, etc.

The *DCFL* method can be structured in many different ways. In order to illustrate the lookup process, assume that we are performing packet classification on four fields and a header arrives with field values $\{w, x, y, z\}$. One possible configuration of a *DCFL* search is shown in Figure 7.1 and proceeds as follows:

- In parallel, find subsets $F_1(w)$, $F_2(x)$, $F_3(y)$, and $F_4(z)$
- In parallel, find subsets $F_{1,2}(w, x)$ and $F_{3,4}(y, z)$ as follows:
  - Let $F_{query}(w, x)$ be the set of possible field value pairs formed from the crossproduct of $F_1(w)$ and $F_2(x)$
  - For each field value pair in $F_{query}(w, x)$, query for set membership in $F_{1,2}$, if the field value pair is in set $F_{1,2}$ add it to set $F_{1,2}(w, x)$
  - Perform the symmetric operations to find subset $F_{3,4}(y, z)$
- Find subset $F_{1,2,3,4}(w, x, y, z)$ by querying set $F_{1,2,3,4}$ with the field value combinations formed from the crossproduct of $F_{1,2}(w, x)$ and $F_{3,4}(y, z)$
- Select the highest priority exclusive filter and r highest priority non-exclusive filters in $F_{1,2,3,4}(w, x, y, z)$

Note that there are several variants which are not covered by this example. For instance, we could alter the aggregation process to find the subset $F_{1,2,3}(w, x, y)$ by querying $F_{1,2,3}$ using the crossproduct of $F_{1,2}(w, x)$ and $F_3(y)$. We can then find the subset $F_{1,2,3,4}(w, x, y, z)$ by querying $F_{1,2,3,4}$ using the crossproduct of $F_{1,2,3}(w, x, y)$ and $F_4(z)$. A primary focus of this chapter is determining subsets ($F_{1,2}(w, x)$, $F_{3,4}(y, z)$, etc.) via optimized set membership data structures.

As shown in Figure 7.1, *DCFL* employs three major components: a set of parallel search engines, an aggregation network, and a priority resolution stage. Each search engine F_i independently searches for all filter fields matching the given header field using an algorithm or architecture optimized for the type of search. For example, the search engines for the IP address fields may employ compressed multi-bit tries while the search engine for the protocol and flag fields use simple hash tables. We provide a brief overview of options for performing the independent searches on packet fields in Section 7.5. As previously discussed in Chapter 5 and shown in Table 5.9, each set of matching labels for each header field is typically less than five for real filter tables. The sets of matching labels generated by each search engine are fed to the aggregation network which computes the set of all matching filters for the given packet in a multi-stage, distributed fashion. Finally, the priority resolution stage selects the highest priority exclusive filter and the r highest priority non-exclusive filters. The priority resolution stage may be realized by a number of efficient algorithms and logic circuits; hence, we do not discuss it further.

The first key concept in *DCFL* is labeling unique field values with locally unique labels; thus, sets of matching field values can be represented as sets of labels. Table 7.1 shows the sets of

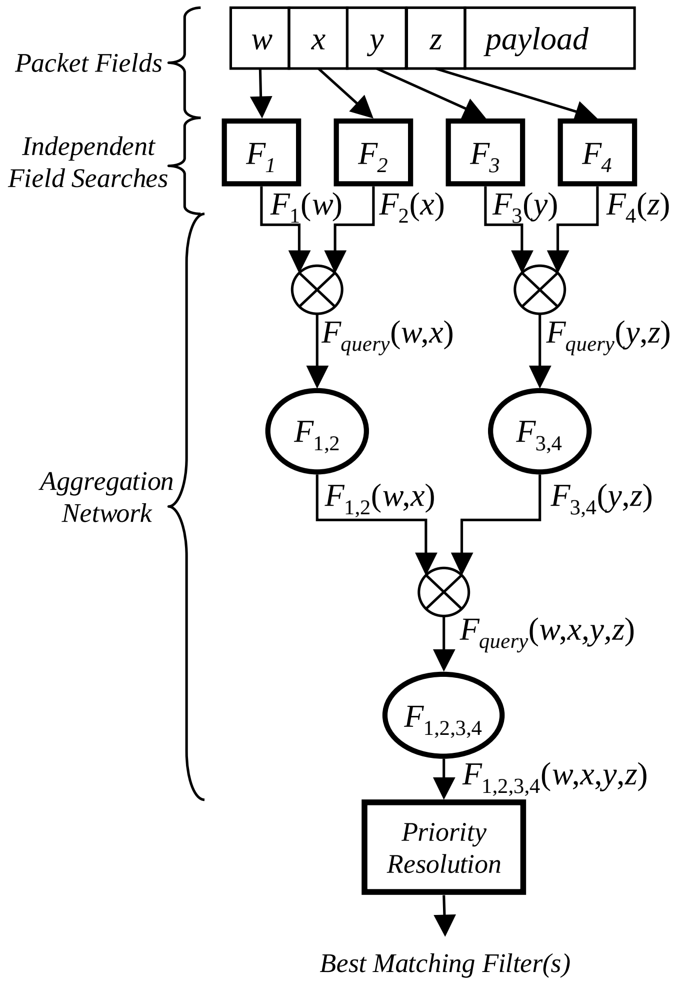


Figure 7.1: Example configuration of *Distributed Crossproducting of Field Labels* (*DCFL*); field search engines operate in parallel and may be locally optimized; aggregation nodes also operate in parallel; aggregation network may be constructed in a variety of ways.

unique source and destination addresses specified by the filters in Table 1.1. Note that each unique field value also has an associated "count" value which records the number of filters which specify the field value. The "count" value is used to support dynamic updates; a data structure in a field search engine or aggregation node only needs to be updated when the "count" value changes from 0 to 1 or 1 to 0. We identify unique combinations of field values by assigning either (1) a composite label formed by concatenating the labels for each field value in the combination, or (2) a new *meta-label* which uniquely identifies the combination in the set of unique combinations[1]. *Meta-Labeling* essentially compresses the size of the label used to uniquely identify the field combination. In addition to reducing the memory requirements for explicitly storing composite labels, this optimization has another subtle benefit. *Meta-Labeling* compresses the space addressed by the label, thus the

[1]Meta-labeling can be thought of as simply numbering the set of unique fi eld combinations

Table 7.1: Sets of unique specifications for each field in the sample filter set.

SA	*Label*	*Count*
11010010	0	1
10011100	1	1
101101*	2	1
10011100	3	2
*	4	2
100111*	5	2
10010011	6	1
11101100	7	1
111010*	8	1
100110*	9	1
010110*	10	1
01110010	11	2

DA	*Label*	*Count*
*	0	7
001110*	1	1
01101010	2	2
011010*	3	2
01111010	4	1
01011000	5	1
11011000	6	2

PR	*Label*	*Count*
TCP	0	4
*	1	5
UDP	2	6
ICMP	3	1

DP	*Label*	*Count*
[3:15]	0	5
[1:1]	1	2
[0:15]	2	5
[5:5]	3	1
[6:6]	4	1
[0:1]	5	1
[3:3]	6	1

meta-label may be used as an index into a set membership data structure. The use of labels allows us to use set membership data structures that only store labels corresponding to field values and combinations of field values present in the filter table. While storage requirements depend on the structure of the filter set, they scale linearly with the number of filters in the database. Furthermore, at each aggregation node we need not perform set membership queries in any particular order. This property allows us to take advantage of hardware parallelism and multi-port embedded memory technology.

The second key concept in *DCFL* is employing a network of aggregation nodes to compute the set of matching filters for a given packet. The aggregation network consists of a set of interconnected aggregation nodes which perform set membership queries to the sets of unique field value combinations, $F_{1,2}$, $F_{3,4,5}$, etc. By performing the aggregation in a multi-stage, distributed fashion, the number of intermediate results operated on by each aggregation node remains small. Consider the case of finding all matching address prefix pairs in the example filter set in Table 1.1 for a packet with address pair $(x, y) = (10011100, 01101010)$. As shown in Figure 7.2, an aggregation node takes as input the sets of matching field labels generated by the source and destination address search

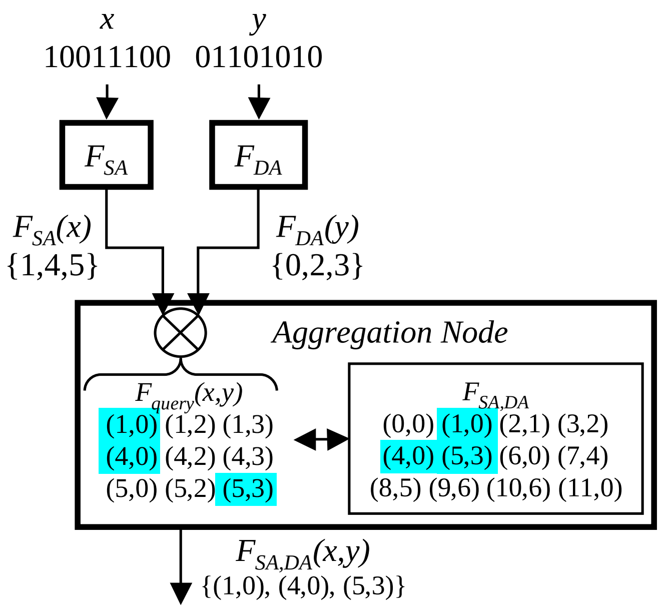


Figure 7.2: Example aggregation node for source and destination address fields.

engines, $F_{SA}(x)$ and $F_{DA}(y)$, respectively. Searching the tables of unique field values shown in Table 7.1, $F_{SA}(x)$ contains labels {1,4,5} and $F_{DA}(y)$ contains labels {0,2,3}. The first step is to form a query set F_{query} of aggregate labels corresponding to potential address prefix pairs. The query set is formed from the crossproduct of the source and destination address label sets. Next, each label in F_{query} is checked for membership in the set of labels stored at the aggregation node, $F_{SA,DA}$. Note that the set of composite labels corresponds to unique address prefix pairs specified by filters in the example filter set shown in Table 1.1. Composite labels contained in the set are added to the matching label set $F_{SA,DA}(x, y)$ and passed to the next aggregation node. Since the number of unique field values and field value combinations is limited in real filter sets, the size of the crossproduct at each aggregation node remains manageable. By performing crossproducting in a distributed fashion across a network of aggregation nodes, we avoid an exponential increase in search time that occurs when aggregating the results from all field search engines in a single step. Note that the aggregation nodes only store unique combinations of fields present in the filter table; therefore, we also avoid the exponential blowup in memory requirements suffered by the original *Crossproducting* technique [53] and *Recursive Flow Classification* [50]. In Section 7.3, we introduce *Field Splitting* which limits the size of F_{query} at aggregation nodes, even when the number matching labels generated by field search engines increases.

DCFL is amenable to various implementation platforms, and where possible, we will highlight the various configurations of the technique that are most suitable for the most popular platforms. In order to illustrate the value of our approach, we focus on the highest performance option for the remainder of this paper. It is important to briefly describe this intended implementation platform here, as it will guide the selection of data structures for aggregation nodes and optimizations in the following sections. Specifically, it is our goal to make full use of the high-degree of parallelism and numerous multi-port embedded memory blocks provided by the current generation of Application Specific Integrated Circuit (ASIC) and Field-Programmable Gate Array (FPGA) technologies discussed in Section 4.7. This requires that we maximize parallel computations and storage efficiency. In Section 7.7 we show that an optimized *DCFL* implementation can support hundreds of thousands of filters in a current generation device without the need for external memory; however, a limited number of high-performance off-chip memory devices such as Dual Data Rate (DDR) and Quad Data Rate (QDR) SRAMs could be employed to support even larger filter sets.

7.2 Aggregation Network

Since all aggregation nodes operate in parallel, the performance bottleneck in the system is the aggregation node with the largest worst-case query set size, $|F_{query}|$. Query set size determines the number of sequential memory accesses performed at the node. The size of query sets vary for different constructions of the aggregation network. We refer to the worst-case query set size, $|F_{query}|$, among all aggregation nodes, $F_1, \ldots, F_{1,\ldots,d}$, as the cost for network construction, G_i. Selecting the most efficient arrangement of aggregation nodes into an aggregation network is a key issue. We want to select the minimum cost aggregation network G_{min} as follows:

$$G_{min} = G : cost(G) = min\left\{cost\left(G_i\right) \forall i\right\} \tag{7.1}$$

where

$$cost\left(G\right) = max\left\{|F_{query}| \forall F_1, \ldots, F_{1,\ldots,d} \in G_i\right\} \tag{7.2}$$

Consider an example for packet classification on three fields. Shown in Figure 7.3 are the maximum sizes for the sets of matching field labels for the three fields and the maximum size for the sets of matching labels for all possible field combinations. For example, label set $F_{1,2}(x, y)$ will contain at most four labels for any values of x and y. Also shown in Figure 7.3 are three possible aggregation networks for a *DCFL* search; the cost varies between 3 and 6 depending on the construction.

In general, an aggregation node may operate on two or more input label sets. Given that we seek to minimize $|F_{query}|$, we limit the number of input label sets to two. The query set size for aggregation nodes fed by field search engines is partly determined by the size of the matching field label sets, which we have found to be small for real filter sets. Also, the *Field Splitting* optimization provides a control point for the size of the query set at the aggregation nodes fed by the field search engines; thus, we restrict the network structure by requiring that at least one of the inputs to each aggregation node be a matching field label set from a field search engine. Figure 7.4 shows a generic aggregation network for packet classification on d fields. Aggregation node $F_{1,\ldots,i}$ operates on matching field label set $F_i(x)$ and matching composite label set $F_{1,\ldots,i-1}(a, \ldots, w)$ generated by upstream aggregation node $F_{1,\ldots,i-1}$. Note that the first aggregation node operates on label sets from

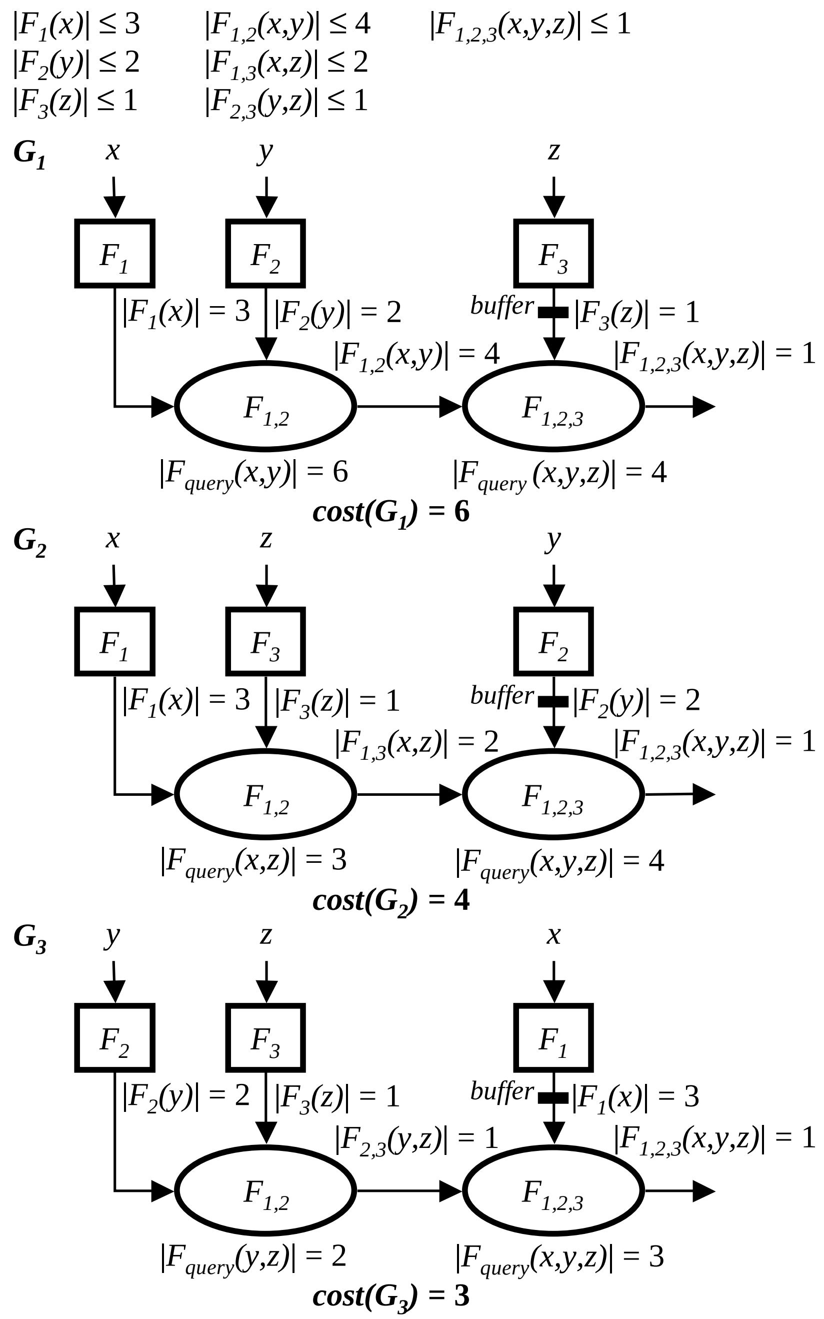


Figure 7.3: Example of variable aggregation network cost for different aggregation network constructions for packet classification on three fields.

two field search engines, $F_1(a)$ and $F_2(b)$. We point out that this seemingly "serial" arrangement of aggregation nodes does not prevent *DCFL* from starting a new search on every pipeline cycle. As shown in Figure 7.4, delay buffers allow field search engines to perform a new lookup on every pipeline cycle. The matching field label sets are delayed by the appropriate number of pipeline cycles such that they arrive at the aggregation node synchronous to the matching label set from the

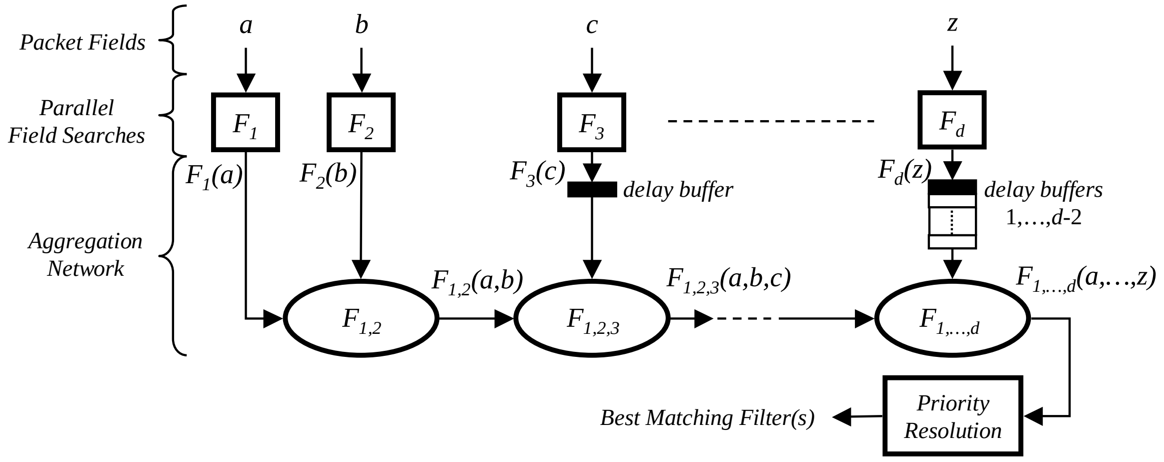


Figure 7.4: Generalized DCFL aggregation network for a search on d fields.

upstream aggregation node. Search engine results experience a maximum delay of $(d-2)$ pipeline cycles which is tolerable given that the pipeline cycle time is on the order of 10ns. With such an implementation, *DCFL* throughput is inversely proportional to the pipeline cycle time.

In this case, the problem is to choose the ordering of aggregation nodes which results in the minimum network cost. For example, do we first aggregate the source and destination field labels, then aggregate the address pair labels with the protocol field labels? We can empirically determine the optimal arrangement of aggregation nodes for a given filter set by computing the maximum query set size for each combination of field values in the filter set. While this computation is manageable for real filter sets of moderate size, the computational complexity increases exponentially with filter set size. For our set of 12 real filter sets, the optimal network aggregated field labels in the order of decreasing maximum matching filter label set size with few exceptions. This observation can be used as a heuristic for constructing efficient aggregation networks for large filter sets and filter sets with large numbers of filter fields. As previously discussed, we do not expect the filter set properties leveraged by *DCFL* to change. We do point out that a static arrangement of aggregation nodes might be subject to degraded performance if the filter set characteristics were dramatically altered by a sequence of updates. Through the use of reconfigurable interconnect in the aggregation network and extra memory for storing off-line aggregation tables, a *DCFL* implementation can minimize the time for restructuring the network for optimal performance. We defer this discussion to future study.

7.3 Field Splitting

As discussed in Section 7.1, the size of the matching field label set, $|F_i(x)|$, affects the size of the crossproduct, $|F_{query}|$, at the following aggregation node. While we observe that $|F_i(x)|$ remains small for real filter sets, we would like to exert control over this value to both increase search speed for existing filter sets and maintain search speed for filter sets with increased address prefix nesting and port range overlaps. Recall that $|F_i(x)| \leq 2$ for all exact match fields such as the transport protocol and protocol flags.

The number of address prefixes matching a given address can be reduced by *splitting* the address prefixes into a set of $(c + 1)$ shorter address prefixes, where c is the number of splits. An example of splitting a 6-bit address field is shown in Figure 7.5. For the original 6-bit address field, *A(5:0)*, the maximum number of field labels matching any address is five. In order to reduce this number, we split the 6-bit address field into a 2-bit address field, *A(5:4)*, and a 4-bit address field, *A(3:0)*. Each original 6-bit prefix creates one entry in each of the new prefix fields as shown. If an original prefix is less than three bits in length, then the entry in field *A(3:0)* is the wildcard. We assign a label to each of the unique prefixes in the new fields and create data structures to search the new fields in parallel in separate search engines. In this example we use binary trees; regardless of the data structure, the search engine must return all matching prefixes. The prefixes originally in *A(5:0)* are now identified by the unique combination of labels corresponding to their entries in *A(5:4)* and *A(3:0)*. For example, the prefix $000*$ in *A(5:0)* is now identified by the label combination $(3, 1)$. A search proceeds by searching *A(5:4)* and *A(3:0)* with the first two bits and remaining 4 bits of the packet address, respectively. Note that the maximum number of field labels returned by the new search engines is three. We point out that the sets of matching labels from *A(5:4)* and *A(3:0)* may be aggregated in any order, with label sets from any other filter field; we need not aggregate the labels from *A(5:4)* and *A(3:0)* in the same aggregation node to ensure correctness. For address prefixes, *Field Splitting* is similar to constructing a variable-stride multi-bit trie; however, with *Field Splitting* we only store one multi-bit node per stride. A matching prefix is denoted by the combination of matching prefixes from the multi-bit nodes in each stride.

Given that the size of the matching field label sets is the property that most directly affects *DCFL* performance, we would like to specify a maximum set size and split those fields that exceed the threshold. Given a field overlap threshold, there is a simple algorithm for determining the number of splits required for an address prefix field. For a given address prefix field, we begin by forming a list of all unique address prefixes in the filter set, sorted in non-decreasing order of prefix length. We simply add each prefix in the list to a binary trie, keeping track of the number of prefixes encountered along the path using a nesting counter. If there is a split at the current prefix length, we reset the nesting counter. The splits for the trie may be stored in a list or an array indexed by the prefix length. If the number of prefixes along the path reaches the threshold, we create a split at that prefix length and reset the nesting counter. It is important to note that the number of splits depends upon the structure of the address trie. In the worst case, a threshold of two overlaps could create a split at every prefix length. We argue that given the structure of real filter sets and reasonable threshold values (four or five), that *Field Splitting* provides a highly useful control point for the size of query sets in aggregation nodes.

A(5:0)	Label	A(5:4)	Label	A(3:0)	Label
*	0	*	0	*	0
0*	1	0*	1	*	0
01*	2	01	2	*	0
000*	3	00	3	0*	1
0110*	4	01	2	10*	2
1010*	5	10	4	10*	2
10100*	6	10	4	100*	3
011010	7	01	2	1010	4

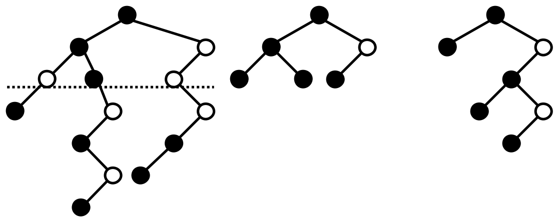


Figure 7.5: An example of splitting a 6-bit address field; maximum number of matching labels per field is reduced from five to three.

Field Splitting for port ranges is much simpler. We simply compute the maximum field overlap, m, for the given port field by adding the set of unique port ranges to a segment tree. Given an overlap threshold, t, the number splits is simply $c = \frac{m-2}{t-1}$. We then create $(c + 1)$ bins in which to sort the set of unique port ranges. For each port range $[i : j]$, we identify the bin, b_i, containing the minimum number of overlapping ranges using a segment tree constructed from the ranges in the bin. We insert $[i : j]$ into bin b_i and insert wildcards into the remaining bins. Once the sorting is complete, we assign locally unique labels to the port ranges in each bin. Like address field splitting, a range in the original filter field is now identified by a combination of labels corresponding to its matching entry in each bin. Again, label aggregation may occur in any order with labels from any other field.

Finally, we point out that *Field Splitting* is a precomputed optimization. It is possible that the addition of new filters to the filter set could cause one the overlap threshold to be exceeded in a particular field, and thus degrade the performance of *DCFL*. While this is possible, our analysis of real filter sets suggests that it is not probable. Currently most filter sets are manually configured, thus updates are exceedingly rare relative to searches. Furthermore, the common structure of filters in a filter set suggests that new filters will most likely be a new combination of fields already in the filter set. For example, a network administrator may add a filter matching all packets for application A flowing between subnets B and C, where specifications A, B, C already exist in the filter set.

7.4 Aggregation Nodes

Well-studied data structures such as hash tables and B-Trees are capable of efficiently representing a set [13]. We focus on three options that minimize the number of sequential memory accesses, *SMA*, required to identify the composite labels in F_{query} which are members of the set $F_{1,\ldots,i}$. The first is a variant on the popular Bloom filter which has received renewed attention in the research literature [15]. The second and third options leverage the compression provided by field labels and meta-labels to index into an array of lists containing the composite labels for the field value combinations in $F_{1,\ldots,i}$. These indexing schemes perform parallel comparisons in order to minimize the required *SMA*; thus, the performance of these schemes depends on the word size m of the memory storing the data-structures. For all three options, we derive equations for the *SMA* and number of memory words W required to store the data-structure.

7.4.1 Bloom Filter Arrays

A Bloom filter is an efficient data structure for set membership queries with tunable false positive errors. In our context, a Bloom filter computes k hash functions on a label L to produce k bit positions in a bit vector of m bits. If all k bit positions are set to 1, then the label is declared to be a member of the set. Broder and Mitzenmacher provide a nice introduction to Bloom filters and their use in recent work [15]. We provide a brief introduction to Bloom filters and a derivation of the equations governing false positive probability in Section 2.1.3. False positive answers to membership queries causes the matching label set, $F_{1,\ldots,i}(a,\ldots,x)$, to contain labels that do not correspond to field combinations in the filter set. These false positive errors can be "caught" at downstream aggregation nodes using explicit representations of label sets. We discuss two options for such data-structures in the next section. This property does preclude use of Bloom filters in the last aggregation node in the network. As we discuss in Section 7.7, this does not incur a performance penalty in real filter sets.

Given that we want to minimize the number of sequential memory accesses at each aggregation node, we want to avoid performing multiply memory accesses per set membership query. It would be highly inefficient to perform k separate memory accesses to check if a single bit is set in the vector. In order to limit the number of memory accesses per membership query to one, we propose the use of an array of Bloom filters as shown in Figure 7.6. A *Bloom Filter Array* is a set of Bloom filters indexed by the result of a pre-filter hash function $H(L)$. In order to perform a set membership query for a label L, we read the Bloom filter addressed by $H(L)$ from memory and store it in a register. We then check the bit positions specified by the results of hash functions $h_1(L),\ldots,h_k(L)$. The *Match Logic* checks if all bit positions are set to 1. If so, it adds label L to the set of matching labels $F_{1,\ldots,i}(a,\ldots,x)$.

Set membership queries for the labels in F_{query} need not be performed in any order and may be performed in parallel. Using an embedded memory block with P ports requires P copies of the

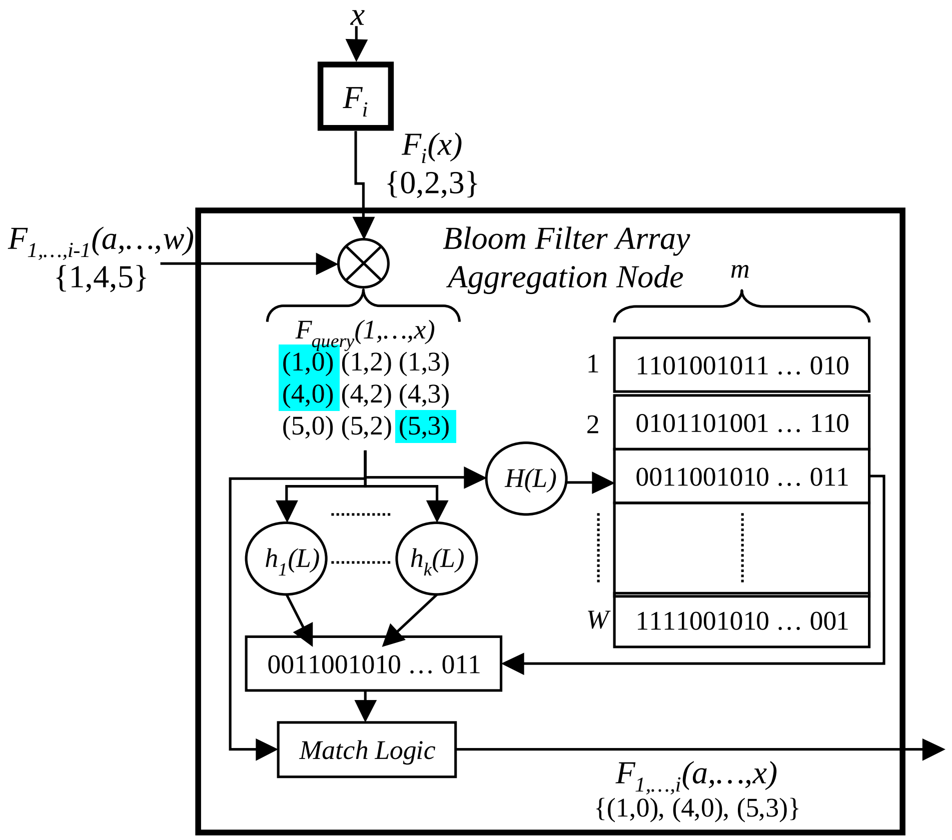


Figure 7.6: Example of an aggregation node using a *Bloom Filter Array* to aggregate field label set $F_i(x)$ with label set $F_{1,\ldots,i-1}(a,\ldots,w)$.

logic for the hash functions and *Match Logic*. Given the ease of implementing these functions in hardware and the fact that P is rarely more than four, the additional hardware cost is tolerable. The number of sequential memory accesses, *SMA*, required to perform set membership queries for all labels in F_{query} is simply

$$\mathit{SMA} = \frac{|F_{query}|}{P} \tag{7.3}$$

The false positive probability is

$$f = \left(\frac{1}{2}\right)^k \tag{7.4}$$

when the following relationship holds

$$k = \frac{m}{n}\ln 2 \tag{7.5}$$

where n is the number of labels $|F_{1,\ldots,i}|$ stored in the Bloom filter. Setting k to four produces a tolerable false positive probability of 0.06. Assuming that we store one Bloom filter per memory word, we can calculate the required memory resources given the memory word size m. Let W be the number of memory words. The hash function $H(L)$ uniformly distributes the labels in $F_{1,\ldots,i}$ across the W Bloom filters in the *Bloom Filter Array*. Thus, the number of labels stored in each

Bloom filter is

$$n = \frac{|F_{1,\ldots,i}|}{W} \tag{7.6}$$

Using Equation 7.5 we can compute the number of memory words, W, required to maintain the false positive probability given by Equation 7.4:

$$W = \left\lceil \frac{k \times |F_{1,\ldots,i}|}{m \times \ln 2} \right\rceil \tag{7.7}$$

The total memory requirement is $m \times W$ bits. Recent work has provided efficient mechanisms for dynamically updating Bloom filters [16, 25].

7.4.2 Meta-Label Indexing

We can leverage the compression provided by meta-labels to construct aggregation nodes that explicitly represent the set of field value combinations, $F_{1,\ldots,i}$. The field value combinations in $F_{1,\ldots,i}$ can be identified by a composite label which is the concatenation of the meta-label for the combination of the first $(i-1)$ fields, $L_{1,\ldots,i-1}$, and the label for field i, L_i. We sort these composite labels into bins based on meta-label $L_{1,\ldots,i-1}$. For each bin, we construct a list of the labels L_i, where each entry stores L_i and the new meta-label for the combination of i fields, $L_{1,\ldots,i}$. We store these lists in an array A_i indexed by meta-label $L_{1,\ldots,i-1}$ as shown in Figure 7.7.

Using $L_{1,\ldots,i-1}$ as an index allows the total number of set membership queries to be limited by the number of meta-labels received from the upstream aggregation node, $|F_{1,\ldots,i-1}(a,\ldots,w)|$. Note that the size of a list entry, s, is

$$s = \lg |F_i| + \lg |F_{1,\ldots,i}| \tag{7.8}$$

and s is typically much smaller than the memory word size, m. In order to limit the number of memory accesses per set membership query, we store N list entries in each memory word, where $N = \lfloor \frac{m}{s} \rfloor$. This requires $N \times |F_i(x)|$ way match logic to compare all of the field labels in the memory word with the set of matching field labels from the field search engine, $F_i(x)$. Since set membership queries may be performed independently, the total number of sequential memory accesses, *SMA*, depends on the size of the index meta-label set, $|F_{1,\ldots,i-1}(a,\ldots,w)|$, the size of the lists indexed by the labels in $F_{1,\ldots,i-1}(a,\ldots,w)$, and the number of memory ports P. In the worst case, the labels index the $|F_{1,\ldots,i-1}(a,\ldots,w)|$ longest lists in A_i. Let $Length$ be an array storing the lengths of the lists in A_i in decreasing order. The worst-case sequential memory accesses is

$$\textit{SMA} = \frac{\sum_{j=1}^{|F_{1,\ldots,i-1}(a,\ldots,w)|} \left\lceil \frac{Length(j)}{N} \right\rceil}{P} \tag{7.9}$$

As with the *Bloom Filter Array*, the use of multi-port memory blocks does require replication of the multi-way match logic. Due to the limited number of memory ports, we argue that this represents

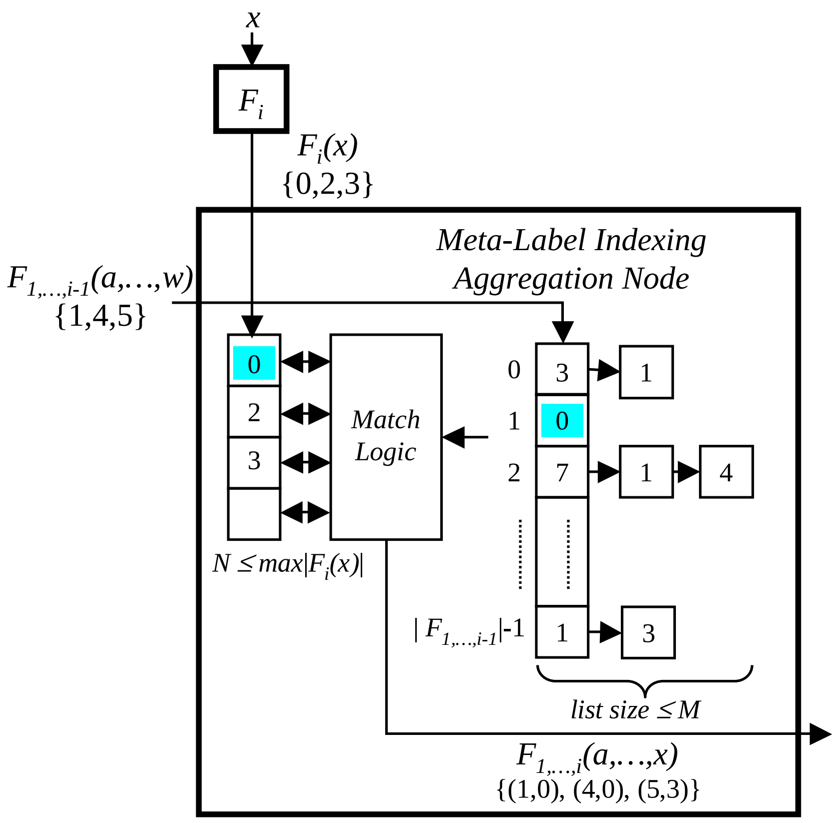


Figure 7.7: Example of an aggregation node using *Meta-Label Indexing* to aggregate field label set $F_i(x)$ with meta-label set $F_{1,\ldots,i-1}(a,\ldots,w)$.

a negligible increase in the resources required to implement *DCFL*. The number of memory words, W, needed to store the data structure is

$$W = \sum_{j=1}^{|F_{1,\ldots,i-1}|} \left\lceil \frac{Length(j)}{N} \right\rceil \tag{7.10}$$

The total memory requirement is $m \times W$ bits. Adding or removing a label from $F_{1,\ldots,i}$ requires an update to a single list entry. Packing multiple list entries on to a single memory word slightly complicates the memory management; however, given that we seek to minimize the number of memory words occupied by a list, the number of individual memory reads and writes per update is small.

Finally, we point out that the data structure may be re-organized to use L_i as the index. This variant, *Field Label Indexing*, is effective when $|F_x|$ approaches $|F_{1,\ldots,x}|$. When this is the case, the number of composite labels $L_{1,\ldots,i}$ containing label L_i is small and the length of the lists indexed by $F_i(x)$ are short.

7.5 Field Search Engines

A primary advantage of *DCFL* is that it allows each filter field to be searched by a search engine optimized for the particular type of search. We discuss a number of single field search techniques in Chapter 2. While the focus of this chapter is the novel aggregation technique, we briefly discuss single field search techniques suitable for use with *DCFL* in order to to highlight the potential performance.

7.5.1 Prefi x Matching

Due to its use of decomposition, *DCFL* requires that the search engines for the IP source and destination addresses return *all* matching prefixes for the given addresses. As discussed in Section 2.3, any longest prefix matching technique can support All Prefix Matching (APM), but some more efficiently than others. The most computationally efficient technique for longest prefix matching is *Binary Search on Prefix Lengths* [24]. When precomputation and marker optimizations are used, the technique requires at most five hash probes per lookup for 32-bit IPv4 addresses. As reported in Section 5.4, real filter sets contain a relatively small number of unique prefix lengths, thus the realized performance should be better for real filter sets. Recall that markers direct the search to longer prefixes that potentially match, thus skipping shorter prefixes that may match. In order to support APM, *Binary Search on Prefix Lengths* must precompute all matching prefixes for each "leaf" in the trie defined by the set of address prefixes. While computationally efficient for searches, this technique does present several challenges for hardware implementation. Likewise, the significant use of precomputation and markers degrades the dynamic update performance, as an update may require many memory transactions.

As we demonstrated in Chapter 3, compressed multi-bit trie algorithms readily map to hardware and provide excellent lookup and update performance with efficient memory and hardware utilization. Specifically, our implementation of the Tree Bitmap technique requires at most 11 memory accesses per lookup and approximately six bytes of memory per prefix. Each search engine consumes less than 1% of the logic resources on a commodity FPGA[2]. As discussed in Section 3.6, there are a number of optimizations to improve the performance of this particular implementation. Use of an initial lookup array for the first 16 bits reduces the number of memory accesses to at most seven. Coupled with a simple two-stage pipeline, the number of sequential memory accesses per lookup can be reduced to at most four. Trie-based LPM techniques such as Tree Bitmap easily support all prefix matching with trivial modifications to the search algorithm. For the purpose of our discussion, we will assume an optimized Tree Bitmap implementation requiring at most four memory accesses per lookup and six bytes per prefix of memory.

[2]If targeted to the low-cost Xilinx Spartan-3 family of FPGAs (less than $12 USD for a one million gate device), each engine would cost approximately $0.12 USD.

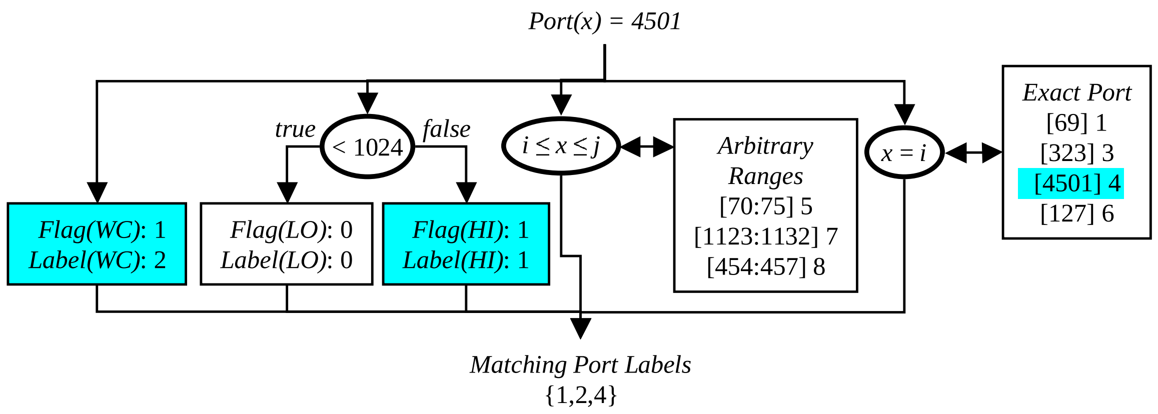


Figure 7.8: Block diagram of range matching using parallel search engines for each port class.

7.5.2 Range Matching

Searching for all arbitrary ranges that overlap a given point presents a greater challenge than prefix matching. We discuss a number of range matching techniques in Section 2.4. Based on the observations reported in Section 5.3.2, range matching can be made sufficiently fast for real filter sets using a set of parallel search engines, one for each port class, as shown in Figure 7.8. Recall that three port classes, WC, HI, and LO, consist of a single range specification. The search engine for the first port class, wildcard (WC), simply consists of a flag specifying whether or not the wildcard is specified by any filters in the filter set and a register for the label assigned to this range specification. Similarly, the search engines for the HI and LO port classes also consist of flags specifying whether or not the ranges are specified by any filters in the filter set and registers for the labels assigned to those range specifications. We also add logic to check if the port is less than 1024; this checks for a match on the HI and LO port ranges, $[1024 : 65535]$ and $[0 : 1023]$, respectively.

For the 12 real filter sets we studied, the number of exact port numbers specified by filters was at most 183. The port ranges in the EM port class may be efficiently searched using any sufficiently fast exact match data-structure. Entries in this data-structure are simply the port number and the assigned label. A simple hash table could bound searches to at most two memory accesses. Finally, the set of arbitrary ranges in the AR port class may be searched with any range matching technique. Fortunately, the set of arbitrary ranges tends to be small; the 12 real filter sets specified at most 27 arbitrary ranges. A simple balanced interval tree data-structure requires at most $O(k \lg n)$ accesses, where k is the number of matching ranges and n is the number of ranges in the tree. Other options for the AR search engine include the *Fat Inverted Segment Tree* discussed in Section 2.4.1 and converting the arbitrary ranges to prefixes as discussed in Section 2.4.3 and employing an all prefix matching search engine. Given the limited number of arbitrary ranges, adding multiple prefixes per range to the data-structure does not cause significant memory inefficiency. With sufficient optimization, we assume that range matching can be performed with at most four sequential memory accesses and the data-structures for the AR and EM port classes easily fit within a standard embedded memory block of 18kb.

7.5.3 Exact Matching

The protocol and flag fields may be easily searched with a simple exact match data-structure such as a hash table. Given the small number of unique protocol and flag specifications in the real filter sets (less than 9 unique protocols and 11 unique flags), the time per search and memory space required is trivial. As we discuss in Section 5.8, we expect that additional filter fields will also require exact match search engines. Given the ease of implementing hash functions in custom and reconfigurable logic, we do not foresee any performance bottlenecks for the search engines for these fields.

7.6 Dynamic Updates

Another strength of *DCFL* is its support of incremental updates. Adding or deleting a filter from the filter set requires approximately the same amount of time as a search operation and does not require that we flush the pipeline and update all data-structures in an atomic operation. An update operation is treated as a search operation in that it propagates through the *DCFL* architecture in the same manner. The query preceding the update in the pipeline operates on data-structures prior to the update; the query following the update in the pipeline operates on data-structures following the update.

Consider inserting a filter to the filter set. We partition the filter into fields (performing field splits, if necessary) and insert each field into the appropriate input buffer of the field search engines. In parallel, each field search engine performs the update operation just as it would perform searches in parallel. As shown in Figure 7.9, an add operation entails a search of the data-structure for the given filter field. If the data-structure does not contain the field, then we add the field to the data-structure and assign the next free label[3]. Finally, we increment the count value for the field entry. Each field search engine returns the label for the filter field. At the next pipeline cycle, the field search engines feed the update operation and field labels to the aggregation network. Logically, the same `Insert` operation is used by both field search engines and aggregation nodes, only the type of *item* and *label* is different for the two. Each aggregation node receives the "insert" command and the labels from the upstream nodes. The *item* is the composite label formed from the labels from the upstream nodes. Note that for an update operation, field search engines and aggregation nodes only pass on one label, thus each aggregation node only operates on one composite label or *item*. If the composite label is not in the set, then the aggregation node adds it to the set. Note that the *label* returned by the `Search` or `Add` operations may be a composite label or meta-label, depending on

[3]We assume that each data-structure keeps a simple list of free labels that is initialized with all available labels. When labels are "freed" due to a delete operation, they are added to the end of the list.

```
Insert(item)

1  label←Search(item)
2  If (label = NULL)
3      label←Add(item)
4  Count[label]++
5  return label
```

Figure 7.9: Pseudocode for *DCFL* update (add).

```
Remove(item)

1  label←Search(item)
2  Count[label]−−
3  If (Count[label] = 0)
4      Delete(item)
5  return label
```

Figure 7.10: Pseudocode for *DCFL* update (delete).

the type of aggregation nodes in use. Finally, the aggregation increments the count for the *label* and passes it on to the next aggregation node. The final aggregation node passes the *label* on to the priority resolution stage which adds the field label to its data-structure according to its priority tag.

Removing a filter from the filter set proceeds in the same way. Both field search engines and aggregation nodes perform the same logical `Remove` operation shown in Figure 7.10. We first find the *label* for the *item*, then decrement the count value for the *item*. A `Delete` operation is performed if the count value for the *item* is zero. The *label* is passed on to the next node in the *DCFL* structure. The final aggregation node passes the filter label to the priority resolution stage which removes the field label from its data-structure.

Note that `Add` and `Delete` operations on field search engine and aggregation node data-structures are only performed when count values change from zero to one and one to zero, respectively. The limited number of unique field values in real filter sets suggests significant sharing of unique field values among filters. We expect typical updates to only change a couple field search engine data-structures and aggregation node data-structures. In the worst case, inserting or removing a filter produces an update to d field search engine data-structures and $(d - 1)$ updates to aggregation node data-structures, where d is the number of filter fields.

7.7 Performance Evaluation

In order to evaluate the performance of *DCFL*, we used 12 real filter sets and the *ClassBench* tools suite to perform simulations testing scalability and sensitivity to filter set properties. The real filter sets were graciously provided from ISPs, a network equipment vendor, and other researchers in the field. The filter sets range in size from 68 to 4557 filters and we discuss their relevant properties in Chapter 5. As described in Chapter 6, we constructed a *ClassBench parameter file* for each filter set and used these files to generate large synthetic filter sets that retain the structural properties of the real filter sets. The *ClassBench Trace Generator* was used to generate input traffic for both the real filter sets and the synthetic filter sets used in the performance evaluation. For all simulations, header trace size is at least an order of magnitude larger than filter set size. The metrics of interest for *DCFL* are the maximum number of sequential memory accesses per lookup at any aggregation node, *SMA*, and the memory requirements. We choose to report the memory requirements in bytes per filter, *BpF*, in order to better assess the scalability of our technique.

The type of embedded memory technology directly influences the achievable performance and efficiency of *DCFL*; thus, for each simulation run we compute the *SMA* and total memory words required for various memory word sizes. Standard embedded memory blocks provide 36-bit memory word widths [107, 74]; therefore, we computed results for memory word sizes of 36, 72, 144, 288, and 576 bits corresponding to using 1, 2, 4, 8, and 16 memory blocks per aggregation node. All results are reported relative to memory word size. The choice of memory word size allows us to explore the tradeoff between memory efficiency and lookup speed. We assert that the use of 16 embedded memory blocks to achieve a memory word size of 576 bits is reasonable given current technology, but certainly near the practical limit. For simplicity, we assume all memory blocks are single-port, $(P = 1)$. Given that all set membership queries are independent, the *SMA* for a given implementation of *DCFL* may be reduced by a factor of P.

In order to demonstrate the achievable performance of *DCFL*, each simulation performs lookups on all possible aggregation network constructions. At the end of the simulation, we compute the optimal aggregation network by choosing the optimal network structure and optimal node type for each aggregation node in the graph. The three node types are discussed in Section 7.4 along with the derivation of the equations for *SMA* and memory requirements for each type: *Bloom Filter Array*, *Meta-Label Indexing*, and *Field Label Indexing*. In the case that two node types produce the same *SMA* value, we choose the node type with the smaller memory requirements. Our simulation also allows us to select the aggregation network structure and node types in order to optimize worst-case or average-case performance. Worst-case optimal aggregation networks select the structure and node types such that the value of the maximum *SMA* for any aggregation node in the network is minimized. Likewise, average-case optimal selects the structure and node types such that the maximum value of the average *SMA* for any aggregation node in the network is minimized. Computing the optimal aggregation network at the end of the simulation allows us to observe trends in the optimal network structure and node type for filter sets of various type, structure, and size. We observe that optimal network structure and node type largely depends on filter set structure. With few exceptions, variables such as filter set size and memory word size do not affect the composition of the optimal aggregation network. We observe that the *Bloom Filter Array* technique is commonly selected as the optimal choice for the first one or two nodes in the aggregation network. With rare exceptions, *Meta-Label Indexing* is chosen for aggregation nodes at the end of the aggregation network. This is a convenient result, as the final aggregation node in the network cannot use the *Bloom Filter Array* technique in order to ensure correctness. We find this result to be somewhat intuitive since the size of a meta-label increases with the number of unique combinations in the set which typically increases with the number of fields in the combination. When using meta-labels to index into an array of lists, a larger meta-label addresses a larger space which in turn “spreads” the labels across a larger array and limits the length of the lists at each array index.

In the first set of tests we used the 12 real filter sets and generated header traces using the *ClassBench Trace Generator*. The number of headers in the trace was 50 times the number of filters in the filter set. As shown in Figure 7.11(a), the worst-case *SMA* for all 12 real filter sets is ten or less for a worst-case optimal aggregation network using memory blocks with a word size of 288 bits. Also note that the largest filter set, *acl5*, of 4557 filters achieves the best performance with a worst-case *SMA* of two for worst-case optimal aggregation network using memory blocks with a word size of 144 bits. In order to translate these results into achievable lookup rates, assume a current generation ASIC with dual-port memory blocks, $(P = 2)$, operating at 500 MHz. The worst-case *SMA* for all 12 filter sets is then five or less using a word size of 288 bits. Under these assumptions, the pipeline cycle time can be 10ns allowing the *DCFL* implementation to achieve 100 million searches per second which is comparable to current TCAMs. Search performance can be doubled by doubling the clock frequency or using quad-port memory blocks, both of which are possible in current generation ASICs.

As shown in Figure 7.11(c), the average *SMA* for all filter sets falls to four or less using a memory word size of 288 bits. Filter set *acl5* also achieves the best average performance with an average *SMA* of 1.2 for a word size of 288. As in many other packet classification techniques, average performance is significantly better than worst-case performance.

Worst-case optimal memory consumption is shown in Figure 7.11(e). Most filter sets required at most 40 bytes per filter (*BpF*) for all word sizes; thus, 1MB of embedded memory would be sufficient to store 200k filters. There are two notable exceptions. The results for filter set *acl1* show a significant increase in memory requirements for larger word sizes. For memory word sizes of 36, 72, and 144 bits, *acl1* requires less than 11 bytes per filter; however, memory requirements increase to 61 and 119 bytes per filter for word sizes 288 and 576, respectively. We also note that increasing the memory word size for *acl1* yields no appreciable reduction in *SMA*; all memory word sizes yielded an *SMA* of five or six. These two pieces of data suggest that in the aggregation node data-structures, the size of the lists at each index entry are short; thus, increasing the memory

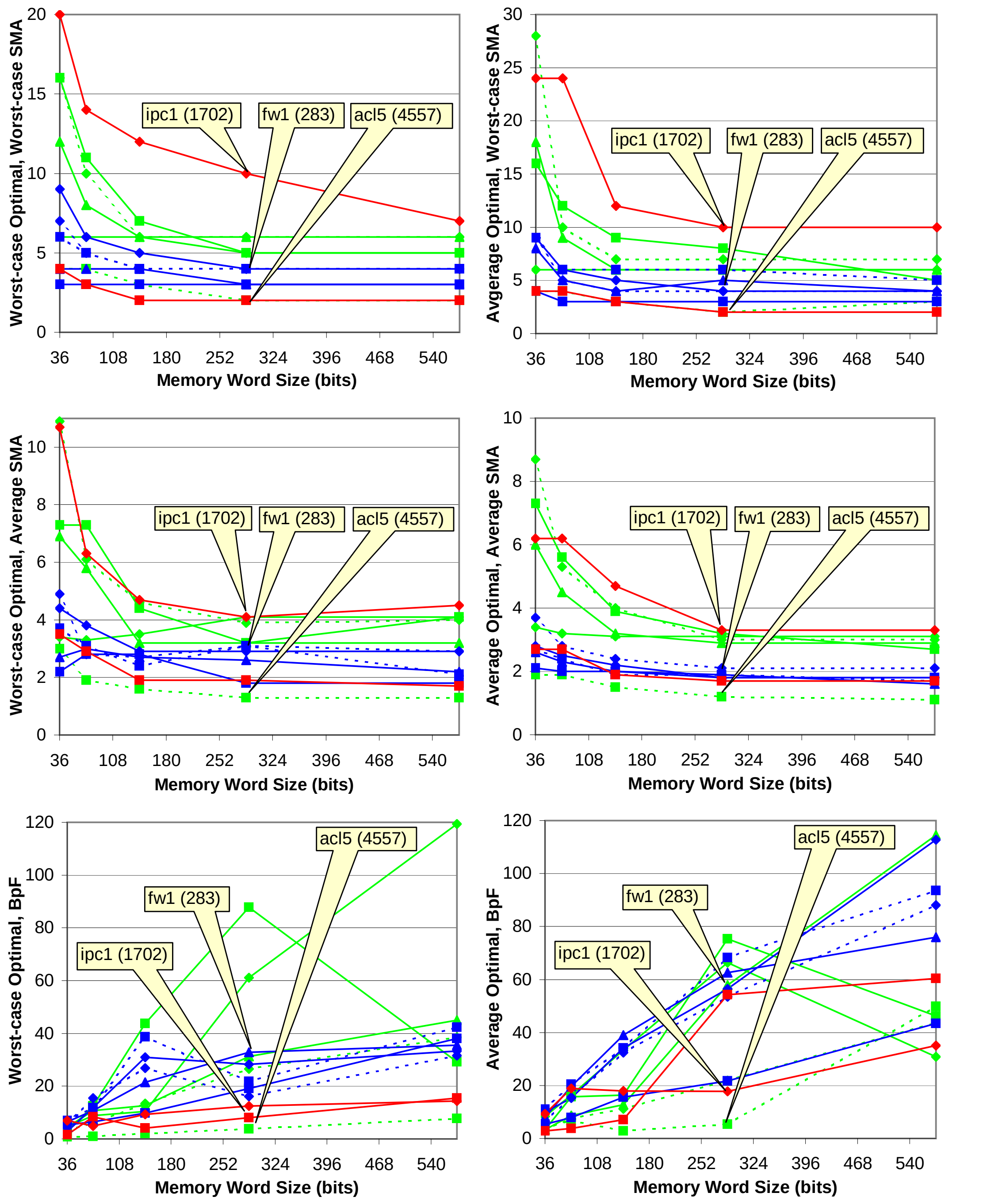


Figure 7.11: Performance results for 12 real filter sets; left-column shows worst-case sequential memory accesses (SMA), average SMA, and memory requirements in bytes per filter (BpF) for aggregation network optimized for worst-case SMA; right-column shows same results for aggregation network optimized for average-case SMA; call-outs highlight three specific filter sets of various sizes and types (filter set size given in parentheses).

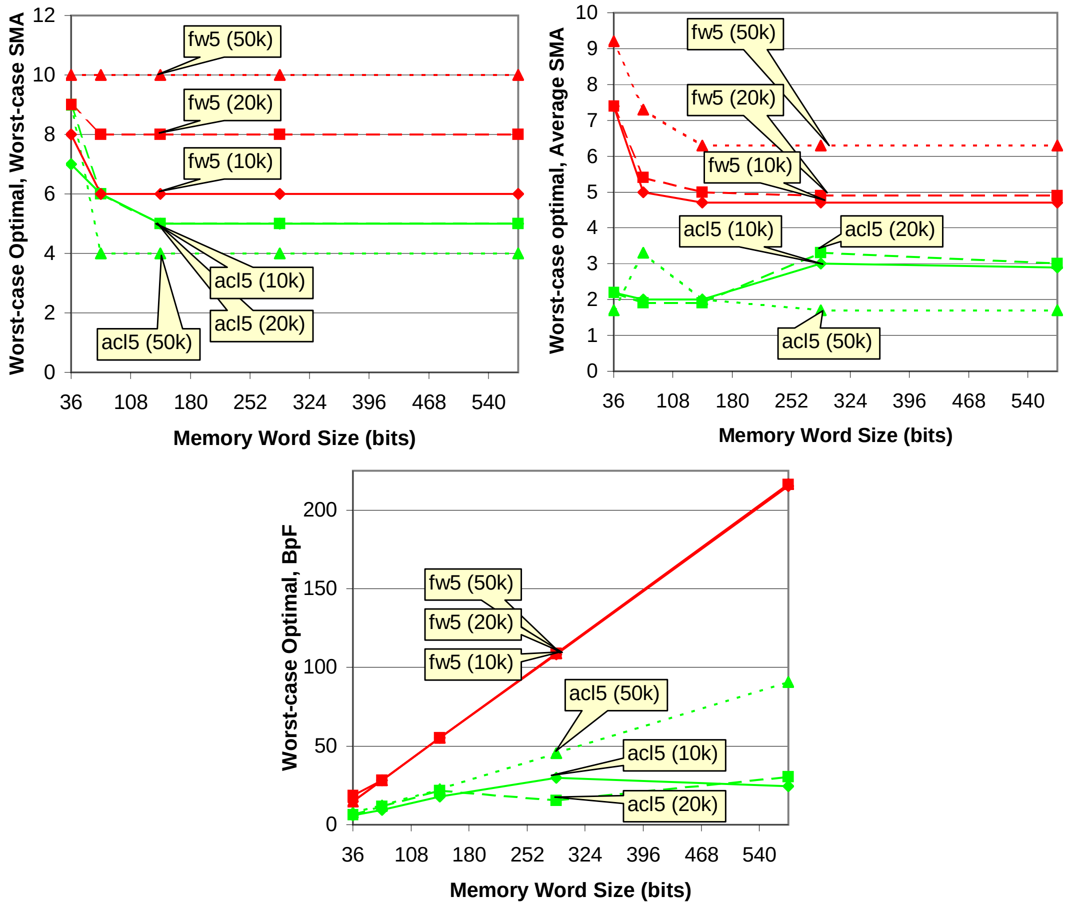


Figure 7.12: Performance results for synthetic filter sets containing 10k, 20k, and 50k filters, generated with parameter files from filter sets *acl5* and *fw5*; call-outs highlight most pronounced effects (number of filters given in parentheses).

word-size linearly increases the memory inefficiency without yielding any fewer memory accesses. We believe that this is also the case with the optimal aggregation network for *acl2* with memory word size 288. Clearly, finding the optimum balance of lookup performance and memory efficiency requires careful selection of memory word size.

Figure 7.11(b) shows the worst-case *SMA* for all 12 real filter sets for an average-case optimal aggregation network. Figure 7.11(d) shows the average *SMA* for all 12 real filter sets for an average-case optimal aggregation network. When optimizing for average *SMA*, average performance is improved by approximately 25%, but worst-case performance suffers by approximately 50%. With the exception of rare application environments, sacrificing worst-case performance for average performance is unfavorable. For the remaining simulations, we only report worst-case optimal results.

The second set of simulations investigates the scalability of *DCFL* to larger filter sets. Results are shown in Figure 7.12. This set of simulations utilized the *ClassBench* tools suite to generate synthetic filter sets containing 10k, 20k, and 50k filters using *parameter files* extracted from filter sets *acl5* and *fw5*. As shown in Figure 7.12(a), the worst-case *SMA* is ten or less for all filter sets and memory word sizes. The most striking feature of each simulation is the flat response to memory word size. For all filter sets generated with the *fw5 parameter file*, the worst-case *SMA* performance remains constant for memory word sizes greater than or equal to 72 bits. For all filter sets generated with the *acl5 parameter file*, the worst-case *SMA* performance remains constant for memory word sizes greater than or equal to 144 bits. The *ClassBench Synthetic Filter Set Generator* maintains the field overlap properties specified in the *parameter file*. Coupled with the results in Figure 7.12, this confirms that the property of filter set structure most influential on *DCFL* performance is the maximum number of unique field values matching any packet header field. As discussed in Chapter 5, we expect this property to hold as filter sets scale in size. If field overlap does increase, the *Field Splitting* optimization provides a way to reduce this to a desired threshold. As shown in Figure 7.12(c), the memory requirements increase with memory word size. Given the favorable *SMA* performance there is no need to increase the word size beyond 144 bits, as it only results in a linear increase in memory inefficiency. These results imply that tuning the memory word size is less critical for large filter sets.

The third set of simulations investigates the effect of filter scope on the performance of *DCFL*. Recall that scope is measure of the specificity of the filters in the filter set. *ClassBench* provides high-level control over the average scope of the filters in the filter set via an input parameter s. We generated synthetic filter sets containing 16000 filters using *parameter files* from a variety of filter sets.For each *parameter file*, we generated filter sets using scope parameters -1, 0, and 1. Note that these filter sets are used in the evaluation of the *ClassBench* tools suite in Figure 6.4.2. The scope parameter had the most pronounced effects on worst-case *SMA* for the filter sets generated with the *parameter file* from *ipc1*. As shown in Figure 7.13(a), decreasing the average scope of the filters in the filter set ($s = -1$) results in significantly better performance; thus, as filters become more specific the performance of *DCFL* improves. This is a favorable result given the generally accepted conjecture the primary source of future filter set growth will be flow specific filters for applying network services. If we increase the scope of the filters in the filter set, *DCFL* performance suffers. This trend also holds for the average *SMA*. As shown in Figure 7.13(c), filter set specificity has little effect on memory requirements for memory word sizes of 144 bits or less. When using larger memory word sizes, filter sets containing more specific filters require more memory per filter; as filters become less specific they become more memory efficient. We believe this result is due to the fact that less-specific filter fields are more likely to be used by several filters. For example, the port range for all user ports is more likely to be used by multiple filters than a specific port number. When we construct filters with less-specific fields, the sharing of filter fields among filters increases and the memory efficiency of labeling is more apparent.

The fourth set of simulations investigate the efficacy and consequences of the *Field Splitting* optimization. We selected two of the worst-performing real filter sets and performed simulations

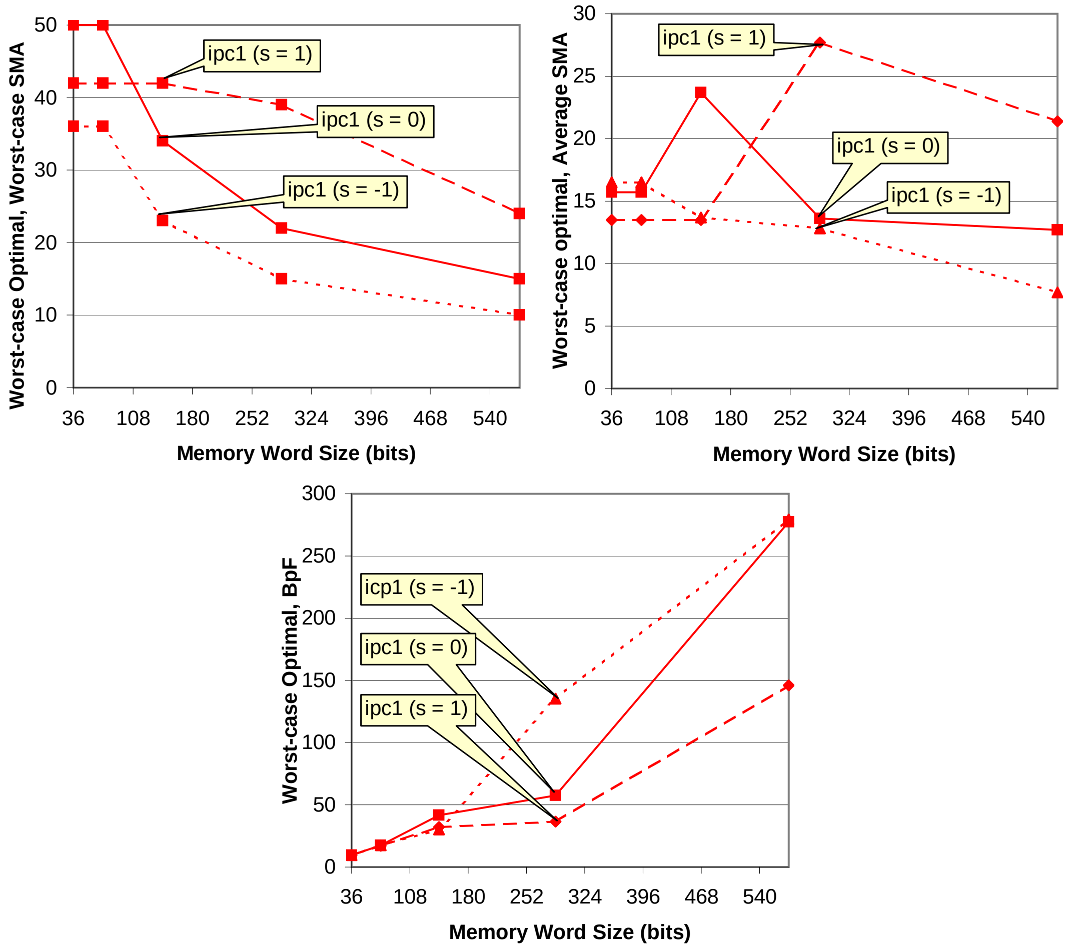


Figure 7.13: Performance results for synthetic filter sets containing 16k filters, generated with the *ipc1 parameter file* with scope parameters s {-1,0,1}; call-outs highlight most pronounced effects (scope parameter given in parentheses); note that these filter sets are used in the evaluation of the *ClassBench* tools suite in Figure 6.4.2.

with various field overlap thresholds. The performance results are summarized in Figure 7.14. For *acl2*, *Field Splitting* reduces the worst-case *SMA* from 16 to 10 for 36-bit memory words. For *fw1*, *Field Splitting* reduces the worst-case *SMA* from 9 to 5 for 36-bit memory words. In these cases, *Field Splitting* provides a 37% and 44% increase in performance, respectively. It is important to note, however, that the impact of *Field Splitting* is reduced as we increase memory word size. Clearly, the primary benefit of *Field Splitting* is that it allows us to achieve better performance using smaller memory word sizes which improves the memory efficiency. As shown in Figure 7.14(c), the memory utilization for all filter sets using memory word sizes of 74-bits or less remains well-below 40 bytes per filter. Consider the specific case of *acl2*. In order to achieve a worst-case *SMA* of eight or less without *Field Splitting*, we must use a memory word-size of 144 bits resulting in memory requirements of 44 bytes per filter. Using *Field Splitting* with a field overlap threshold of three, we

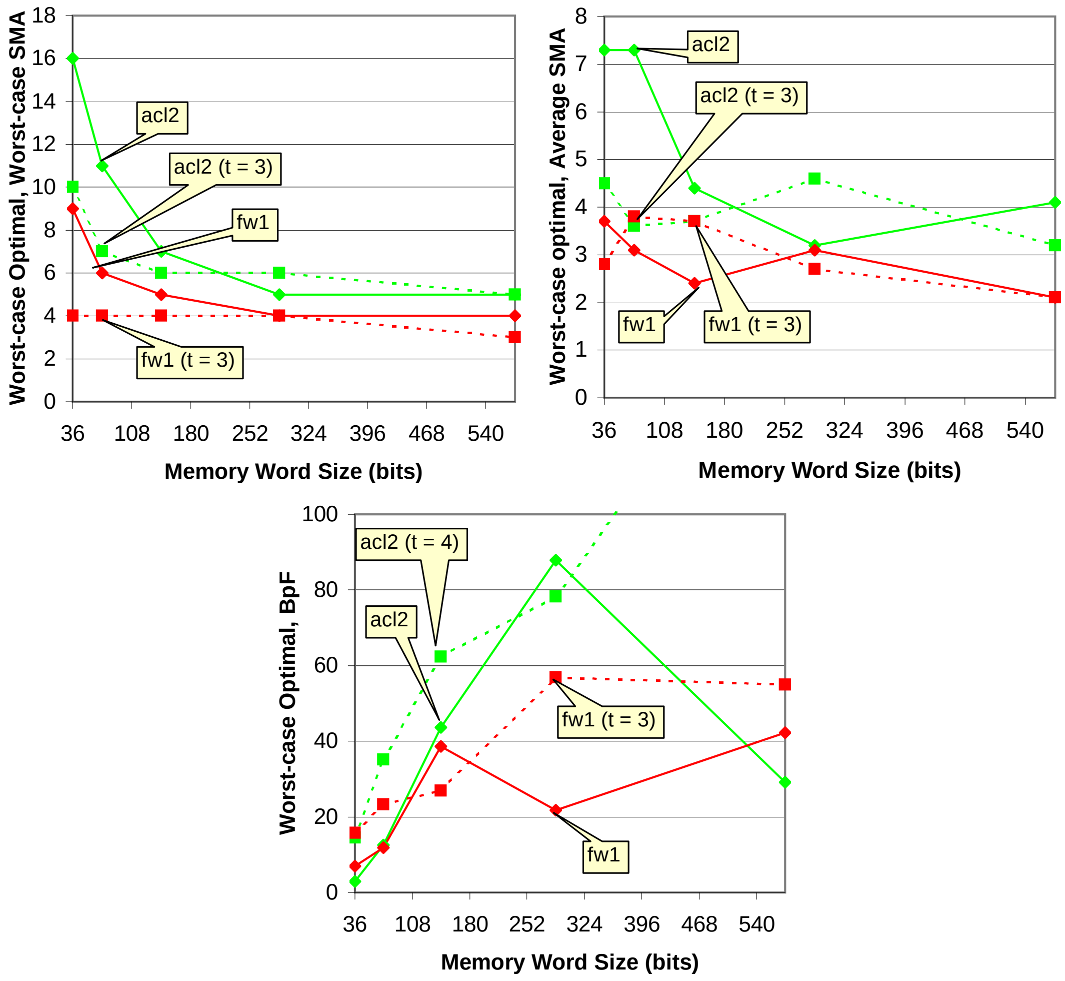


Figure 7.14: Performance results for real filter sets (*acl2* and *fw1*) using the *Field-Splitting* optimization; call-outs highlight most pronounced effects (field overlap threshold given in parentheses).

achieve the desired worst-case *SMA* performance using a memory word-size of 72 bits resulting in memory requirements of 35 bytes per filter. Recall that *Field Splitting* does increase the number of aggregation nodes in the aggregation network, thus increasing the number of memory blocks and logic required for implementation. However, these results show that the total memory requirements are actually reduced for a particular performance target. It is important to note that we do reach a point of diminishing returns with *Field Splitting*. The aggregation network can grow too large if too many splits are required to achieve a particularly low field overlap threshold. In this case, the impact on worst-case *SMA* is minimal while the memory resource requirements increase drastically due to the additional overhead. This situation is reflected in Figure 7.14(c) for filter set *fw1* with a field overlap threshold of three and memory word size of 288 bits.

The fifth and final set of simulations investigate the scalability of *DCFL* to additional filter fields. Using the *ClassBench* tools suite, we generated four filter sets containing 16000 filters using

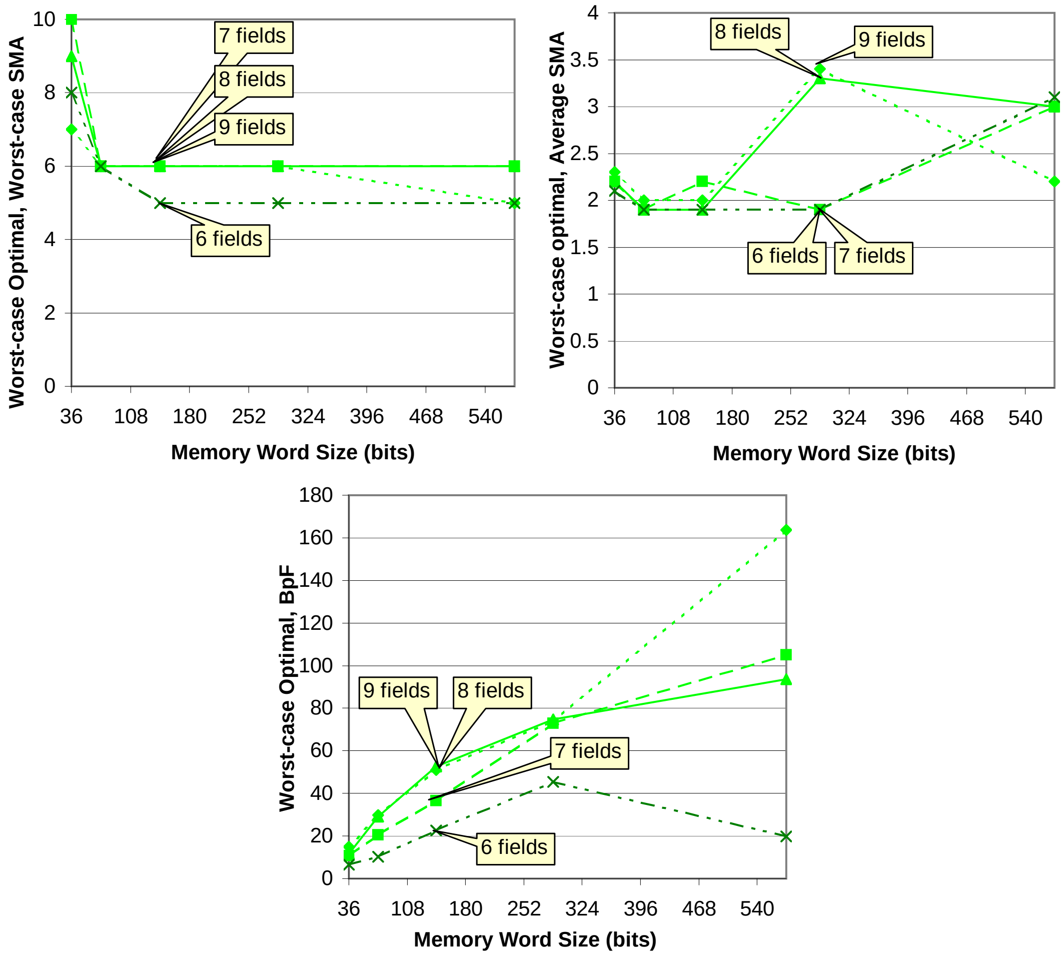


Figure 7.15: Performance results for synthetic filter sets containing 16k filters, generated with parameter file from filter set *acl5* with extra filter fields; call-outs highlight most pronounced effects (number of filter fields given in parentheses).

the *acl5 parameter file*. No *smoothing* or *scope* adjustments were applied. The first filter set was generated such that half of the filters specifying the TCP or UDP protocols specified one non-wildcard field in addition to the standard six filter fields (the 5-tuple plus protocol flags). The non-wildcard field value was selected from a set of 100 random values using a uniform random variable. The remaining filter sets were generated in the same manner with two, three, and four extra field values. As shown in Figure 7.15(a), extra filter fields have a negligible effect on worst-case *SMA* performance. We believe that this is attributable to two impetuses: (1) the additional filter fields allow filters to be more specific, and (2) the additional filter fields are exact match fields and the maximum fields overlap is at most two. As reflected in Figure 7.15(c), the increase in memory requirements for an additional filter field is small for memory word sizes of 144 bits or less. Specifically, when using 144-bit memory words the memory requirements increase by 14 bytes per filter when adding a seventh field and 16 bytes per filter when adding an eighth filter field.

There is no observable increase when adding the ninth filter field. This is constitutes an average of 10 bytes per filter for each additional field. Given our reasonable assumptions regarding the nature of additional filter fields in future filter sets, we assert that the performance and scalability of *DCFL* will make it an even more compelling solution for packet classification as filter sets scale in size and the number of filter fields.

7.8 Related Work

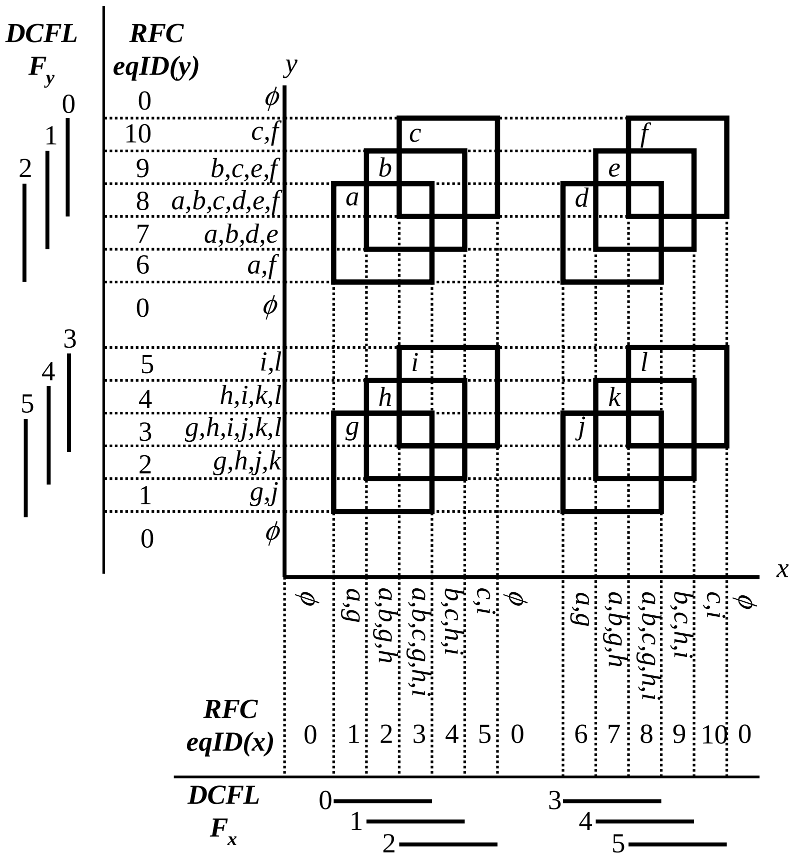
In general, there have been two major threads of research efforts addressing the packet classification problem: algorithmic and architectural. A few pioneering groups of researchers posed the problem, provided complexity bounds, and offered a collection of algorithmic solutions [50, 51, 52, 53]. Subsequently, the design space has been thoroughly explored by many offering new algorithms and improvements upon existing algorithms [54, 27, 29]. Given the inability of early algorithms to meet performance constraints imposed by high speed links, researchers in industry and academia devised architectural solutions to the problem. This thread of research produced the most widely-used packet classification device technology, Ternary Content Addressable Memory (TCAM) [55, 56, 17, 57]. While they provide sufficient speed, current TCAM-based solutions consume exorbitant amounts of power and hardware resources relative to implementations of efficient algorithms. Recent work has addressed many of the unfavorable aspects of current TCAM-based solutions [108, 32]; however, there remain fundamental limits to their scalability and efficiency.

The most promising algorithmic research embraces the practice of leveraging the statistical structure of filter sets to improve average performance [50, 54, 58, 51, 59]. Several algorithms in this class are amenable to high-performance hardware implementation. New architectural research combines intelligent algorithms and novel architectures to eliminate many of the unfavorable characteristics of current TCAMs [32]. We observe that the community appears to be converging on a combined algorithmic and architectural approach to the problem [28]. Our solution, *Distributed Crossproducting of Field Labels* (*DCFL*), employs this combined approach to provide a scalable, high-performance packet classifier. Chapter 4 provides a thorough survey of packet classification techniques using a taxonomy that frames each technique according to its high-level approach. In this section, we highlight the sources of the key ideas and data structures which we distill and utilize in *DCFL*. In order to demonstrate the value of our solution relative to the state of the art, we also contrast it with two leading solutions which are arguably the top solutions from the algorithmic and architectural threads.

As clearly indicated by the name, *DCFL* draws upon the seminal *Crossproducting* technique introduced by Srinivasan, Varghese, Suri, and Waldvogel [53]. *DCFL* avoids the exponential blowup in memory requirements experienced by *Crossproducting* by only storing the labels for field values and combinations of field values present in the filter table. It retains high-performance by aggregating intermediate results in a distributed fashion. Gupta and McKeown introduced *Recursive Flow*

Classification (*RFC*) which provides high lookup rates at the cost of memory inefficiency [50]. Similar to the *Crossproducting* technique, *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, rather than a single step as in *Crossproducting*. The result of each "chunk" lookup and aggregation step in (*RFC*) is an equivalence class identifier, *eqID*, that represents the set of potentially matching filters for the packet. There is a subtle, yet powerful difference between the use of equivalence classes in *RFC* and field labels in *DCFL*. In essence, the number of labels in *DCFL* grows linearly with the number of unique field values in the filter table. The number of *eqIDs* in *RFC* depends upon the number of distinct sets of filters that can be matched by a packet. The number of *eqIDs* in an aggregation step scales with the number of unique overlapping regions formed by filter projections. Another major difference between *DCFL* and *RFC* is the means of aggregating intermediate results. *RFC* lookups in "chunk" and aggregation tables utilize indexing, causing *RFC* to make very inefficient use of memory. The index tables used for aggregation also require significant precomputation in order to assign the proper *eqID* for the combination of the *eqID*s of the previous phases. Such extensive precomputation precludes dynamic updates at high rates. As we have shown, *DCFL* uses efficient set membership data structures which can be engineered to provide fast lookup and update performance. Each data structure only stores labels for unique field combinations present in the filter table; hence, they make efficient use of memory and do not require significant precomputation. In order to illustrate the differences between *RFC* and *DCFL*, we provide an example of an *RFC* search for two "chunks" of a search on n "chunks" in Figure 7.16. The squares $[a \dots l]$ represent the unique projections of the two "chunks" x and y for all filters in a filter table. The number of *eqIDs* for the "chunk" lookups is 11 for each dimension x and y, as 11 unique sets of filters are formed by the projections onto the x and y axes. Since *RFC* utilizes indexing for lookups, each "chunk" table requires 2^b entries, where b is the size in bits of the "chunk". Note that if the number of unique projections were *labeled* as in *DCFL*, only six labels for each dimension would be required, and the set membership data structure would only need to store six entries. In order for *RFC* to aggregate the *eqIDs* from "chunks" x and y, it must compute all of the unique sets of filters for the two-dimensional overlaps. As shown in Figure 7.16, this results in 25 *eqIDs*. The aggregation table requires $2^{4+4} = 256$ entries, as *eqID(x)* and *eqID(y)* are four bits in size and *RFC* utilizes indexing to find *eqID(x,y)*. Note that in *DCFL*, a label would simply be assigned to each unique 2-d projection $[a \dots l]$ and stored in a set membership data structure. In general, *DCFL* can provide line-speed lookups, like *RFC*, but with much more efficient use of memory and support for dynamic updates at high rates.

Our approach also shares similarities with the *Parallel Packet Classification* (P^2C) scheme introduced by van Lunteren and Engbersen [28]. Specifically, both *DCFL* and P^2C fall into the class of techniques using independent field searches coupled with novel encoding and aggregation of intermediate results. The primary advantage of *DCFL* over P^2C is its use of SRAM and amenability to implementation in commodity hardware technology; P^2C requires the use of a separate TCAM



DCFL F_{xy}
(list of unique 2-D projections)

a (0,2)	*g* (0,5)
b (1,1)	*h* (1,4)
c (2,0)	*i* (2,3)
d (3,2)	*j* (3,5)
e (4,1)	*k* (4,4)
f (5,0)	*l* (5,3)

RFC eqID(x,y)
(list of unique 2-D overlaps)

ϕ	0	*g*	13
a	1	*g,h*	14
a,b	2	*h*	15
b	3	*g,h,i*	16
a,b,c	4	*h,i*	17
b,c	5	*i*	18
c	6	*j*	19
d	7	*j,k*	20
d,e	8	*k*	21
e	9	*j,k,l*	22
d,e,f	10	*k,l*	23
e,f	11	*l*	24
f	12		

Figure 7.16: Contrast between unique field value labels in *Distributed Crossproducting of Field Labels* (*DCFL*) and equivalence class identifiers (*eqIDs*) in *Recursive Flow Classification*; example shows two fields of a d field search. Squares $[a \dots l]$ represent the unique projections of two fields x and y for all filters in a filter table.

or a custom ASIC with embedded TCAM. *DCFL* also provides more efficient support of dynamic updates.

Given the volume of work in packet classification, we must show how our technique adds value to the state of the art. In our opinion, *HyperCuts* is one of the most promising new algorithmic solutions [59]. Introduced by Singh, Baboescu, Varghese, and Wang, the algorithm improves upon the *HiCuts* algorithm developed by Gupta and McKeown [51] and also shares similarities with the *Modular Packet Classification* algorithms introduced by Woo [29]. In essence, *HyperCuts* is a decision tree algorithm that attempts to minimize the depth of the tree by selecting "cuts" in multi-dimensional space that optimally segregate packet filters into lists of bounded size. According to performance results given in [59], traversing the *HyperCuts* decision tree required between 8 and 35 memory accesses, and memory requirements for the decision tree ranged from 5.4 to 145.9 bytes per filter. We assert that *DCFL* exhibits advantages in all metrics of interest: worst-case *SMA*, memory requirements, and dynamic update performance. *DCFL* also provides the opportunity to strike a favorable tradeoff between performance and memory requirements, as the various parameters may be tuned to achieve the desired results. All new algorithmic approaches must make a strong case for their advantage relative to Ternary Content Addressable Memory (TCAM). Due to its performance, efficiency, scalability, and use of commodity hardware technology, *DCFL* has the ability to provide equivalent lookup performance at much lower cost and power consumption.

7.9 Discussion

By transforming the problem of aggregating results from independent field search engines into a distributed set membership query, *Distributed Crossproducting of Field Labels* (*DCFL*) avoids the exponential increases in time and memory required by previous approaches. We introduced several new concepts including field labeling, *Meta-labeling* unique field combinations, and *Field Splitting*, as well as optimized set membership data structures such as *Bloom Filter Arrays* that minimize the number of memory accesses required to perform a set membership query. Using a combination of real and synthetic filter sets, we demonstrated that *DCFL* can achieve over 100 million searches per second using existing hardware technology. Furthermore, we have also shown that *DCFL* retains its lookup performance and memory efficiency when the number of filters and number of fields in the filters increases. Scalability to classify on additional fields is a distinct advantage *DCFL* exhibits over existing decision tree algorithms and TCAM-based solutions. We continue to explore optimizations to improve the search rate and memory efficiency of *DCFL*. We also believe that *DCFL* has potential value for other searching tasks beyond traditional packet classification.

Chapter 8

Summary

Only the curious will learn and only the resolute overcome the obstacles to learning.
The quest quotient has always excited me more than the intelligence quotient.
Eugene S. Wilson, Dean of Admissions, Amherst

All grand visions of the "next-generation" Internet assume that route lookup and packet classification search engines will scale to support fast links, larger route tables and filter sets, and more complex packet classification filters. The work described in this dissertation provides several contributions that help meet these challenges. While the fruits of our work have addressed a number of the open problems in packet classification, there remain a number of enticing opportunities for future work.

8.1 Contributions

As evidenced by the number of proposed solutions discussed in Chapters 2 and 4, the route lookup and packet classification problems are well-studied problems. Despite the energetic attention of the research community, there remain a number of ripe areas for contribution. Three of the most pressing issues are efficient search engine implementations, standardized performance evaluation tools, and viable alternatives to TCAMs for packet classification. While many search engine implementations exist, many are targeted to general purpose processor systems or ASICs and most are not open-source or otherwise available for study by the research community. Due to the lack of standard performance evaluation tools, researchers offering new solutions produce their own test vectors, thus comparison of competing solutions is exceedingly difficult. As clearly indicated by recent search engine market dynamics, router designers are increasingly concerned with power consumption and scalability, thus they are beginning to favor algorithmic packet classification solutions over TCAMs. We addressed all three of these areas throughout the course of this dissertation.

Chapter 3 presented the design and analysis of a scalable implementation of Eatherton and Dittia's Tree Bitmap algorithm for route lookup. The Fast Internet Protocol Lookup (FIPL) search engine provides approximately one million lookups per second per engine and several engines may be combined to provide even greater throughputs. Furthermore, each FIPL engine consumes less than 1% of commodity reconfigurable logic device. We have made the VHDL code for the search engine and evaluation environment publicly available. FIPL engines have already been incorporated in a System-on-Chip (SoC) packet processor for the Network Services Platform (NSP) [43] which forms part of the infrastructure for the Open Network Laboratory (ONL) [109]. ONL allows researchers to remotely configure and perform experiments on real networks comprised of heterogeneous hosts, links, and open-platform extensible routers.

In Chapter 4, we provided a survey of packet classification techniques and developed a taxonomy which frames each technique according to its high-level approach to the problem. Through the use of a limited set of running examples, the survey presents a more coherent view of the state-of-the-art and more clearly highlights potential areas for future contributions. We assert that the taxonomy enables a better understanding of the packet classification algorithms, as opposed to simply reporting asymptotic performance bounds or reported performance results for each technique.

Chapter 5 presented a detailed analysis of real filter sets as well as the forces influencing their composition. This is the most comprehensive study of filter set structure that we are aware of. The results of this analysis include an analysis of the storage inefficiency of standard TCAMs and a novel study of the *field overlap* in real filter sets. The latter findings led to the development of *Distributed Crossproducting of Field Labels*, the new packet classification algorithm presented in Chapter 7.

In response to the lack of publicly available filter sets and performance evaluation tools, we developed *ClassBench*. We presented the design and analysis of the *ClassBench* tools in Chapter 6. The combination of the *Synthetic Filter Set Generator* and *parameter files* extracted from real filter sets eliminates confidentiality concerns, and hence removes the access barrier to realistic test vectors. In addition to providing high-level control of the composition of the filters in the synthetic filter sets, the *ClassBench* tools also produce synthetic header traces with variable locality of reference. We have made the *ClassBench* tools publicly available along with *parameter files* from 12 real filter sets and several research groups are already using the tools.

Chapter 7 presented *Distributed Crossproducting of Field Labels* (*DCFL*), a novel combination of new and existing packet classification techniques that leverages key observations of filter set structure and takes advantage of the capabilities of modern hardware technology. We introduced several new concepts including field labeling, *Meta-labeling* unique field combinations, and *Field Splitting*. *DCFL* minimizes the number of sequential memory accesses required per lookup by transforming the problem of aggregating results from independent field search engines into a distributed set membership query. In order to support this novel approach, we developed three efficient data structures including *Bloom Filter Arrays*. Using a set of 12 real filter sets and the *ClassBench* tools suite, we demonstrated that *DCFL* not only provides sufficient lookup performance, but also scales to larger filter sets and more complex filters. Given the anticipated effects of Internet growth and diversification on the size and composition of filter sets, *DCFL* will become an increasingly attractive alternative to TCAMs for packet classification.

8.2 Future Directions

The contributions of this dissertation provide a solid foundation for further research. We plan to promote broader use of *ClassBench* with the hope of refining the tools and developing a formal benchmarking methodology. If embraced by the research community, the consensus building and standardizing effort could be taken up by the Internet Engineering Task Force (IETF), leading to one or more Request for Comment (RFC) documents detailing a packet classification benchmarking methodology.

In order to demonstrate the realizable performance, determine hardware resource consumption, and measure dynamic power consumption, we would like to design and implement a prototype of the *Distributed Crossproducting of Field Labels* algorithm. As shown in Figure 8.1, we envision a scalable, modular design which would allow the use of various field search engines and dynamic reconfiguration of the aggregation network. The Field-programmable Port eXtender or similar open-platform research system with reconfigurable hardware and adequate memory would provide a suitable implementation platform. This design effort would require adequate research funding and human resources to accomplish in a timely manner.

Independent of a hardware prototyping effort, we believe *DCFL* has the potential to provide better performance for a variety of complex searching problems. Several researchers in the networking community have directed their attention to high-performance string matching techniques due to their use in network intrusion detection systems. Some Internet worms and viruses contain a known “signature” or sequence of characters. Searching packet payloads for these signatures at the edge of the network can prevent the spread of malicious programs. Intrusion detection is just one of the applications falling under the broad heading of “deep packet inspection”. Other applications include load-balancing for web server farms which requires inspection of the HTTP header in order to direct the web-page request to the most lightly-loaded server containing the page. Given that the scaling properties and performance of *DCFL* is independent of the type of field search performed, our approach could provide better performance for a variety of hybrid search techniques comprised of exact, range, prefix, and string matching.

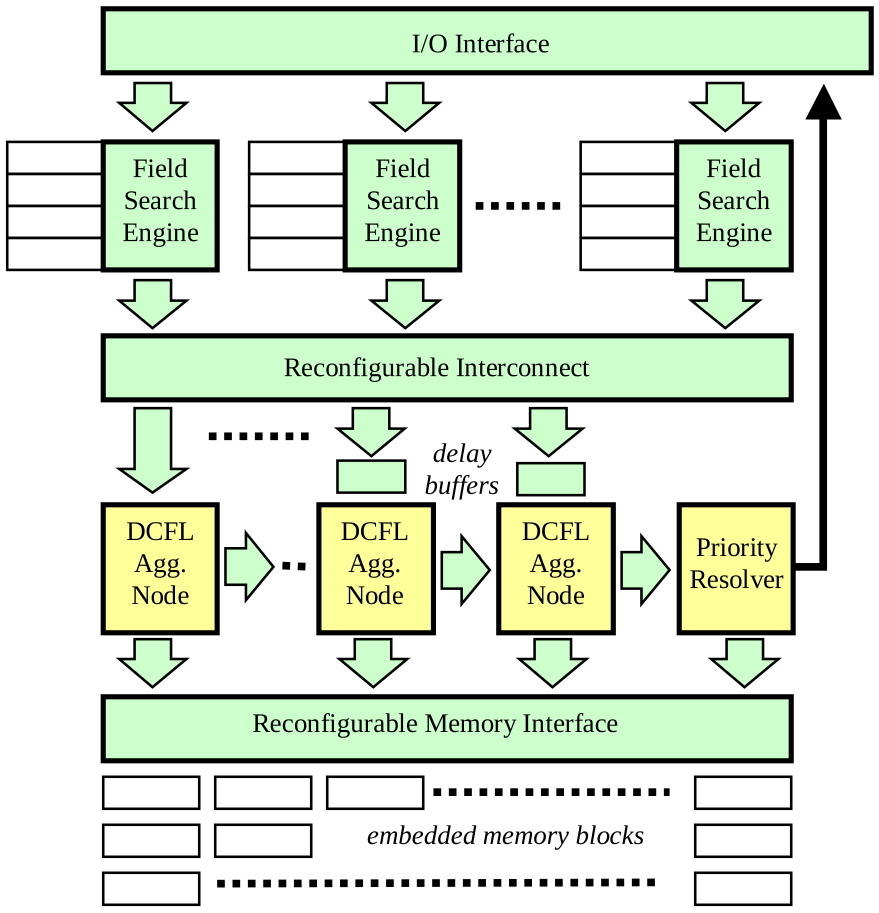
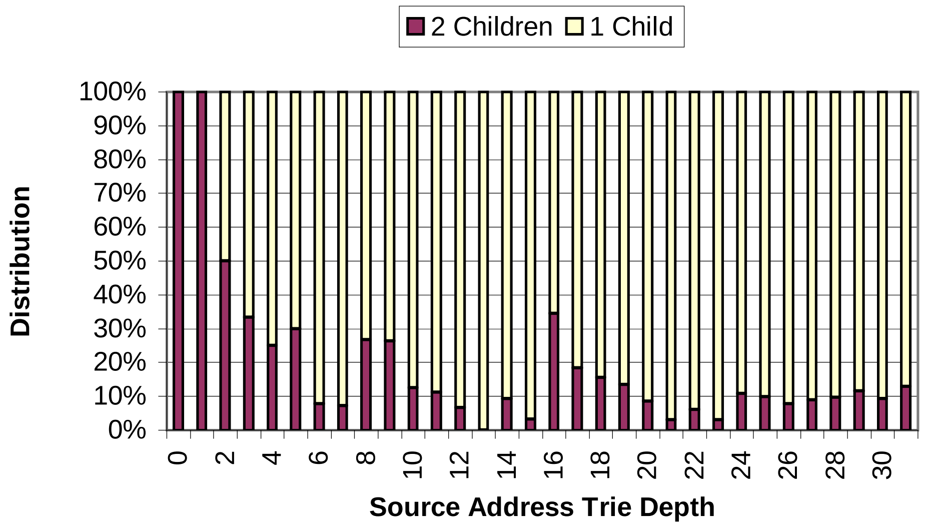


Figure 8.1: Potential implementation architecture for *Distributed Crossproducting of Field Labels*.

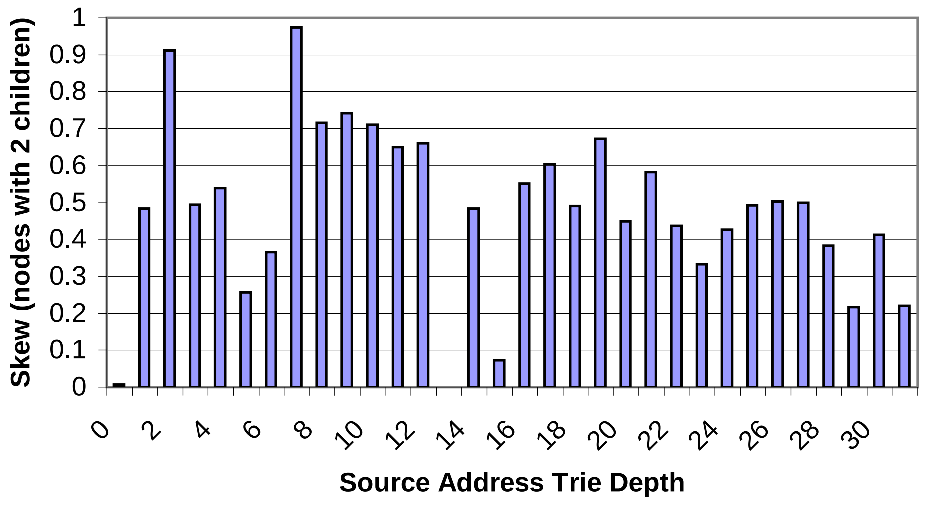
Appendix A

Additional Data from Real Filter Sets

The following figures are a supplement to the data presented in Chapter 5.

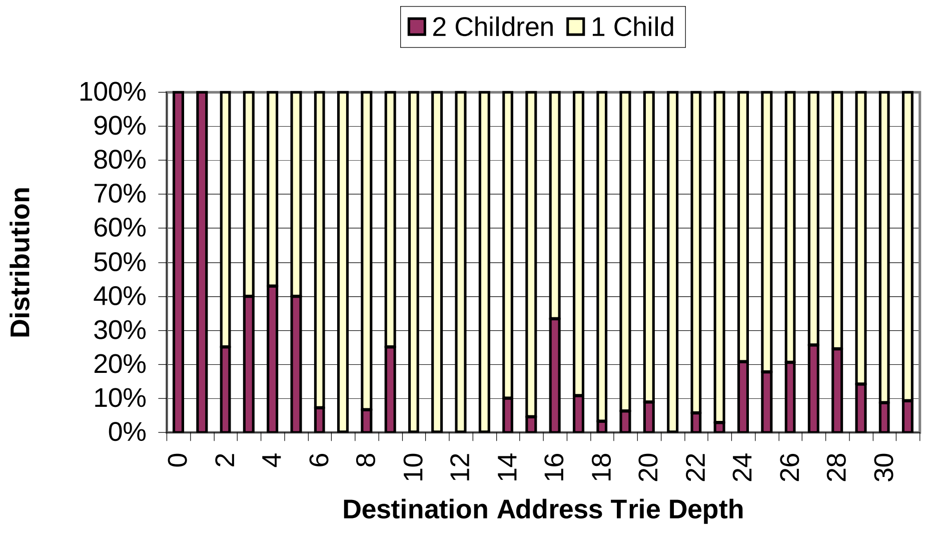


(a) Source address branching probability; average per level.

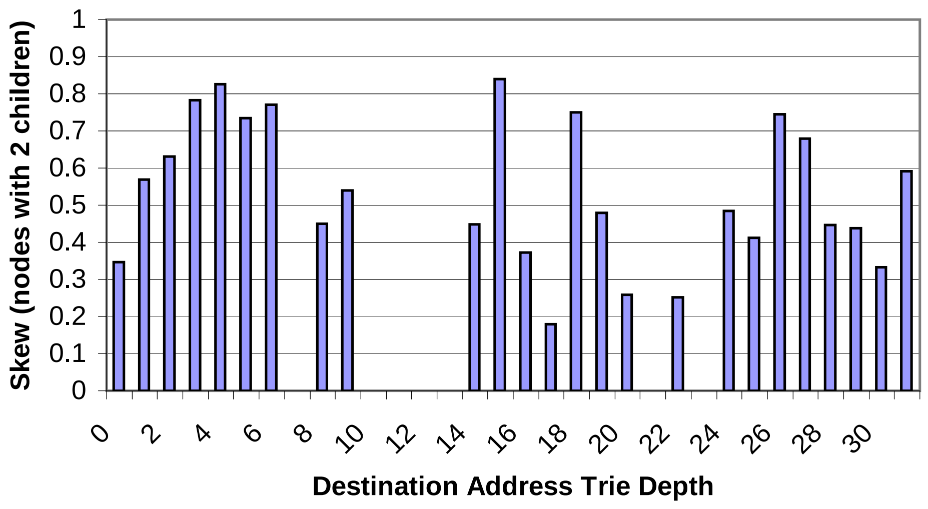


(b) Source address skew; average per level for nodes with two children.

Figure A.1: Source address branching probability and skew for filter set ipc1.

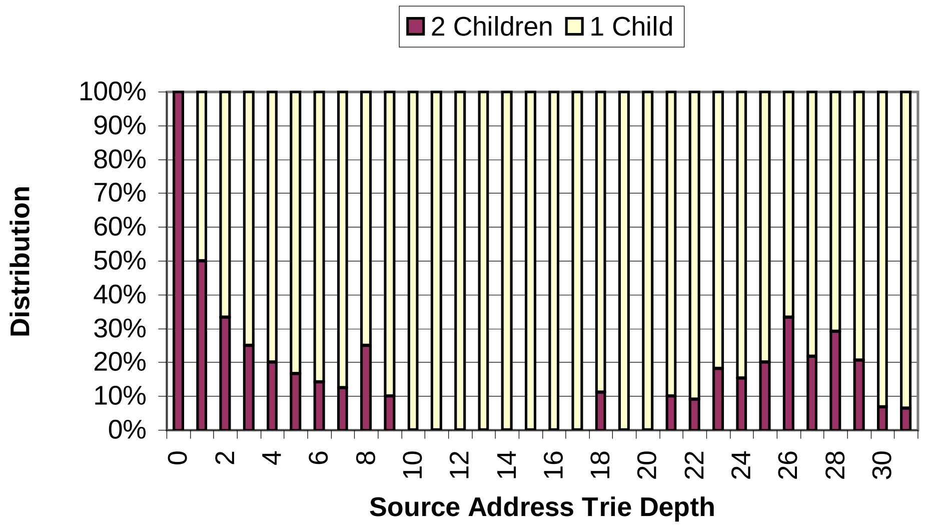


(a) Destination address branching probability; average per level.

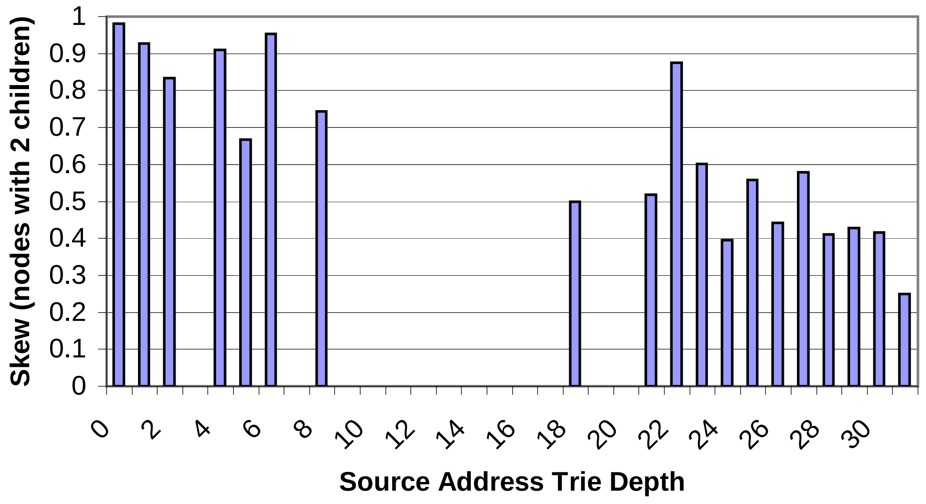


(b) Destination address skew; average per level for nodes with two children.

Figure A.2: Destination address branching probability and skew for filter set ipc1.

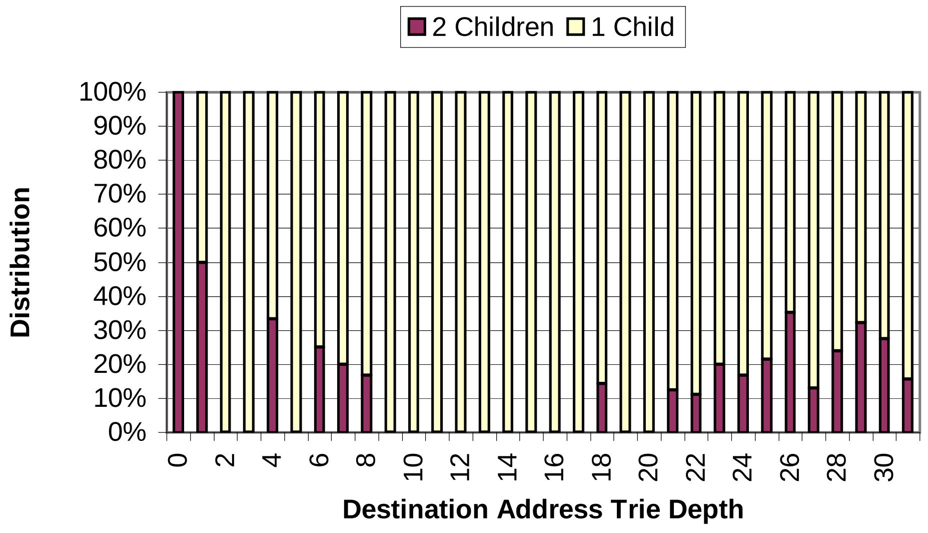


(a) Source address branching probability; average per level.

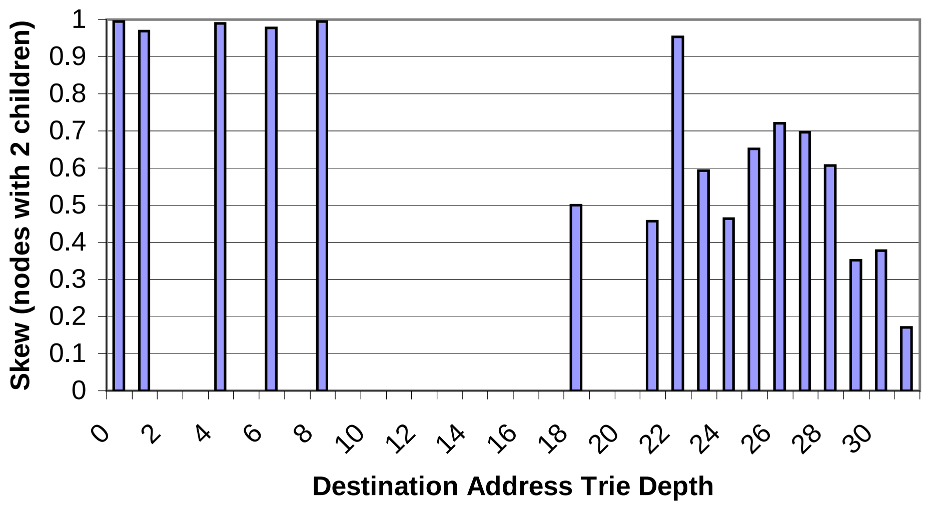


(b) Source address skew; average per level for nodes with two children.

Figure A.3: Source address branching probability and skew for filter set fw1.

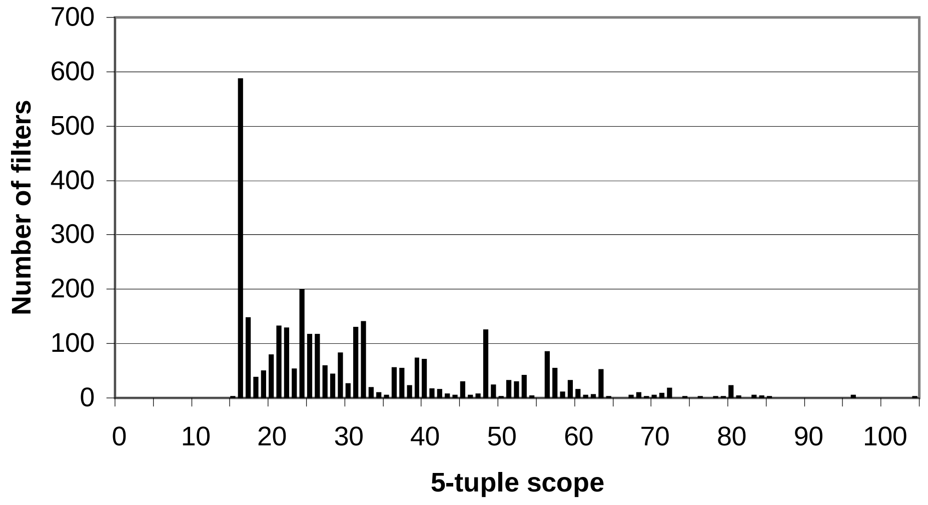


(a) Destination address branching probability; average per level.

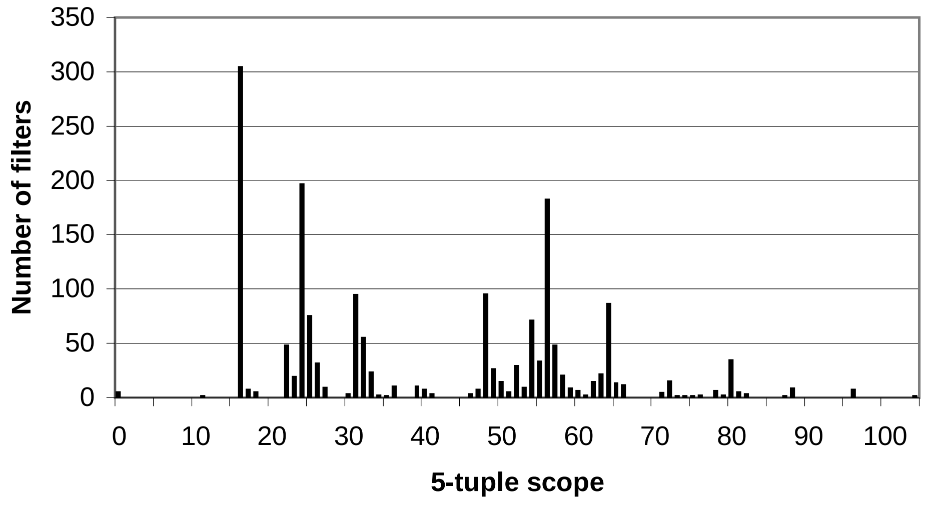


(b) Destination address skew; average per level for nodes with two children.

Figure A.4: Destination address branching probability and skew for filter set fw1.

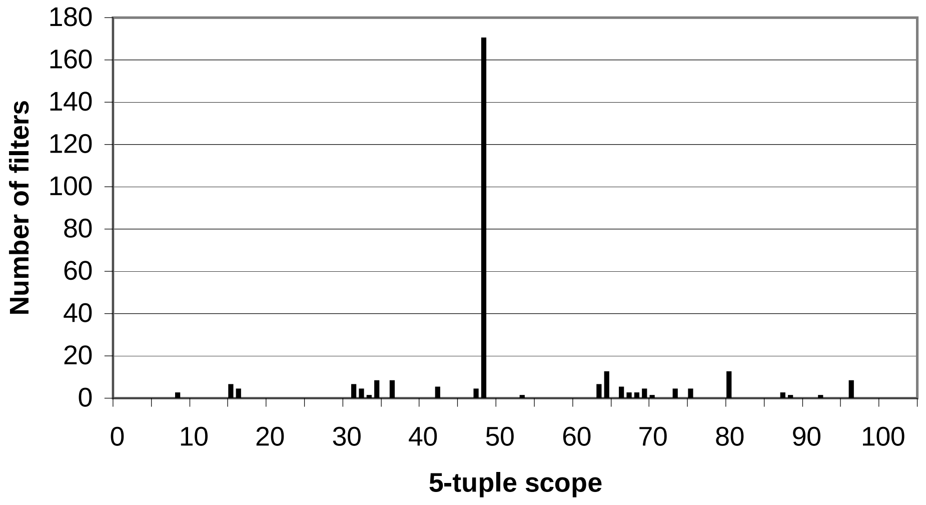


(a) acl4, $\mu = 30.9$, $\sigma = 15.1$

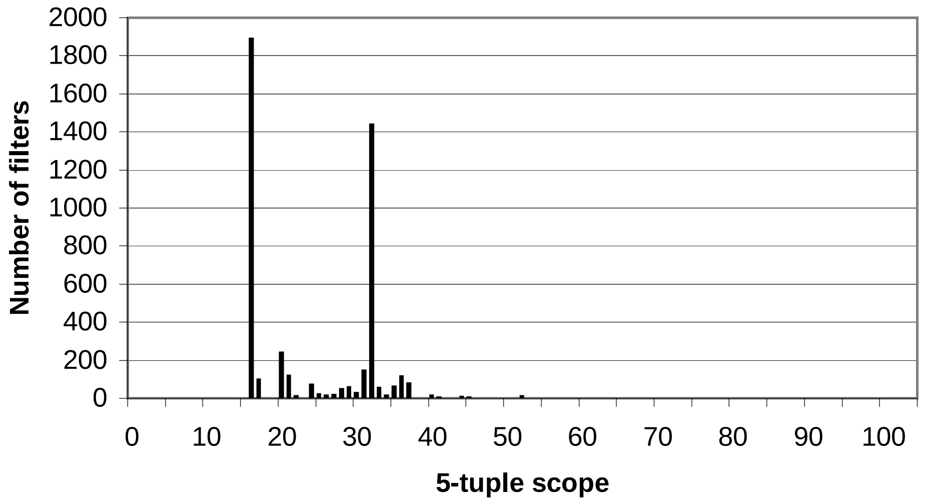


(b) ipc1, $\mu = 39.7$, $\sigma = 19.5$

Figure A.5: Distribution of 5-tuple scope for filters in filter sets acl4 and ipc1.



(a) fw1, $\mu = 51.2$, $\sigma = 15.7$



(b) fw5, $\mu = 55.8$, $\sigma = 17.0$

Figure A.6: Distribution of 5-tuple scope for filters in filter sets fw1 and fw5.

References

[1] D. Clark, “The Design Philosophy of the DARPA Internet Protocols,” 1988.

[2] “Internet Domain Survey.” Internet Systems Consortium, January 2004.

[3] ClickZ, “Population Explosion!.” http://www.clikz.com, May 2004. ClickZ Stats.

[4] Reuters, “U.S. online content spending rises in 2003,” May 2004. San Francisco.

[5] CNET, “Online holiday apending up, up and away.” http://news.com.com, December 2003.

[6] Linley, “Search Engine Market Maturing.” The Linley Wire, Volume 4, Issue 11, June 2004.

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

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

[9] United States Department of Defense, “Next-Generation Internet Protocol to Enable Net-Centric Operations.” News Release No. 413-03, June 2003.

[10] D. Cheriton, “Internet Architecture - It’s Future and Why it matters.” Keynote Address, ACM SIGCOMM, August 2003.

[11] W. N. Eatherton, “Hardware-Based Internet Protocol Prefix Lookups.” thesis, Washington University in St. Louis, 1998. Available at `http://www.arl.wustl.edu/`.

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

[13] T. Cormen, C. Leiserson, and R. Rivest, *Introduction to Algorithms*. McGraw-Hill Book Company, 1990.

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

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

[16] 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.

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

[18] K. Sklower, "A tree-based routing table for Berkeley Unix," tech. rep., University of California, Berkeley, 1993.

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

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

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

[22] D. E. Taylor, J. S. Turner, J. W. Lockwood, T. S. Sproull, and D. B. Parlour, "Scalable IP Lookup for Internet Routers," *IEEE Journal on Selected Areas in Communications*, vol. 21, pp. 522–534, May 2003.

[23] 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.

[24] 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.

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

[26] C.-F. Su, "High-Speed Packet Classification Using Segment Tree," in *Proceedings of IEEE Globecom*, 2000.

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

[28] 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.

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

[30] F. P. Preparata and M. I. Shamos, *Computational Geometry: An Introduction*. Texts and Monographs in Computer Science, Springer-Verlag, 1985.

[31] R. E. Tarjan, *Data Structures and Network Algorithms*. CBMS-NSF 44, Society for Industrial and Applied Mathematics, 1983.

[32] E. Spitznagel, D. Taylor, and J. Turner, "Packet Classification Using Extended TCAMs," in *Proceedings of IEEE International Conference on Network Protocols (ICNP)*, 2003.

[33] S. Choi, J. Dehart, R. Keller, F. Kuhns, J. Lockwood, P. Pappu, J. Parwatikar, W. D. Richard, E. Spitznagel, D. Taylor, J. Turner, , and K. Wong, "Design of a High Performance Dynamically Extensible Router," in *DARPA Active Networks Conference and Exposition*, May 2002.

[34] J. S. Turner, "Gigabit Technology Distribution Program." `http://www.arl.wustl.-edu/gigabitkits/kits.html`, Aug. 1999.

[35] J. Turner, T. Chaney, A. Fingerhut, and M. Flucke, "Design of a Gigabit ATM Switch," in *In Proceedings of Infocom 97*, Mar. 1997.

[36] S. Choi, J. Dehart, R. Keller, J. W. Lockwood, J. Turner, and T. Wolf, "Design of a flexible open platform for high performance active networks," in *Allerton Conference*, (Champaign, IL), 1999.

[37] J. W. Lockwood, J. S. Turner, and D. E. Taylor, "Field programmable port extender (FPX) for distributed routing and queuing," in *ACM International Symposium on Field Programmable Gate Arrays (FPGA'2000)*, (Monterey, CA, USA), pp. 137–144, Feb. 2000.

[38] J. W. Lockwood, N. Naufel, J. S. Turner, and D. E. Taylor, "Reprogrammable Network Packet Processing on the Field Programmable Port Extender (FPX)," in *ACM International Symposium on Field Programmable Gate Arrays (FPGA'2001)*, (Monterey, CA, USA), pp. 87–93, Feb. 2001.

[39] P. Newman *et al.*, "Transmission of flow labelled IPv4 on ATM data links." Internet RFC 1954, May 1996.

[40] T. S. Sproull, J. W. Lockwood, and D. E. Taylor, "Control and Configuration Software for a Reconfigurable Networking Hardware Platform," in *FCCM'02: 2002 IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2002.

[41] J. M. Anderson, M. Ilyas, and S. Hsu, "Distributed network management in an internet environment," in *Globecom'97*, vol. 1, (Pheonix, AZ), pp. 180–184, Nov. 1997.

[42] “Internet Routing Table Statistics.” `http://www.merit.edu/ipma-/routing_table/`, May 2001.

[43] D. Taylor, A. Chandra, Y. Chen, S. Dharmapurikar, J. Lockwood, W. Tang, and J. Turner, “System-on-Chip Packet Processor for an Experimental Network Services Platform,” in *IEEE Globecom*, December 2003.

[44] Xilinx, “Virtex-II Pro Platform FPGAs: Introduction and Overview.” DS083-1 (v3.0), December 2003.

[45] P. Newman, G. Minshall, and L. Huston, “IP Switching and Gigabit Routers.” IEEE Communications Magazine, January 1997.

[46] G. Chandranmenon and G. Varghese, “Trading Packet Headers for Packet Processing,” *IEEE/ACM Transactions on Networking*, vol. 4, pp. 141–152, April 1996.

[47] Y. Rekhter, B. Davie, D. Katz, E. Rosen, and G. Swallow, “Cisco Systems’ Tag Switching Architecture Overview.” RFC 2105, February 1997.

[48] E. Rosen, A. Viswanathan, and R. Callon, “Multiprotocol Label Switching Architecture.” RFC 3031, January 2001.

[49] SiberCore Technologies Inc., “SiberCAM Ultra-2M SCT2000.” Product Brief, 2000.

[50] P. Gupta and N. McKeown, “Packet Classification on Multiple Fields,” in *ACM Sigcomm*, August 1999.

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

[52] T. V. Lakshman and D. Stiliadis, “High-Speed Policy-based Packet Forwarding Using Efficient Multi-dimensional Range Matching,” in *ACM SIGCOMM‘98*, September 1998.

[53] V. Srinivasan, S. Suri, G. Varghese, and M. Waldvogel, “Fast and Scalable Layer Four Switching,” in *ACM Sigcomm*, June 1998.

[54] F. Baboescu and G. Varghese, “Scalable Packet Classification,” in *ACM Sigcomm*, August 2001.

[55] R. A. Kempke and A. J. McAuley, “Ternary CAM Memory Architecture and Methodology.” United States Patent 5,841,874, November 1998. Motorola, Inc.

[56] G. Gibson, F. Shafai, and J. Podaima, “Content Addressable Memory Storage Device.” United States Patent 6,044,005, March 2000. SiberCore Technologies, Inc.

[57] R. K. Montoye, "Apparatus for Storing "Don't Care" in a Content Addressable Memory Cell." United States Patent 5,319,590, June 1994. HaL Computer Systems, Inc.

[58] F. Baboescu, S. Singh, and G. Varghese, "Packet Classification for Core Routers: Is there an alternative to CAMs?," in *IEEE Infocom*, 2003.

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

[60] D. E. Taylor and J. S. Turner, "Scalable Packet Classification using Distributed Crossproducting of Field Labels," tech. rep., Department of Computer Science and Engineering, Washington University in Saint Louis, 2004.

[61] SiberCore Technologies Inc., "SiberCAM Ultra-18M SCT1842." Product Brief, 2002.

[62] Micron Technology Inc., "Harmony TCAM 1Mb and 2Mb." Datasheet, January 2003.

[63] Micron Technology Inc., "36Mb DDR SIO SRAM 2-Word Burst." Datasheet, December 2002.

[64] 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.

[65] J. van Lunteren, "Searching very large routing tables in wide embedded memory," in *Proceedings of IEEE Globecom*, vol. 3, pp. 1615–1619, November 2001.

[66] V. Srinivasan, S. Suri, and G. Varghese, "Packet classification using tuple space search," in *SIGCOMM 99*, pp. 135–146, 1999.

[67] V. Srinivasan, "A Packet Classification and Filter Management System." Microsoft Research, 2001.

[68] P. Warkhede, S. Suri, and G. Varghese, "Fast Packet Classification for Two-Dimensional Conflict-Free Filters," in *IEEE Infocom*, 2001.

[69] A. Hari, S. Suri, and G. Parulkar, "Detecting and Resolving Packet Filter Conflicts," in *Proceedings of IEEE Infocom*, 2000.

[70] J. L. Hennessy and D. A. Patterson, *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 2 ed., 1996.

[71] F. Chang, K. Li, and W. chang Feng, "Approximate Packet Classification Caching," Tech. Rep. CSE-03-002, OGI School of Science and Engineering at OHSU, 2003.

[72] M. M. I. Chvets, "Multi-zone Caches for Accelerating IP Routing Table Lookups," in *Proceedings of High-Performance Switching and Routing*, 2002.

[73] K. Li, F. Chang, D. Berger, and W. chang Fang, “Architectures for Packet Classification Caching,” in *Proceedings of IEEE ICON*, 2003.

[74] IBM Blue Logic, “Embedded SRAM Selection Guide,” November 2002.

[75] Micron Technology Inc., “256Mb Double Data Rate (DDR) SDRAM.” Datasheet, October 2002.

[76] P. Crowley, M. Franklin, H. Hadimioglu, and P. Onufryk, *Network Processor Design: Issues and Practices*, vol. 1. Morgan Kaufmann, 2002.

[77] M. E. Kounavis, A. Kumar, H. Vin, R. Yavatkar, and A. T. Campbell, “Directions in Packet Classification for Network Processors,” in *Second Workshop on Network Processors (NP2)*, February 2003.

[78] N. Shah, “Understanding network processors,” Tech. Rep. Version 1.0, EECS, University of California, Berkeley, September 2001.

[79] Cisco, “CiscoWorks VPN/Security Management Solution,” tech. rep., Cisco Systems, Inc., 2004.

[80] Lucent, “Lucent Security Management Server: Security, VPN, and QoS Management Solution,” tech. rep., Lucent Technologies Inc., 2004.

[81] J. Postel, “Transmission Control Protocol.” RFC 793, September 1981.

[82] J. Postel, “User Datagram Protocol.” RFC 768, August 1980.

[83] J. Postel, “Internet Control Message Protocol.” RFC 792, September 1981.

[84] D. Farinacci, T. Li, S. Hanks, D. Meyer, and P. Traina, “General Routing Encapsulation.” RFC 2784, March 2000.

[85] J. Moy, “OSPF Version 2.” RFC 2784, July 1997.

[86] Cisco, “Enhanced Interior Gateway Routing Protocol (EIGRP).” white paper, 2003. Cisco Systems Inc.

[87] S. Kent and R. Atkinson, “IP Encapsulating Security Payload (ESP).” RFC 2406, November 1998.

[88] S. Kent and R. Atkinson, “IP Authentication Header.” RFC 2402, November 1998.

[89] C. Perkins, “IP Encapsulation within IP.” RFC 2003, October 1996.

[90] C. Bormann, et. al., “RObust Header Compression (ROHC): Framework and four profiles: RTP, UDP, ESP, and uncompressed.” RFC 3095, July 2001. IETF Network Working Group.

[91] "IPv6 Operational Report." `http://net-stats.ipv6.tilab.com/bgp/-bgp-table-snapshot.txt/`, February 2003.

[92] R. Hinden and S. Deering, "Internet Version 6 (IPv6) Addressing Architecture." RFC 3513, April 2003.

[93] R. Hinden, S. Deering, and E. Nordmark, "IPv6 Global Unicast Address Format." Internet Draft, February 2003.

[94] IANA, "IPv6 Address Allocation and Assignment Policy." http://www.iana.org/ipaddress/ipv6-allocation-policy-26jun02, June 2002.

[95] V. Paxson, G. Almes, J. Mahdavi, and M. Mathis, "Framework for ip performance metrics." RFC 2330, May 1998.

[96] S. Bradner, "Benchmarking Terminology for Network Interconnect Devices." RFC 1242, July 1991.

[97] S. Bradner and J. McQuaid, "Benchmarking Methodology for Network Interconnect Devices." RFC 2544, March 1999.

[98] G. Trotter, "Terminology for Forwarding Information Base (FIB) based Router Performance." RFC 3222, December 2001.

[99] G. Trotter, "Methodology for Forwarding Information Base (FIB) based Router Performance." Internet Draft, January 2002.

[100] D. Newman, "Benchmarking Terminology for Firewall Performance." RFC 2647, August 1999.

[101] B. Hickman, D. Newman, S. Tadjudin, and T. Martin, "Benchmarking Methodology for Firewall Performance." RFC 3511, April 2003.

[102] P. Chandra, F. Hady, and S. Y. Lim, "Framework for Benchmarking Network Processors." Network Processing Forum, 2002.

[103] F. Baboescu and G. Varghese, "Fast and Scalable Conflict Detection for Packet Classifiers," in *Proceedings of IEEE International Conference on Network Protocols (ICNP)*, 2002.

[104] V. Sahasranaman and M. Buddhikot, "Comparative Evaluation of Software Implementations of Layer 4 Packet Classification Schemes," in *Proceedings of IEEE International Conference on Network Protocols*, 2001.

[105] Wikipedia, "Pareto distribution." Wikipedia, The Free Encyclopedia, April 2004. http://en.wikipedia.org/wiki/Pareto_distribution.

[106] G. Narlikar, A. Basu, and F. Zane, “CoolCAMs: Power-Efficient TCAMs for Forwarding Engines,” in *Proc. of Infocom*, 2003.

[107] Xilinx, “Virtex-II Platform FPGAs: Introduction and Overview.” DS031-1 (v2.0), August 2003.

[108] D. Shah and P. Gupta, “Fast incremental updates on ternary-cams for routing lookups and packet classification,” in *Hot Interconnects (HotI-8)*, p. 6.1, Aug. 2000.

[109] “Open Network Laboratory: A Resource for Networking Researchers.” http://www.arl.wustl.edu/arl/projects/onl/. Applied Research Laboratory, Washington University in Saint Louis.

Vita

David Edward Taylor

Date of Birth	December 11, 1975
Place of Birth	Saint Louis, Missouri, United States of America
Degrees	**Washington University in Saint Louis** Master of Science in Electrical Engineering, 2002 Master of Science in Computer Engineering, 2002 Bachelor of Science in Electrical Engineering cum laude, 1998 Bachelor of Science in Computer Engineering cum laude, 1998
Experience	IBM Zurich Research Laboratory, Network Processor Hardware Group, Summer 2002 Applied Research Laboratory, Washington University in Saint Louis, 1999 – 2004 Mentor Graphics Higher Education Project, Summer 1998
Professional Societies	Institute of Electrical and Electronics Engineers (IEEE) IEEE Communications Society IEEE Computer Society Association for Computing Machines (ACM)
Professional Activities	Reviewer for IEEE/ACM Transactions on Networking Reviewer for IEEE Journal on Selected Areas in Communications Reviewer for Computer Networks (Elsevier) Reviewer for IEEE Micro Reviewer for IEEE Communications Letters Reviewer for IEEE Infocom 2003, 2004 Reviewer for IEEE Globecom 2003, 2004
Scholarships & Awards	Graduate Student Representative to the Board of Trustees (2003 – 2004) Research Assistantship (January 1999 – present) Dean’s Honorary Scholarship (1994 – 1998) Missouri Higher Education Academic Scholarship (1994 – 1998) Graduate of the LeaderShape Institute (Summer 1997) Eta Kappa Nu (International Electrical Engineering Honor Fraternity) Eagle Scout, Boy Scouts of America

Journal Publications

David E. Taylor, Jonathan S. Turner, John W. Lockwood, Todd Sproull, David B. Parlour, *Scalable IP Lookup for Internet Routers*, IEEE Journal on Selected Areas in Communications, May 2003, Volume 21, Number 4.

David E. Taylor, Jonathan S. Turner, John W. Lockwood, Edson L. Horta *Dynamic Hardware Plugins (DHP): Exploiting Reconfigurable Hardware for High-Performance Programmable Routers*, Computer Networks, February 2002, Volume 38, Issue 3, pp. 295-310, Elsevier Science.

William D. Richard, David E. Taylor, David M. Zar, *A Capstone Computer Engineering Design Course*, IEEE Transactions on Education, November 1999, Volume 42, Number 4, pp. 288-294.

Conference Publications

David E. Taylor, Jonathan S. Turner, *Scalable Packet Classification using Distributed Crossproducting of Field Labels*, ACM Sigcomm'04 Student Poster Session, 8/04.

David Taylor, Alex Chandra, Yuhua Chen, Sarang Dharmapurikar, John Lockwood, Wenjing Tang, Jonathan Turner, *System-on-Chip Packet Processor for an Experimental Network Services Platform*, IEEE Globecom'03, December 1-5, 2003, San Francisco, CA.

Ed Spitznagel, David Taylor, Jonathan Turer, *Packet Classification Using Extended TCAMs*, 11th IEEE International Conference on Network Protocols (ICNP), November 4-7, 2003, Atlanta, GA.

Sarang Dharmapurikar, Praveen Krishnamurthy, David E. Taylor, *Longest Prefix Matching using Bloom Filters*, ACM SIGCOMM'03, August 25-29, 2003, Karlsruhe, Germany.

David E. Taylor, John W. Lockwood, Todd Sproull, Jonathan S. Turner, David B. Parlour, *Scalable IP Lookup for Programmable Routers*, IEEE INFOCOM 2002: 21st Annual Joint Conference of the IEEE Computer and Communications Societies, New York, NY, 6/02.

Edson L. Horta, John W. Lockwood, David E. Taylor, David Parlour, *Dynamic Hardware Plugins in an FPGA with Partial Run-time Reconfiguration*, Design Automation Conference (DAC), New Orleans, LA, 6/02.

Sumi Choi, John Dehart, Ralph Keller, Fred Kuhns, John Lockwood, Prashanth Pappu, Jyoti Parwatikar, W. David Richard, Ed Spitznagel, David Taylor, Jonathan Turner, and Ken Wong, *Design of a High Performance Dynamically Extensible Router* Proceedings of the DARPA Active Networks Conference and Exposition, 5/02.

Todd S. Sproull, John W. Lockwood, David E. Taylor, *Control and Configuration Software for a Reconfigurable Networking Hardware Platform*, FCCM'02: 2002 IEEE Symposium on Field-Programmable Custom Computing Machines, Napa, CA, 4/02.

David E. Taylor, Jonathan S. Turner, John W. Lockwood, *Dynamic Hardware Plugins (DHP): Exploiting Reconfigurable Hardware for High-Performance Programmable Routers*, IEEE OPENARCH 2001: 4th IEEE Conference on Open Architectures and Network Programming, Anchorage, AK, 4/01.

John W. Lockwood, Naji Naufel, David E. Taylor, Jon S. Turner, *Reprogrammable Network Packet Processing on the Field Programmable Port Extender (FPX)*, FPGA 2001: Ninth ACM International Symposium on Field-Programmable Gate Arrays, Monterey, CA, 2/01.

William D. Richard, David E. Taylor, *Development of an FPGA-Based South Bridge using Spectrum and ModelSim*, Mentor Graphics User's Group International Conference, Portland, OR, 10/00.

John W. Lockwood, Jon S. Turner, David E. Taylor, *Field Programmable Port Extender (FPX) for Distributed Routing and Queueing*, FPGA 2000: Eighth ACM International Symposium on Field-Programmable Gate Arrays, Monterey, CA, 2/00.

David E. Taylor, David M. Zar, *Developing an FPGA Workflow to Ease Novice and Experienced Designers*, Mentor Graphics User's Group International Conference, Portland, OR, 10/98. **Awarded Best Paper for University/Research Track**.

Technical Reports

David E. Taylor, Jonathan S. Turner, *Scalable Packet Classification using Distributed Crossproducting of Field Labels*, WUCSE-2004-38, 6/04.

David E. Taylor, Jonathan S. Turner, *ClassBench: A Packet Classification Benchmark*, WUCSE-2004-28, 5/04.

David E. Taylor, *Survey & Taxonomy of Packet Classification Techniques*, WUCSE-2004-24, 5/04.

David E. Taylor, Jonathan S. Turner, *Towards a Packet Classification Benchmark*, WUCSE-2003-42, 5/03.

David Taylor, Alex Chandra, Yuhua Chen, Sarang Dharmapurikar, John Lockwood, Wenjing Tang, Jonathan Turner, *System-on-Chip Packet Processor for an Experimental Network Services Platform*, WUCSE-2003-22, 3/03.

David E. Taylor, John W. Lockwood, Todd Sproull, Jonathan S. Turner, David B. Parlour, *Scalable IP Lookup for Programmable Routers*, WUCS-01-33, 10/01.

John D. DeHart, William D. Richard, Edward W. Spitznagel, David E. Taylor, *The Smart Port Card: An Embedded Unix Processor Architecture for Network Management and Active Networking*, WUCS-01-18, 8/01.

David E. Taylor, John W. Lockwood, Naji Naufel, *RAD Module Infrastructure of the Field Programmable Port Extender (FPX) Version 2.0*, WUCS-TM-01-16, 7/01.

David E. Taylor, John W. Lockwood, Sarang Dharmapurikar, *Generalized RAD Module Interface Specification of the Field Programmable Port Extender (FPX) Version 2.0*, WUCS-TM-01-15, 7/01.

August 2004