ALGORITHMS FOR ROUTING LOOKUPS AND PACKET CLASSIFICATION

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Pankaj Gupta

December 2000

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Prof. Nicholas W. McKeown
(Principal Adviser)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Prof. Balaji Prabhakar
(Co-adviser)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Prof. Mark Horowitz

Approved for the University Committee on Graduate Studies:

To my mother&land

Abstract

The work presented in this thesis is motivated by the twin goals of increasing the capacity and the flexibility of the Internet. The Internet is comprised of packet-processing nodes, called routers, that route packets towards their destinations, and physical links that transport packets from one router to another. Owing to advances in optical technologies, such as Wavelength Division Multiplexing, the data rates of links have increased rapidly over the years. However, routers have failed to keep up with this pace because they must perform expensive per-packet processing operations.

Every router is required to perform a forwarding decision on an incoming packet to determine the packet's next-hop router. This is achieved by looking up the destination address of the incoming packet in a forwarding table. Besides increased packet arrival rates because of higher speed links, the complexity of the forwarding lookup mechanism and the large size of forwarding tables have made routing lookups a bottleneck in the routers that form the core of the Internet. The first part of this thesis describes fast and efficient routing lookup algorithms that attempt to overcome this bottleneck.

The second part of this thesis concerns itself with increasing the flexibility and functionality of the Internet. Traditionally, the Internet provides only a “best-effort” service, treating all packets going to the same destination identically, and servicing them in a first-come-first-served manner. However, Internet Service Providers are seeking ways to provide differentiated services (on the same network infrastructure) to different users based on their different requirements and expectations of quality from the Internet. For this, routers need to have the capability to distinguish and isolate traffic belonging to different flows. The ability to classify each incoming packet to determine the flow it belongs to is called packet classification, and could be based on an arbitrary number of fields in the packet header. The second part of this thesis highlights some of the issues in designing efficient packet classification algorithms, and describes novel algorithms that enable routers to perform fast packet classification on multiple fields.

Acknowledgments

First and foremost, I would like to thank my adviser, Nick McKeown, for his continued support and guidance throughout my Ph.D. Nick, I have learned a lot from you — whether in doing research, writing papers or giving presentations, yours has been an inspiring influence.

I am extremely grateful to my co-adviser, Balaji Prabhakar, for being a constant source of encouragement and advice. Balaji, I truly appreciate the time and respect you have given me over the years.

Several people have helped greatly in the preparation of this thesis. I am especially grateful to Mark Ross (Cisco Systems), Youngmi Joo and Midge Eisele for reading a draft of my entire thesis and giving me invaluable feedback. I also wish to acknowledge the immensely helpful feedback from Daniel Awduche (UUNET/Worldcom) and Greg Watson (PMC-Sierra) on Chapter 1, Srinivasan Venkatachary (Microsoft Research) on Chapter 2 and Lili Qiu (Cornell University) on Chapter 4. Darren Kerr (Cisco) kindly gave me access to the real-life classifier datasets that enabled me to evaluate the performance of two of the algorithms mentioned in this thesis.

I have had the pleasure of sharing office space at Gates 342 with the following people: Youngmi, Pablo, Jason, Amr, Rolf, Da, Steve and Adisak. Besides them, I appreciate the chance of being able to interact with some of the smartest people in Balaji's and Nick's research groups: Devavrat, Paul, Kostas, Rong, Sundar, Ramana, Elif, Mayank, Chandra. To all of you, my Stanford experience has been greatly enriched by your company.

There are several other amazing people whom I met while at Stanford, became friends with, and whom I will remember for a long long time. Aparna, Suraj, Shankar, Mohan, Krista, Noelle, Prateek, Sanjay, John, Sunay, Pankaj, Manish and Rohit — I thank you for your companionship, best wishes and support.

I feel privileged to have been a part of ASHA Stanford (an action group for basic education in India). To witness the quality of dedication, commitment and dynamism displayed by ASHA's members is a treat without a parallel.

Last, and certainly the most, I thank my family: my father T.C. Gupta, my mother Raj-Kumari, my sister Renu and the relatively new and welcome additions (my brother-in-law Ashish, and my cute niece Aashna) for their unfailing love, encouragement and support.

Contents

CHAPTER 3

CHAPTER 4

List of Tables

List of Figures

कर्मण्येवाधिकारस्ते मा फलेषु कदाचन ।
मा कर्मफलहेतुर्भूर् मा ते सङ्गोऽस्त्वकर्मणि ॥

You have the right to perform your prescribed
duty (Karma) but not to the fruits of action.
Never consider yourself the cause of the results
of your activities and never get attached to inaction.

Lord Krishna to Arjuna in Bhagvad-gita

CHAPTER 1

Introduction

The Internet is comprised of a mesh of routers interconnected by links. Communication among nodes on the Internet (routers and end-hosts) takes place using the Internet Protocol, commonly known as IP. IP datagrams (packets) travel over links from one router to the next on their way towards their final destination. Each router performs a forwarding decision on incoming packets to determine the packet's next-hop router.

The capability to forward packets is a requirement for every IP router [3]. Additionally, an IP router may also choose to perform special processing on incoming packets. Examples of special processing include filtering packets for security reasons, delivering packets according to a pre-agreed delay guarantee, treating high priority packets preferentially, and maintaining statistics on the number of packets sent by different networks. Such special processing requires that the router classify incoming packets into one of several *flows* — all packets of a flow obey a pre-defined rule and are processed in a similar manner by the router. For example, all packets with the same source IP address may be defined to form a flow. A flow could also be defined by specific values of the destination IP address and by specific protocol values. Throughout this thesis, we will refer to routers that classify packets into flows as *flow-aware* routers. On the other hand, flow-unaware routers treat each incoming packet individually and we will refer to them as *packet-by-packet* routers.

This thesis is about two types of algorithms: (1) algorithms that an IP router uses to decide *where* to forward packets next, and, (2) algorithms that a flow-aware router uses to *classify* packets into flows.[1] In particular, this thesis is about fast and efficient algorithms that enable routers to process many packets per second, and hence increase the capacity of the Internet.

This introductory chapter first describes the packet-by-packet router and the method it uses to make the forwarding decision, and then moves on to describe the flow-aware router and the method it uses to classify incoming packets into flows. Finally, the chapter presents the goals and metrics for evaluation of the algorithms presented later in this thesis.

1 Packet-by-packet IP router and route lookups

A packet-by-packet IP router is a special-purpose packet-switch that satisfies the requirements outlined in RFC 1812 [3] published by the Internet Engineering Task Force (IETF).[2] All packet-switches — by definition — perform two basic functions. First, a packet-switch must perform a forwarding decision on each arriving packet for deciding where to send it next. An IP router does this by looking up the packet's destination address in a forwarding table. This yields the address of the next-hop router[3] and determines the

[1] As explained later in this chapter, the algorithms in this thesis are meant for the router data-plane (i.e., the datapath of the packet), rather than the router control-plane which configures and populates the forwarding table.

[2] IETF is a large international community of network equipment vendors, operators, engineers and researchers interested in the evolution of the Internet Architecture. It comprises of groups working on different areas such as routing, applications and security. It publishes several documents, called RFCs (Request For Comments). An RFC either overviews an introductory topic, or acts as a standards specification document.

[3] A packet may be sent to multiple next-hop routers. Such packets are called multicast packets and are sent out on multiple egress ports. Unless explicitly mentioned, we will discuss lookups for unicast packets only.

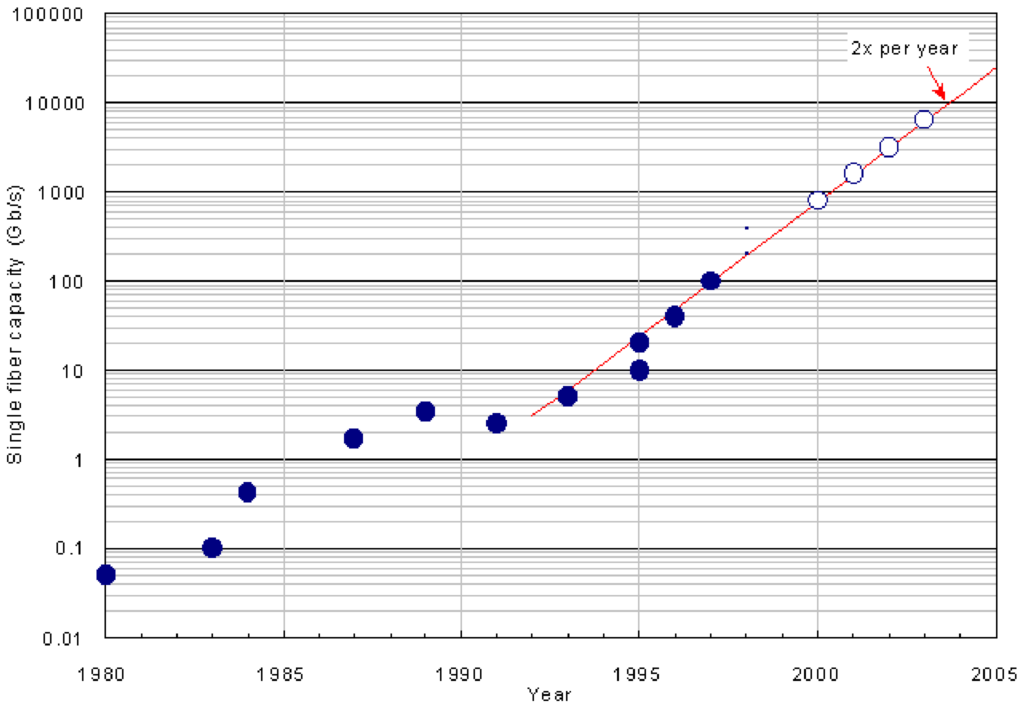


Figure 1.1 The growth in bandwidth per installed fiber between 1980 and 2005. (Source: Lucent Technologies.)

egress port through which the packet should be sent. This lookup operation is called a *route lookup* or an *address lookup* operation. Second, the packet-switch must transfer the packet from the ingress to the egress port identified by the address lookup operation. This is called *switching*, and involves physical movement of the bits carried by the packet.

The combination of route lookup and switching operations makes per-packet processing in routers a time consuming task. As a result, it has been difficult for the packet processing capacity of routers to keep up with the increased data rates of physical links in the Internet. The data rates of links have increased rapidly over the years to hundreds of gigabits per second in the year 2000 [133] — mainly because of advances in optical technologies such as WDM (Wavelength Division Multiplexing). Figure 1.1 shows the increase in bandwidth per fiber during the period 1980 to 2005, and Figure 1.2 shows the increase in

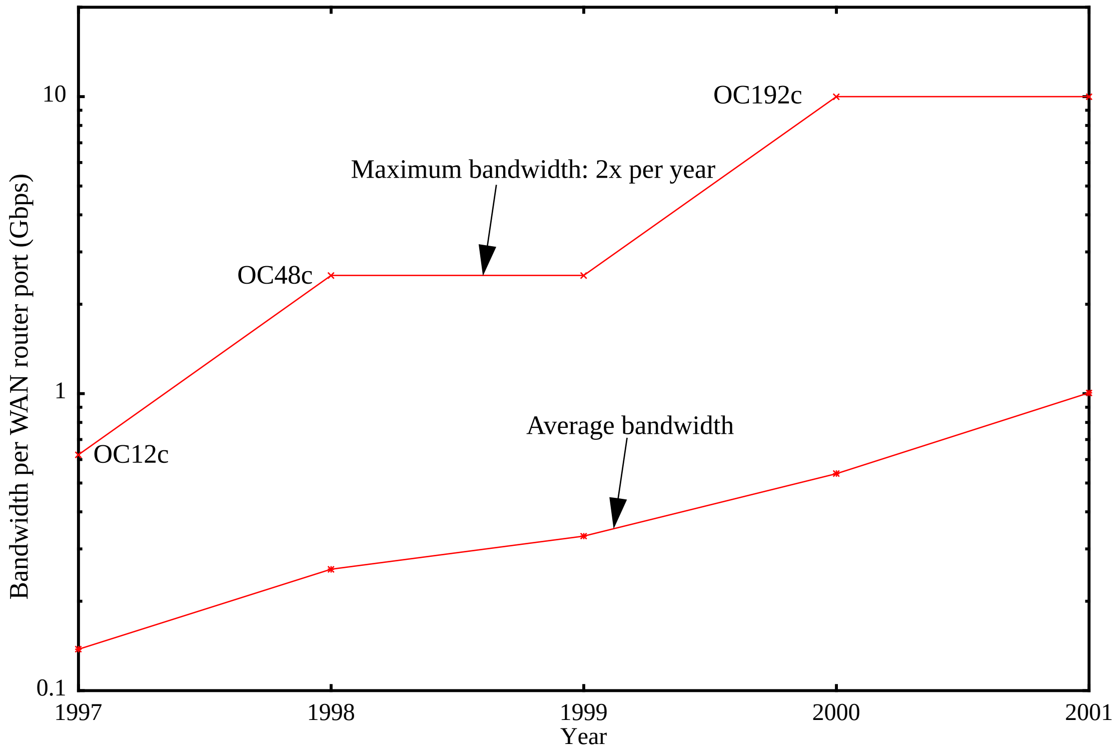


Figure 1.2 The growth in maximum bandwidth of a wide-area-network (WAN) router port between 1997 and 2001. Also shown is the average bandwidth per router port, taken over DS3, ATM OC3, ATM OC12, POS OC3, POS OC12, POS OC48, and POS OC192 ports in the WAN. (Data courtesy Dell'Oro Group, Portola Valley, CA)

the maximum bandwidth of a router port in the period 1997 to 2001. These figures highlight the gap in the data rates of routers and links — for example, in the year 2000, a data rate of 1000 Gbps is achievable per fiber, while the maximum bandwidth available is limited to 10 Gbps per router port. Figure 1.2 also shows the average bandwidth of a router port over all routers — this average is about 0.53 Gbps in the year 2000. The work presented in the first part of this thesis (Chapters 2 and 3) is motivated by the need to alleviate this mismatch in the speeds of routers and physical links — in particular, the need to perform route lookups at high speeds. High-speed switching [1][55][56][57][58][104] is an important problem in itself, but is not considered in this thesis.

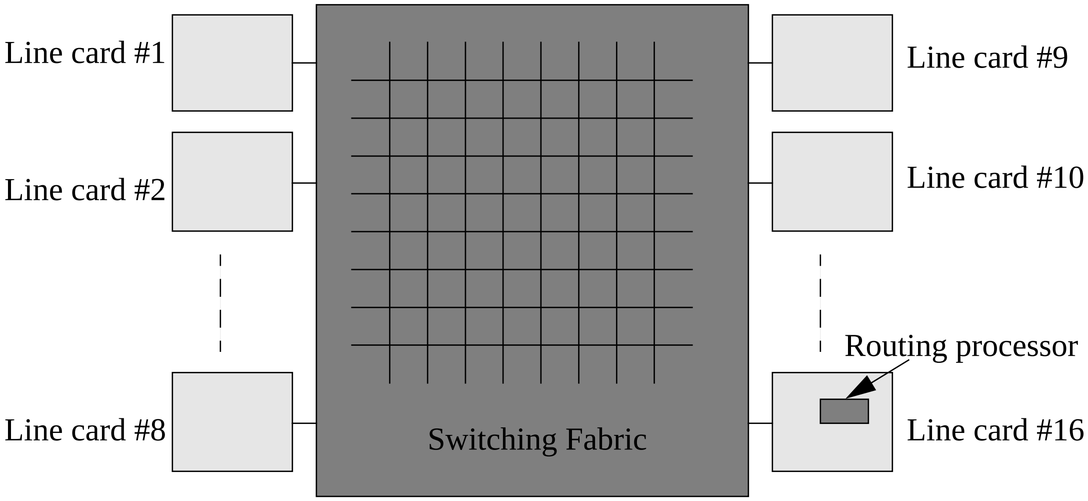


Figure 1.3 The architecture of a typical high-speed router.

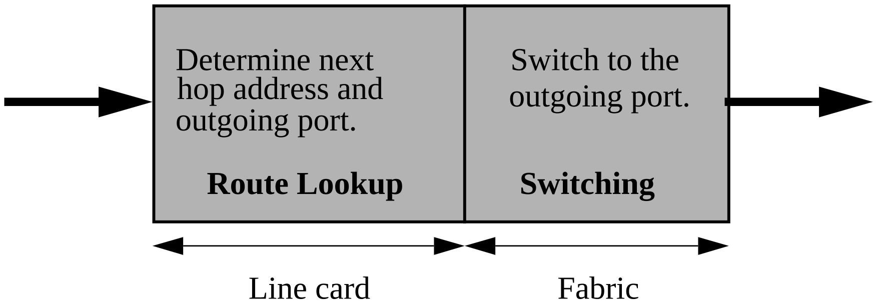

Determine next hop address and outgoing port.
Route Lookup
Switch to the outgoing port.
Switching
Line card
Fabric

Figure 1.4 Datapath of a packet through a packet-by-packet router.

1.1 Architecture of a packet-by-packet router

Figure 1.3 shows a block diagram of the architecture of a typical high speed router. It consists of one line card for each port and a switching fabric (such as a crossbar) that interconnects all the line cards. Typically, one of the line cards houses a processor functioning as the central controller for the router. The path taken by a packet through a packet-by-packet router is shown in Figure 1.4 and consists of two main functions on the packet: (1) performing route lookup based on the packet's destination address to identify the outgoing port, and (2) switching the packet to the output port.

The routing processor in a router performs one or more routing protocols such as RIP [33][51], OSPF [65] or BGP [80] by exchanging protocol messages with neighboring routers. This enables it to maintain a *routing table* that contains a representation of the network topology state information and stores the current information about the best known paths to destination networks. The router typically maintains a version of this routing table in all line cards so that lookups on incoming packets can be performed locally on each line card, without loading the central processor. This version of the central processor's routing table is what we have been referring to as the line card's *forwarding table* because it is directly used for packet forwarding. There is another difference between the routing table in the processor and the forwarding tables in the line cards. The processor's routing table usually keeps a lot more information than the forwarding tables. For example, the forwarding table may only keep the outgoing port number, address of next-hop, and (optionally) some statistics with each route, whereas the routing table may keep additional information: e.g., time-out values, the actual paths associated with the route, etc.

The routing table is dynamic — as links go down and come back up in various parts of the Internet, routing protocol messages may cause the table to change continuously. Changes include addition and deletion of prefixes, and the modification of next-hop information for existing prefixes. The processor communicates these changes to the line card to maintain up-to-date information in the forwarding table. The need to support routing table updates has implications for the design of lookup algorithms, as we shall see later in this thesis.

1.2 Background and definition of the route lookup problem

This section explains the background of the route lookup operation by briefly describing the evolution of the Internet addressing architecture, and the manner in which this impacts the complexity of the lookup mechanism. This leads us to the formal definition of the lookup problem, and forms a background to the lookup algorithms presented thereafter.

1.2.1 Internet addressing architecture and route lookups

In 1993, the Internet addressing architecture changed from *class-based addressing* to today's *classless addressing* architecture. This change resulted in an increase in the complexity of the route lookup operation. We first briefly describe the structure of IP addresses and the route lookup mechanism in the original class-based addressing architecture. We then describe the reasons for the adoption of classless addressing and the details of the lookup mechanism as performed by Internet routers.

IP version 4 (abbreviated as IPv4) is the version of Internet Protocol most widely used in the Internet today. IPv4 addresses are 32 bits long and are commonly written in the dotted-decimal notation — for example, 240.2.3.1, with dots separating the four bytes of the address written as decimal numbers. It is sometimes useful to view IP addresses as 32-bit unsigned numbers on the number line, $[0\ldots(2^{32}-1)]$, which we will refer to as the **IP number line**. For example, the IP address 240.2.3.1 represents the decimal number 4026663681 $(240 \times 2^{24} + 2 \times 2^{16} + 3 \times 2^{8} + 1)$ and the IP address 240.2.3.10 represents the decimal number 4026663690. Conceptually, each IPv4 address is a pair *(netid, hostid)*, where *netid* identifies a network, and *hostid* identifies a host on that network. All hosts on the same network have the same *netid* but different *hostid*s. Equivalently, the IP addresses of all hosts on the same network lie in a contiguous range on the IP number line.

The class-based Internet addressing architecture partitioned the IP address space into five classes — classes *A*, *B* and *C* for unicast traffic, class *D* for multicast traffic and class *E* reserved for future use. Classes were distinguished by the number of bits used to represent the *netid*. For example, a class *A* network consisted of a 7-bit *netid* and a 24-bit *hostid*, whereas a class *C* network consisted of a 21-bit *netid* and an 8-bit *hostid*. The first few

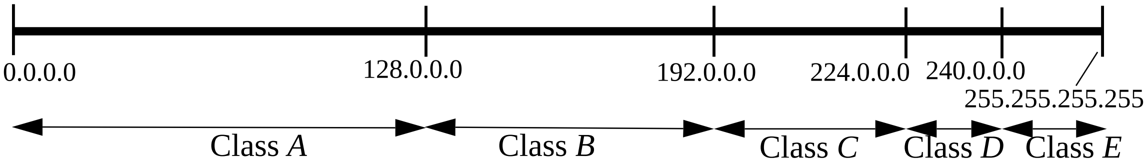


Figure 1.5 The IP number line and the original class-based addressing scheme. (The intervals represented by the classes are not drawn to scale.)

most-significant bits of an IP address determined its class, as shown in Table 1.1 and depicted on the IP number line in Figure 1.5.

TABLE 1.1. Class-based addressing.

Class	Range	Most significant address bits	netid	hostid
A	0.0.0.0 - 127.255.255.255	0	bits 1-7	bits 8-31
B	128.0.0.0 - 191.255.255.255	10	bits 2-15	bits 16-31
C	192.0.0.0 - 223.255.255.255	110	bits 3-23	bits 24-31
D (multicast)	224.0.0.0 - 239.255.255.255	1110	-	-
E (reserved for future use)	240.0.0.0 - 255.255.255.255	11110	-	-

The class-based addressing architecture enabled routers to use a relatively simple lookup operation. Typically, the forwarding table had three parts, one for each of the three unicast classes *A*, *B* and *C*. Entries in the forwarding table were tuples of the form <*netid*, *address of next hop*>. All entries in the same part had *netid*s of fixed-width — 7, 14 and 21 bits respectively for classes *A*, *B* and *C*, and the lookup operation for each incoming packet proceeded as in Figure 1.6. First, the class was determined from the most-significant bits of the packet's destination address. This in turn determined which of the three

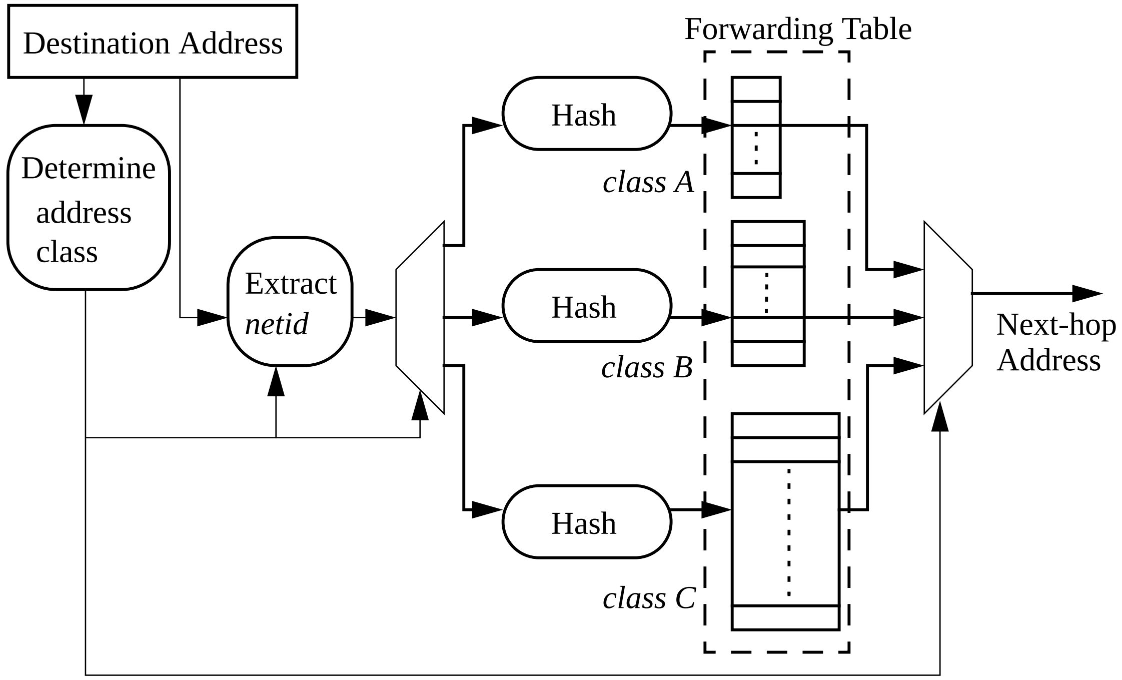


Figure 1.6 Typical implementation of the lookup operation in a class-based addressing scheme.

parts of the forwarding table to use. The router then searched for an exact match between the *netid* of the incoming packet and an entry in the selected part of the forwarding table. This exact match search could be performed using, for example, a hashing or a binary search algorithm [13].

The class-based addressing scheme worked well in the early days of the Internet. However, as the Internet grew, two problems emerged — a depletion of the IP address space, and an exponential growth of routing tables.

The allocation of network addresses on fixed *netid-hostid* boundaries (i.e., at the 8th, 16th and 24th bit positions, as shown in Table 1.1) was too inflexible, leading to a large number of wasted addresses. For example, a class *B netid* (good for 2^{16} *hostid*s) had to be allocated to any organization with more than 254 hosts.[1] In 1991, it was predicted

[1] While one class *C netid* accommodates 256 *hostid*s, the values 0 and 255 are reserved to denote network and broadcast addresses respectively.

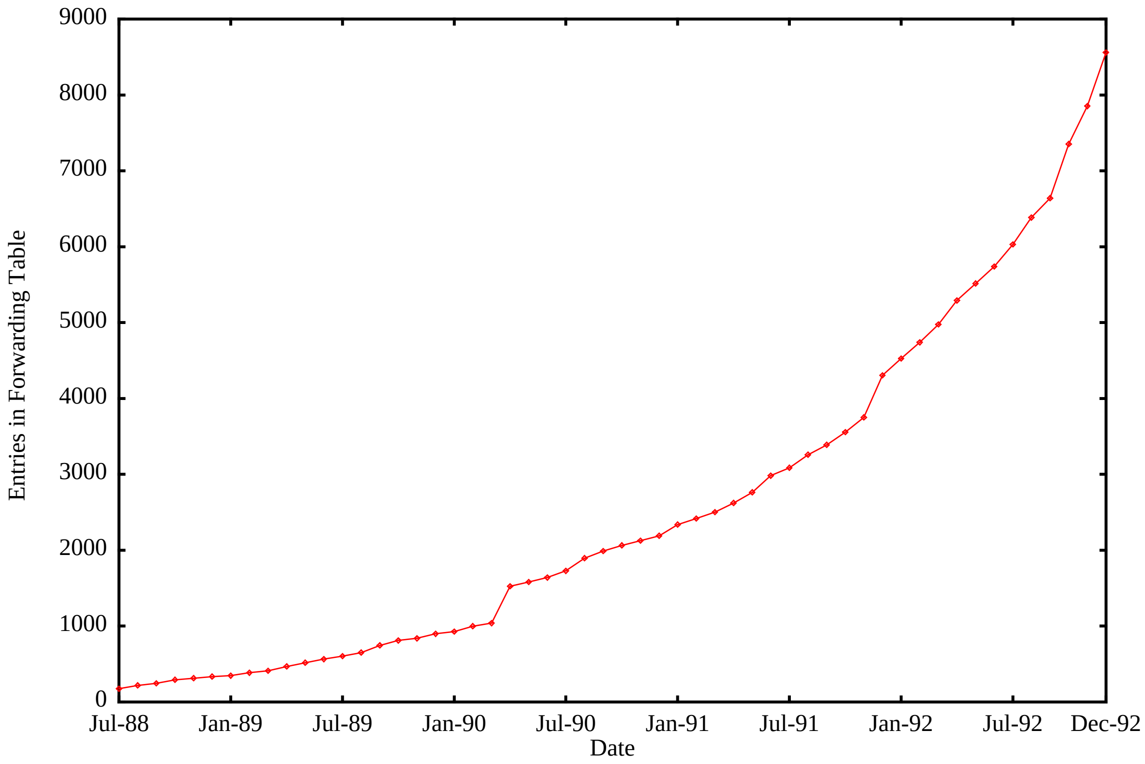


Figure 1.7 Forwarding tables in backbone routers were growing exponentially between 1988 and 1992 (i.e., under the class-based addressing scheme). (Source: RFC1519 [26])

[44][91][92] that the class B address space would be depleted in less than 14 months, and the whole IP address space would be exhausted by 1996 — even though less than 1% of the addresses allocated were actually in use [44].

The second problem was due to the fact that a backbone IP router stored every allocated *netid* in its routing table. As a result, routing tables were growing exponentially, as shown in Figure 1.7. This placed a high load on the processor and memory resources of routers in the backbone of the Internet.

In an attempt to slow down the growth of backbone routing tables and allow more efficient use of the IP address space, an alternative addressing and routing scheme called CIDR (Classless Inter-domain Routing) was adopted in 1993 [26][81]. CIDR does away with the class-based partitioning of the IP address space and allows *netid*s to be of arbitrary length rather than constraining them to be 7, 14 or 21 bits long. CIDR represents a *netid* using an IP *prefix* — a prefix of an IP address with a variable length of 0 to 32 significant bits and remaining wildcard bits.[1] An IP prefix is denoted by *P/l* where *P* is the prefix or *netid*, and *l* its length. For example, 192.0.1.0/24 is a 24-bit prefix that earlier belonged to class *C*. With CIDR, an organization with, say, 300 hosts can be allocated a prefix of length 23 (good for $2^{32-23} = 2^9 = 512$ *hostid*s) leading to more efficient address allocation.

This adoption of variable-length prefixes now enables a hierarchical allocation of IP addresses according to the physical topology of the Internet. A service provider that connects to the Internet backbone is allocated a short prefix. The provider then allocates longer prefixes out of its own address space to other smaller Internet Service Providers (ISPs) or sites that connect to it, and so on. Hierarchical allocation allows the provider to aggregate the routing information of the sites that connect to it, before advertising routes to the routers higher up in the hierarchy. This is illustrated in the following example:

Example 1.1: (see Figure 1.8) Consider an ISP P and two sites S and T connected to P. For instance, sites S and T may be two university campuses using P's network infrastructure for communication with the rest of the Internet. P may itself be connected to some backbone provider. Assume that P has been allocated a prefix 192.2.0.0/22, and it chooses to allocate the prefix 192.2.1.0/24 to S and 192.2.2.0/24 to T. This implies that routers in the backbone (such as R1 in Figure 1.8) only need to keep one table entry for the prefix 192.2.0.0/22 with P's network as the next-hop, i.e., they do not need to keep separate routing information for individual sites S and T. Similarly, Routers inside P's network (e.g., R5 and R6) keep entries to distinguish traffic among S and T, but not for any networks or sites that are connected downstream to S or T.

[1] In practice, the shortest prefix is 8 bits long.

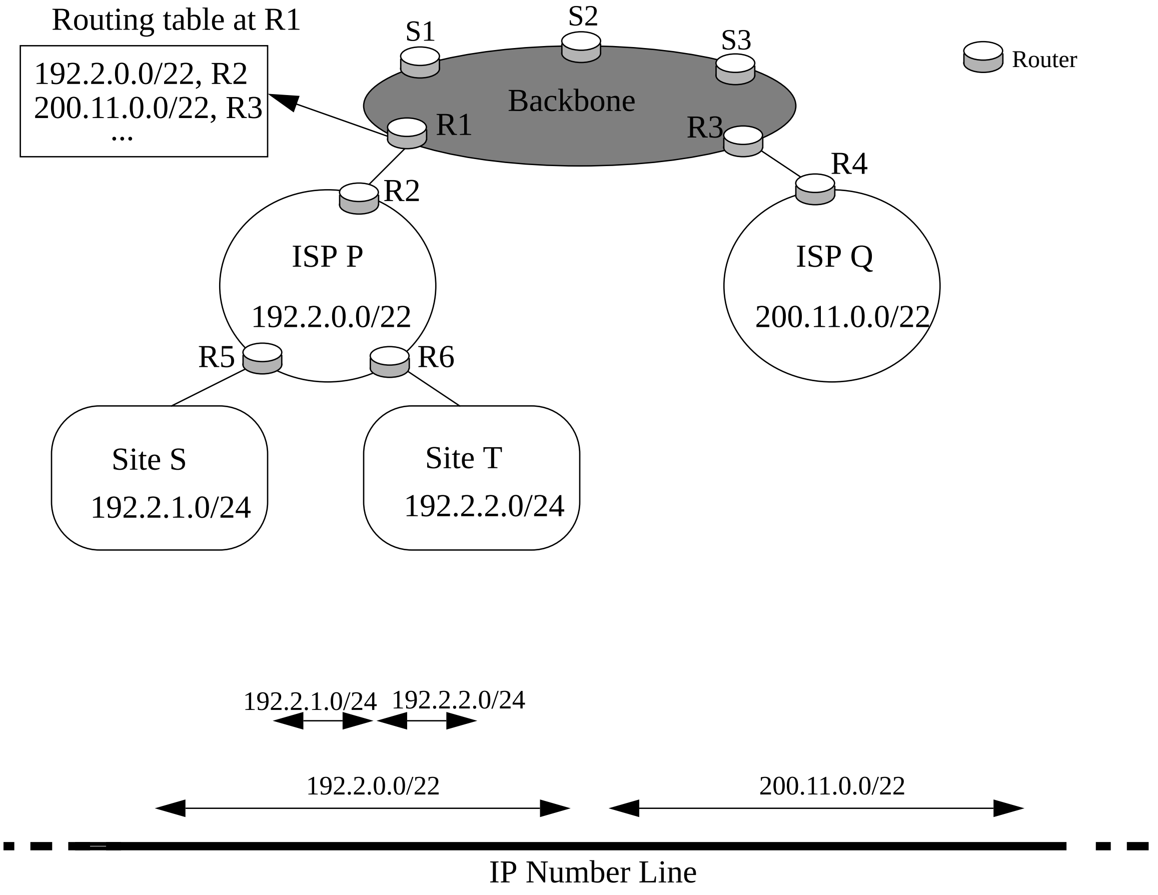


Figure 1.8 Showing how allocation of addresses consistent with the topology of the Internet helps keep the routing table size small. The prefixes are shown on the IP number line for clarity.

The aggregation of prefixes, or "route aggregation," leads to a reduction in the size of backbone routing tables. While Figure 1.7 showed an exponential growth in the size of routing tables before widespread adoption of CIDR in 1994, Figure 1.9 shows that the growth turned linear thereafter — at least till January 1998, since when it seems to have become faster than linear again.[1]

[1] It is a bit premature to assert that routing tables are again growing exponentially. In fact, the portion of the plot in Figure 1.9 after January 1998 fits well with an exponential as well as a quadratic curve. While not known definitively, the increased rate of growth could be because: (1) Falling costs of raw transmission bandwidth are encouraging decreased aggregation and a finer mesh of granularity; (2) Increasing expectations of reliability are forcing network operators to make their sites multi-homed.

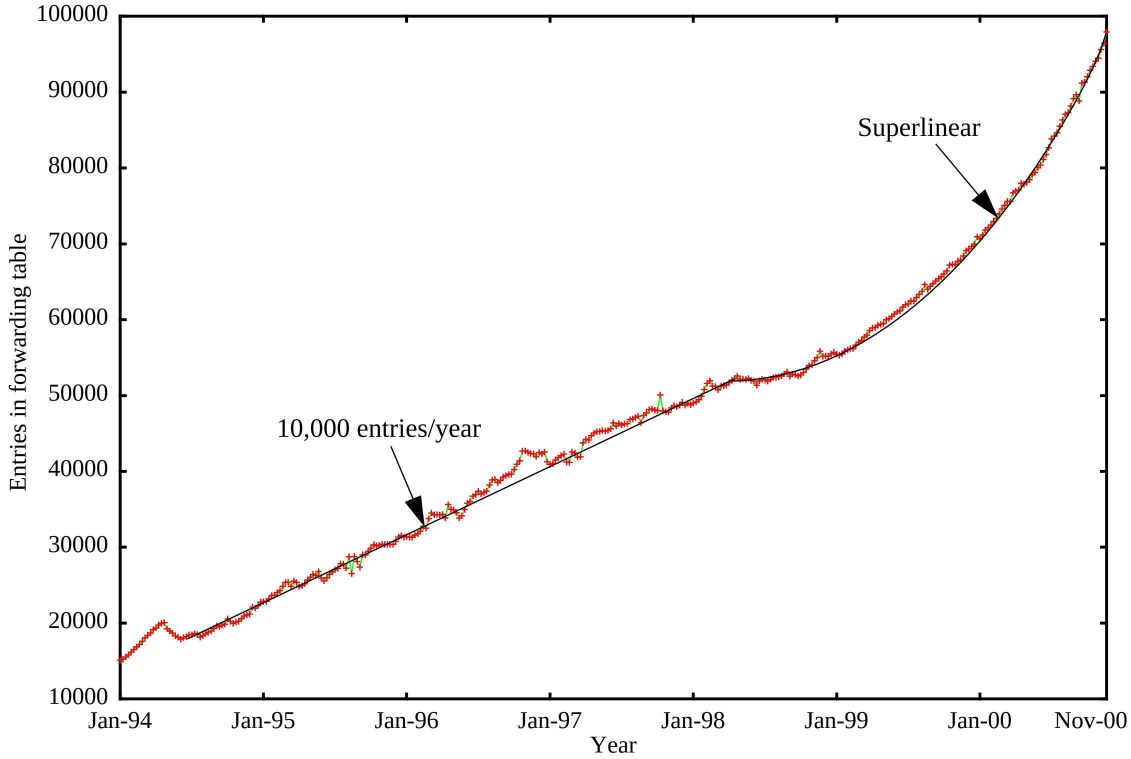


Figure 1.9 This graph shows the weekly average size of a backbone forwarding table (source [136]). The dip in early 1994 shows the immediate effect of widespread deployment of CIDR.

Hierarchical aggregation of addresses creates a new problem. When a site changes its service provider, it would prefer to keep its prefix (even though topologically, it is connected to the new provider). This creates a “hole” in the address space of the original provider — and so this provider must now create specific entries in its routing tables to allow correct forwarding of packets to the moved site. Because of the presence of specific entries, routers are required to be able to forward packets according to the *most specific route* present in their forwarding tables. The same capability is required when a site is multi-homed, i.e., has more than one connection to an upstream carrier or a backbone provider. The following examples make this clear:

Example 1.2: Assume that site T in Figure 1.8 with address space 192.2.2.0/24 changed its ISP to Q, as shown in Figure 1.10. The routing table at router R1 needs to have an additional entry corresponding to 192.2.2.0/24 pointing to Q’s network. Packets des-

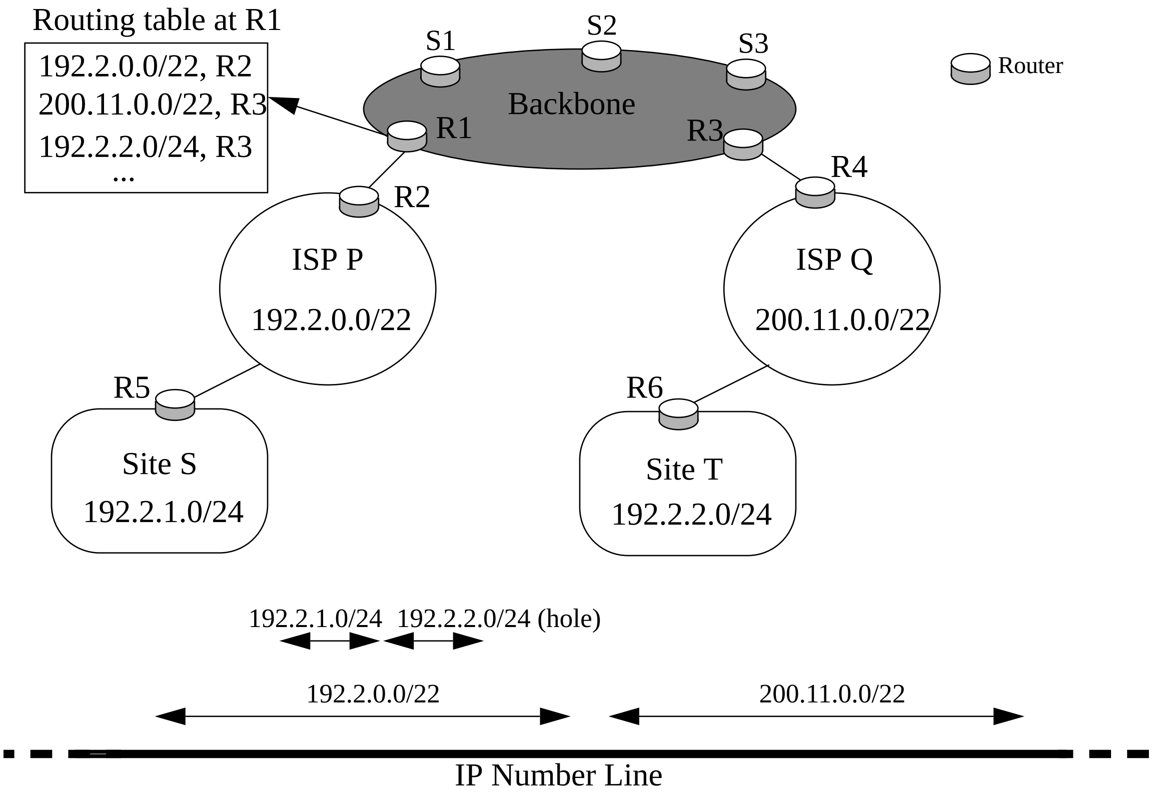


Figure 1.10 Showing the need for a routing lookup to find the most specific route in a CIDR environment.

tined to T at router R1 match this more specific route and are correctly forwarded to the intended destination in T (see Figure 1.10).

Example 1.3: Assume that ISP Q of Figure 1.8 is multi-homed, being connected to the backbone also through routers S4 and R7 (see Figure 1.11). The portion of Q's network identified with the prefix 200.11.1.0/24 is now better reached through router R7. Hence, the forwarding tables in backbone routers need to have a separate entry for this special case.

Lookups in the CIDR environment

With CIDR, a router's forwarding table consists of entries of the form <*route-prefix, next-hop-addr*>, where *route-prefix* is an IP prefix and *next-hop-addr* is the IP address of the next hop. A destination address *matches* a route-prefix if the significant bits of the pre-

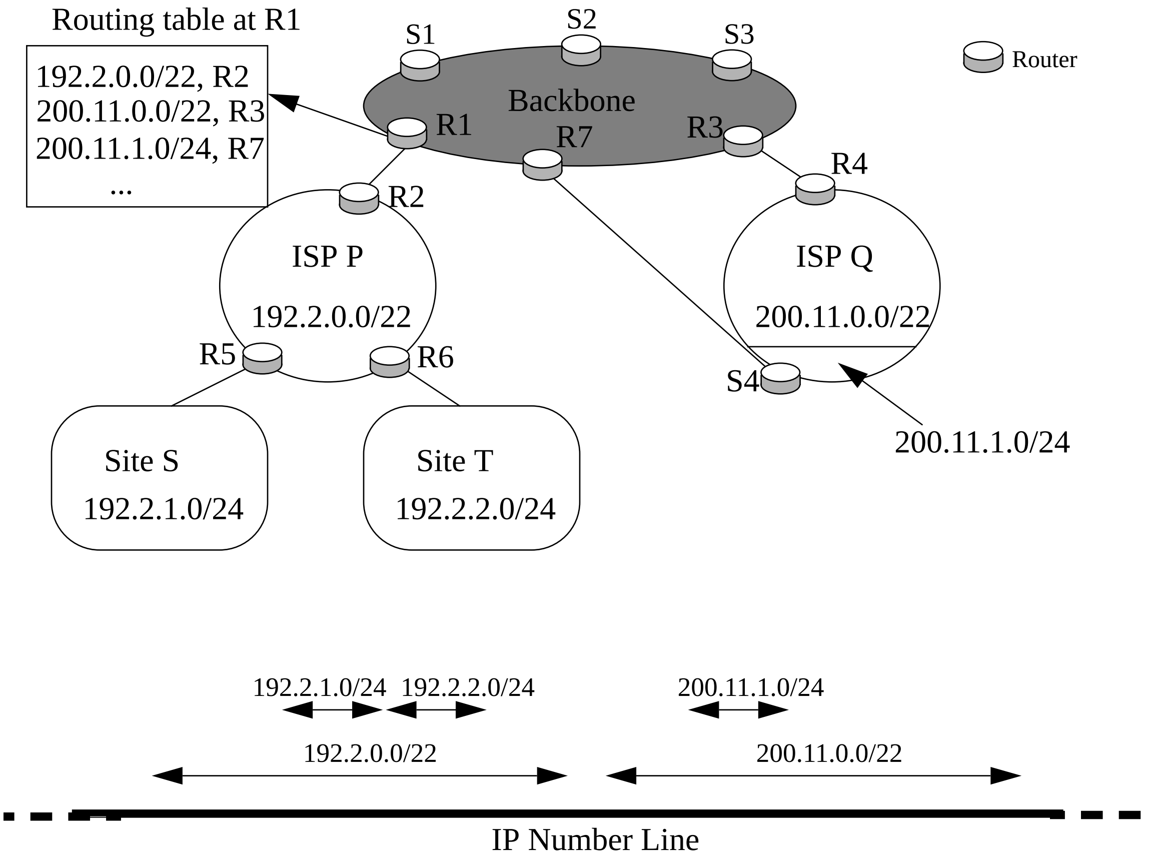


Figure 1.11 Showing how multi-homing creates special cases and hinders aggregation of prefixes.

fix are identical to the first few bits of the destination address. A routing lookup operation on an incoming packet requires the router to find the most specific route for the packet. This implies that the router needs to solve the **longest prefix matching problem**, defined as follows.

Definition 1.1: *The **longest prefix matching** problem is the problem of finding the forwarding table entry containing the longest prefix among all prefixes (in other forwarding table entries) matching the incoming packet's destination address. This longest prefix is called the longest matching prefix.*

Example 1.4: The forwarding table in router R1 of Figure 1.10 is shown in Table 1.2. If an incoming packet at this router has a destination address of 200.11.0.1, it will match only the prefix 200.11.0.0/22 (entry #2) and hence will be forwarded to router R3.

If the packet's destination address is 192.2.2.4, it matches two prefixes (entries #1 and #3). Because entry #3 has the longest matching prefix, the packet will be forwarded to router R3.

TABLE 1.2. The forwarding table of router R1 in Figure 1.10.

Entry Number	Prefix	Next-Hop
1.	192.2.0.0/22	R2
2.	200.11.0.0/22	R3
3.	192.2.2.0/24	R3

Difficulty of longest prefix matching

The destination address of an arriving packet does not carry with it the information needed to determine the length of the longest matching prefix. Hence, we cannot find the longest match using an exact match search algorithm (for example, using hashing or a binary search procedure). Instead, a search for the longest matching prefix needs to determine both the length of the longest matching prefix as well as the forwarding table entry containing the prefix of this length that matches the incoming packet's destination address. One naive longest prefix matching algorithm is to perform 32 different exact match search operations, one each for all prefixes of length i, $1 \le i \le 32$. This algorithm would require 32 exact match search operations. As we will see later in this thesis, faster algorithms are possible.

In summary, the need to perform longest prefix matching has made routing lookups more complicated now than they were before the adoption of CIDR when only one exact match search operation was required. Chapters 2 and 3 of this thesis will present efficient longest prefix matching algorithms for fast routing lookups.

2 Flow-aware IP router and packet classification

As mentioned earlier, routers may optionally classify packets into flows for special processing. In this section, we first describe why some routers are flow-aware, and how they use packet classification to recognize flows. We also provide a brief overview of the architecture of flow-aware routers. We then provide the background leading to the formal definition of the packet classification problem. Fast packet classification is the subject of the second part of this thesis (Chapters 4 and 5).

2.1 Motivation

One main reason for the existence of flow-aware routers stems from an ISP's desire to have the capability of providing differentiated services to its users. Traditionally, the Internet provides only a "best-effort" service, treating all packets going to the same destination identically, and servicing them in a first-come-first-served manner. However, the rapid growth of the Internet has caused increasing congestion and packet loss at intermediate routers. As a result, some users are willing to pay a premium price in return for better service from the network. To maximize their revenue, the ISPs also wish to provide different levels of service at different prices to users based on their requirements, while still deploying one common network infrastructure.[1]

In order to provide differentiated services, routers require additional mechanisms. These mechanisms — admission control, conditioning (metering, marking, shaping, and policing), resource reservation (optional), queue management and fair scheduling (such as weighted fair queueing) — require, first of all, the capability to distinguish and isolate traffic belonging to different users based on service agreements negotiated between the ISP and its customer. This has led to demand for flow-aware routers that negotiate these

[1] This is analogous to the airlines, who also provide differentiated services (such as economy and business class) to different users based on their requirements, while still using the same common infrastructure.

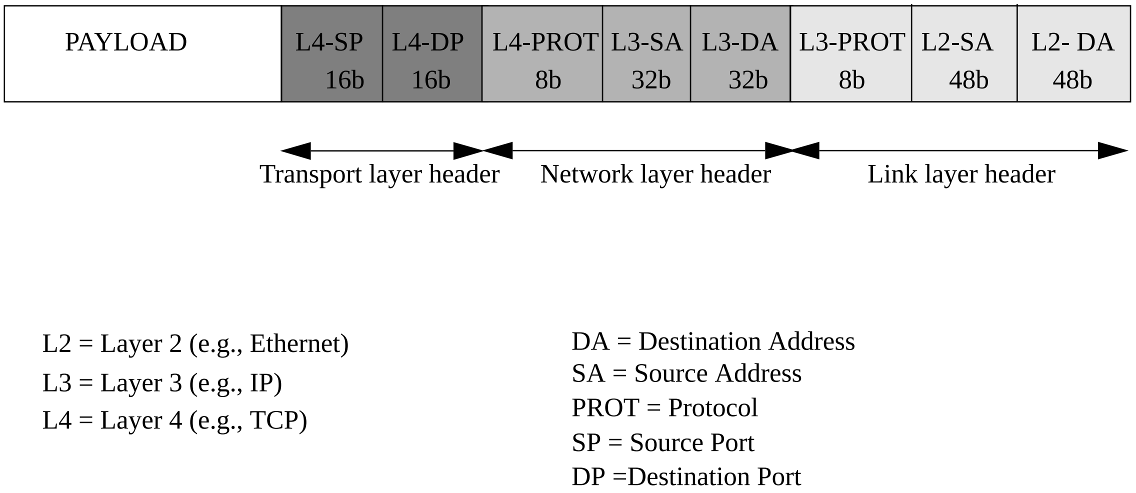


Figure 1.12 This figure shows some of the header fields (and their widths) that might be used for classifying a packet. Although not shown in this figure, higher layer (e.g., application-layer) fields may also be used for packet classification.

service agreements, express them in terms of *rules* or *policies* configured on incoming packets, and isolate incoming traffic according to these rules.

We call a collection of rules or policies a *policy database*, *flow classifier*, or simply a *classifier*.[1] Each rule specifies a flow that a packet may belong to based on some criteria on the contents of the packet header, as shown in Figure 1.12. All packets belonging to the same flow are treated in a similar manner. The identified flow of an incoming packet specifies an *action* to be applied to the packet. For example, a firewall router may carry out the action of either *denying* or *allowing* access to a protected network. The determination of this action is called *packet classification* — the capability of routers to identify the action associated with the "best" rule an incoming packet matches. Packet classification allows ISPs to differentiate from their competition and gain additional revenue by providing different value-added services to different customers.

[1] Sometimes, the functional datapath element that classifies packets is referred to as a classifier. In this thesis, however, we will consistently refer to the policy database as a classifier.

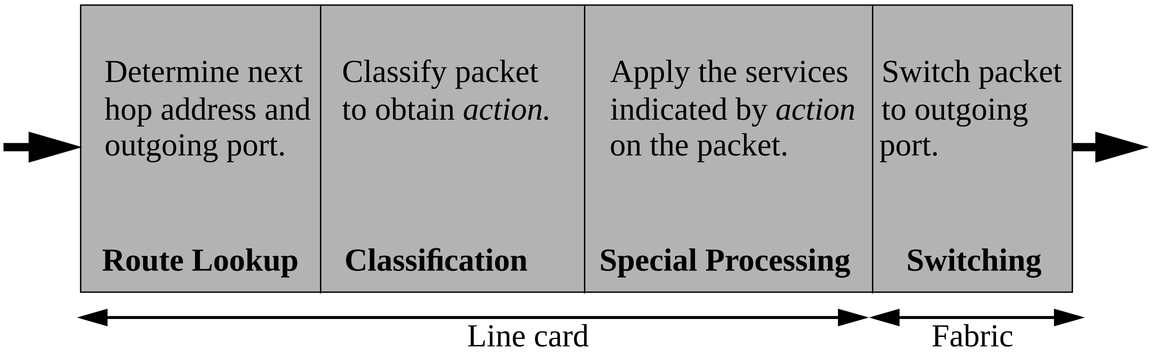

Determine next hop address and outgoing port.
Classify packet to obtain *action.*
Apply the services indicated by *action* on the packet.
Switch packet to outgoing port.
Route Lookup
Classification
Special Processing
Switching
Line card
Fabric

Figure 1.13 Datapath of a packet through a flow-aware router. Note that in some applications, a packet may need to be classified both before and after route lookup.

2.2 Architecture of a flow-aware router

Flow-aware routers perform a superset of the functions of a packet-by-packet router. The typical path taken by a packet through a flow-aware router is shown in Figure 1.13 and consists of four main functions on the packet: (1) performing route lookup to identify the outgoing port, (2) performing classification to identify the flow to which an incoming packet belongs, (3) applying the action (as part of the provisioning of differentiated services or some other form of special processing) based on the result of classification, and (4) switching to the output port. The various forms of special processing in function (3), while interesting in their own right, are not the subject of this thesis. The following references describe a variety of actions that a router may perform: admission control [42], queueing [25], resource reservation [6], output link scheduling [18][74][75][89] and billing [21].

2.3 Background and definition of the packet classification problem

Packet classification enables a number of additional, non-best-effort network services other than the provisioning of differentiated qualities of service. One of the well-known applications of packet classification is a firewall. Other network services that require packet classification include policy-based routing, traffic rate-limiting and policing, traffic

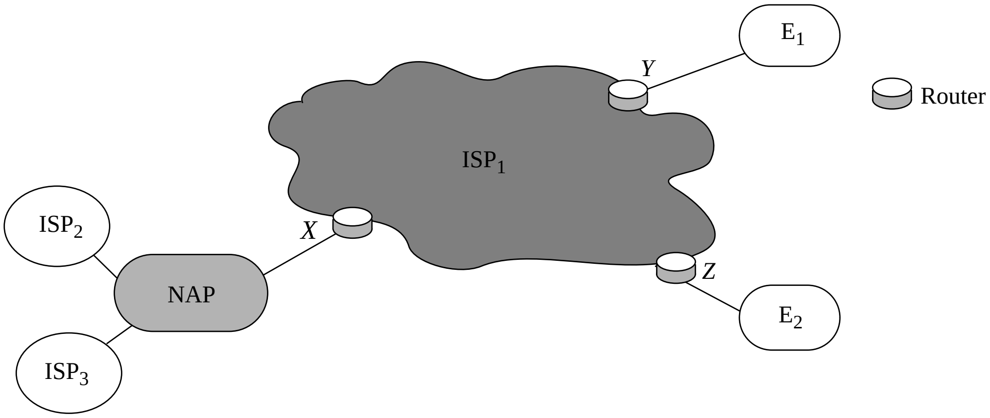


Figure 1.14 Example network of an ISP (ISP$_1$) connected to two enterprise networks (E$_1$ and E$_2$) and to two other ISP networks across a network access point (NAP).

shaping, and billing. In each case, it is necessary to determine which flow an arriving packet belongs to so as to determine — for example — whether to forward or filter it, where to forward it to, what type of service it should receive, or how much should be charged for transporting it.

TABLE 1.3. Some examples of value-added services.

Service	**Example**
Packet Filtering	Deny all traffic from ISP$_3$ (on interface *X*) destined to E$_2$.
Policy Routing	Send all voice-over-IP traffic arriving from E$_1$ (on interface *Y*) and destined to E$_2$ via a separate ATM network.
Accounting & Billing	Treat all video traffic to E$_1$ (via interface *Y*) as highest priority and perform accounting for such traffic.
Traffic Rate-limiting	Ensure that ISP$_2$ does not inject more than 10 Mbps of email traffic and 50 Mbps of total traffic on interface *X*.
Traffic Shaping	Ensure that no more than 50 Mbps of web traffic is sent to ISP$_2$ on interface *X*.

To help illustrate the variety of packet classifiers, let us consider some examples of how packet classification can be used by an ISP to provide different services. Figure 1.14 shows ISP$_1$ connected to three different sites: two enterprise networks E$_1$ and E$_2$, and a

Network Access Point[1] (NAP), which is in turn connected to two other ISPs — ISP_2 and ISP_3. ISP_1 provides a number of different services to its customers, as shown in Table 1.3.

Table 1.4 shows the categories that an incoming packet must be classified into by the router at interface *X*. Note that the classes specified may or may not be mutually exclusive.

TABLE 1.4. Given the rules in Table 1.3, the router at interface X must classify an incoming packet into the following categories.

Service	Flow	Relevant Packet Fields
Packet Filtering	From ISP_3 and going to E_2	Source link-layer address, destination network-layer address
Traffic rate-limiting	Email and from ISP_2	Source link-layer address, source transport port number
Traffic shaping	Web and to ISP_2	Destination link-layer address, destination transport port number
	All other packets	—

For example, the first and second flow in Table 1.4 overlap. This happens commonly, and when no explicit priorities are specified, we follow the convention that rules closer to the top of the list have higher priority.

With this background, we proceed to define the problem of packet classification.

Each rule of the classifier has d components. The i^{th} component of rule *R*, denoted as $R[i]$, is a regular expression on the i^{th} field of the packet header. A packet *P* is said to *match* a particular rule *R*, if $\forall i$, the i^{th} field of the header of *P* satisfies the regular expression $R[i]$. In practice, a rule component is not a general regular expression — often limited by syntax to a simple address/mask or operator/number(s) specification. In an address/mask specification, a 0 at bit position *x* in the mask denotes that the corresponding bit in

[1] A network access point (NAP) is a network location which acts as an exchange point for Internet traffic. An ISP connects to a NAP to exchange traffic with other ISPs at that NAP.

the address is a “don’t care” bit. Similarly, a 1 at bit position *x* in the mask denotes that the corresponding bit in the address is a significant bit. For instance, the first and third most significant bytes in a packet field matching the specification *171.4.3.4/255.0.255.0* must be equal to 171 and 3, respectively, while the second and fourth bytes can have any value. Examples of operator/number(s) specifications are *eq 1232* and *range 34-9339*, which specify that the matching field value of an incoming packet must be equal to 1232 in the former specification, and can have any value between 34 and 9339 (both inclusive) in the latter specification. Note that a route-prefix can be specified as an address/mask pair where the mask is *contiguous* — i.e., all bits with value 1 appear to the left of (i.e., are more significant than) bits with value 0 in the mask. For instance, the mask for an 8-bit prefix is 255.0.0.0. A route-prefix of length l can also be specified as a range of width equal to 2^t where $t = 32 - l$. In fact, most of the commonly occurring specifications in practice can be viewed as range specifications.

We can now formally define packet classification:

Definition 1.2: *A classifier* C *has* N *rules,* R_j, $1 \le j \le N$, *where* R_j *consists of three entities — (1) A regular expression* $R_j[i]$, $1 \le i \le d$, *on each of the* d *header fields, (2) A number* $pri(R_j)$, *indicating the priority of the rule in the classifier, and (3) An action, referred to as* $action(R_j)$. *For an incoming packet* P *with the header considered as a d-tuple of points* $(P_1, P_2, \ldots, P_d)$, *the* ***d-dimensional packet classification*** *problem is to find the rule* R_m *with the highest priority among all rules* R_j *matching the d-tuple; i.e.,* $pri(R_m) > pri(R_j)$, $\forall j \ne m$, $1 \le j \le N$, *such that* P_i *matches* $R_j[i]$, $\forall (1 \le i \le d)$. *We call rule* R_m *the best matching rule for packet* P.

Example 1.5: An example of a classifier in four dimensions is shown in Table 1.5. By convention, the first rule R1 has the highest priority and rule R7 has the lowest priority (‘*’ denotes a complete wildcard specification, and ‘gt v’ denotes any value greater than v). Classification results on some example packets using this classifier are shown in Table 1.6.

TABLE 1.5. An example classifier.

Rule	Network-layer destination (address/mask)	Network-layer source (address/ mask)	Transport-layer destination	Transport-layer protocol	Action
R1	152.163.190.69/ 255.255.255.255	152.163.80.11/ 255.255.255.255	*	*	Deny
R2	152.168.3.0/ 255.255.255.0	152.163.200.157/ 255.255.255.255	eq http	udp	Deny
R3	152.168.3.0/ 255.255.255.0	152.163.200.157/ 255.255.255.255	range 20-21	udp	Permit
R4	152.168.3.0/ 255.255.255.0	152.163.200.157/ 255.255.255.255	eq http	tcp	Deny
R5	152.163.198.4/ 255.255.255.255	152.163.161.0/ 255.255.252.0	gt 1023	tcp	Permit
R6	152.163.198.4/ 255.255.255.255	152.163.0.0/ 255.255.0.0	gt 1023	tcp	Deny
R7	*	*	*	*	Permit

TABLE 1.6. Examples of packet classification on some incoming packets using the classifier of Table 1.5.

Packet Header	Network-layer destination address	Network-layer source address	Transport-layer destination port	Transport-layer protocol	Best matching rule, action
P1	152.163.190.69	152.163.80.11	http	tcp	R1, Deny
P2	152.168.3.21	152.163.200.157	http	udp	R2, Deny
P3	152.168.198.4	152.163.160.10	1024	tcp	R5, Permit

We can see that routing lookup is an instance of one-dimensional packet classification. In this case, all packets destined to the set of addresses described by a common prefix may be considered to be part of the same flow. Each rule has a route-prefix as its only component and has the next hop address associated with this prefix as the action. If we define the priority of the rule to be the length of the route-prefix, determining the longest-matching prefix for an incoming packet is equivalent to determining the best matching rule in the classifier. The packet classification problem is therefore a generalization of the routing lookup problem. Chapters 4 and 5 of this thesis will present efficient algorithms for fast packet classification in flow-aware routers.

3 Goals and metrics for lookup and classification algorithms

A lookup or classification algorithm preprocesses a routing table or a classifier to compute a data structure that is then used to lookup or classify incoming packets. This preprocessing is typically done in software in the routing processor, discussed in Section 1.1. There are a number of properties that we desire for all lookup and classification algorithms:

- High speed.
- Low storage requirements.
- Flexibility in implementation.
- Ability to handle large real-life routing tables and classifiers.
- Low preprocessing time.
- Low update time.
- Scalability in the number of header fields (for classification algorithms only).
- Flexibility in specification (for classification algorithms only).

We now discuss each of these properties in detail.

- *High speed* — Increasing data rates of physical links require faster address lookups at routers. For example, links running at OC192c (approximately 10 Gbps) rates need the router to process 31.25 million packets per second (assuming minimum-sized 40 byte TCP/IP packets).[1] We generally require algorithms to perform well in the worst case, e.g., classify packets at wire-speed. If this were not the case, all packets (regardless of the flow they belong to) would need to be queued before the classification function.This would defeat the purpose of distinguishing and isolating flows, and applying different actions on them. For example, it would make it much harder to control the delay of a flow through the router. At the same time, in some applications, for example, those that do not provide qualities of service, a lookup or classification algorithm that performs well in the *average* case may be acceptable, in fact desirable, because the average lookup performance can be much higher than the worst-case performance. For such applications, the algorithm needs to process packets at the rate of 3.53 million packets per second for OC192c links, assuming an average Internet packet size of approximately 354 bytes [121]. Table 1.7 lists the lookup performance required in one router port to

TABLE 1.7. Lookup performance required as a function of line-rate and packet size.

Year	Line	Line-rate (Gbps)	40-byte packets (Mpps)	84-byte packets (Mpps)	354-byte packets (Mpps)
1995-7	T1	0.0015	0.0468	0.0022	0.00053
1996-8	OC3c	0.155	0.48	0.23	0.054
1997-8	OC12c	0.622	1.94	0.92	0.22
1999-2000	OC48c	2.50	7.81	3.72	0.88
(Now) 2000-1	OC192c	10.0	31.2	14.9	3.53
(Next) 2002-3	OC768c	40.0	125.0	59.5	14.1
1997-2000	1 Gigabit-Ethernet	1.0	N.A.	1.49	0.35

[1] In practice, IP packets are encapsulated and framed before being sent on SONET links. The most commonly used encapsulation method is PPP (Point-to-Point Protocol) in HDLC-like framing. (HDLC stands for High-level Data Link Control). This adds either 7 or 9 bytes of overhead (1 byte flag, 1 byte address, 1 byte control, 2 bytes protocol and 2 to 4 bytes of frame check sequence fields) to the packet. When combined with the SONET overhead (27 bytes of line and section overhead in a 810 byte frame), the lookup rate required for 40 byte TCP/IP packets becomes approximately 25.6 Mpps. (Please see IETF RFC 1661/1662 for PPP/HDLC framing and RFC 1619/2615 for PPP over SONET.)

handle a continuous stream of incoming packets of a given size (84 bytes is the minimum size of a Gigabit-Ethernet frame — this includes a 64-byte packet, 7-byte preamble, 1-byte start-of-frame delimiter, and 12 bytes of inter-frame gap).

- *Flexibility in implementation* — The forwarding engine may be implemented either in software or in hardware depending upon the system requirements. Thus, a lookup or classification algorithm should be suitable for implementation in both hardware and software. For the highest speeds (e.g., for OC192c in the year 2000), we expect that hardware implementation will be necessary — hence, the algorithm design should be amenable to pipelined implementation.
- *Low storage requirements* — We desire that the storage requirements of the data structure computed by the algorithm be small. Small storage requirements enable the use of fast but expensive memory technologies like SRAMs (Synchronous Random Access Memories). A memory-efficient algorithm can benefit from an on-chip cache if implemented in software, and from an on-chip SRAM if implemented in hardware.
- *Ability to handle large real-life routing tables and classifiers* — The algorithm should scale well both in terms of storage and speed with the size of the forwarding table or the classifier. At the time of the writing of this thesis, the forwarding tables of backbone routers contain approximately 98,000 route-prefixes and are growing rapidly (as shown in Figure 1.9). A lookup engine deployed in the year 2001 should be able to support approximately 400,000-512,000 prefixes in order to be useful for at least five years. Therefore, lookup and classification algorithms should demonstrate good performance on current real-life routing tables and classifiers, as well as accommodate future growth.
- *Low preprocessing time* — Preprocessing time is the time taken by an algorithm to compute the initial data structure. An algorithm that supports incremental updates of its data structure is said to be *“dynamic.”* A *“static”* algorithm requires the whole data structure to be recomputed each time a rule is added or

deleted. In general, dynamic algorithms can tolerate larger preprocessing times than static algorithms. (The absolute values differ with applications.)

- *Low update time* — Routing tables have been found to change fairly frequently, often at the peak rate of a few hundred prefixes per second and at the average rate of more than a few prefixes per second [47]. A lookup algorithm should be able to update the data structure at least this fast. For classification algorithms, the update rate differs widely among different applications — a very low update rate may be sufficient in firewalls where entries are added manually or infrequently. On the other hand, a classification algorithm must be able to support a high update rate in so called “stateful” classifiers where a packet may dynamically trigger the addition or deletion of a new rule or a fine-granularity flow.
- *Scalability in the number of header fields* (for classification algorithms only) — A classification algorithm should ideally allow matching on arbitrary fields, including link-layer, network-layer, transport-layer and — in some cases — the application-layer headers.[1] For instance, URL (universal resource locator — the identifier used to locate resources on the World Wide Web) based classification may be used to route a user’s packets across a different network or to give the user a different quality of service. Hence, while it makes sense to optimize for the commonly used header fields, the classification algorithm should not preclude the use of other header fields.
- *Flexibility in specification* (for classification algorithms only) — A classification algorithm should support flexible rule specifications, including prefixes, operators (such as range, less than, greater than, equal to, etc.) and wildcards. Even non-contiguous masks may be required, depending on the application using classification.

[1] That is why packet-classifying routers have sometimes been called “layerless switches”.

4 Outline of the thesis

This thesis proposes several novel lookup and classification algorithms. There is one chapter devoted to each algorithm. Each chapter first presents background work related to the algorithm. It then presents the motivation, key concepts, properties, and implementation results for the algorithm. It also evaluates the algorithm against the metrics outlined above and against previous work on the subject.

Chapter 2 presents an overview of previous work on routing lookups. It proposes and discusses a simple routing lookup algorithm optimized for implementation in dedicated hardware. This algorithm performs the longest prefix matching operation in two memory accesses that can be pipelined to give the throughput of one routing lookup every memory access. This corresponds to 20 million packets per second with 50 ns DRAMs (Dynamic Random Access Memories).

With the motivation of high speed routing lookups, Chapter 3 defines a new problem of minimizing the average lookup time while keeping the maximum lookup time bounded. This chapter then describes and analyzes two algorithms to solve this new problem. Experiments show an improvement by a factor of 1.7 in the average number of memory accesses per lookup over those obtained by worst-case lookup time minimization algorithms. Moreover, the algorithms proposed in this chapter support almost perfect balancing of the incoming lookup load, making them easily parallelizable for high speed designs.

In Chapter 4, we move on to the problem of multi-field packet classification. Chapter 4 provides an overview of previous work and highlights the issues in designing solutions for this problem. This chapter proposes and discusses the performance of a novel algorithm for fast classification on multiple header fields.

Chapter 5 presents another new algorithm for high speed multi-field packet classification. This algorithm is different from the one proposed in Chapter 4 in that it supports fast incremental updates, is otherwise slower, and occupies a smaller amount of storage.

Finally, Chapter 6 concludes by discussing directions for future work in the area of fast routing lookups and packet classification.

CHAPTER 2

An Algorithm for Performing Routing Lookups in Hardware

1 Introduction

This chapter describes a longest prefix matching algorithm to perform fast IPv4 route lookups in hardware. The chapter first presents an overview of previous work on IP lookups in Section 2. As we will see, most longest prefix matching algorithms proposed in the literature are designed primarily for implementation in software. They attempt to optimize the storage requirements of their data structure, so that the data structure can fit in the fast cache memories of high speed general purpose processors. As a result, these algorithms do not lend themselves readily to hardware implementation.

Motivated by the observation in Section 3 of Chapter 1 that the performance of a lookup algorithm is most often limited by the number of memory accesses, this chapter presents an algorithm to perform the longest matching prefix operation for IPv4 route lookups in hardware in two memory accesses. The accesses can be pipelined to achieve one route lookup every memory access. With 50 ns DRAM, this corresponds to approximately 20×10^6 packets per second — enough to forward a continuous stream of 64-byte packets arriving on an OC192c line.

The lookup algorithm proposed in this chapter achieves high throughput by using precomputation and trading off storage space with lookup time. This has the side-effect of increased update time and overhead to the central processor, and motivates the low-overhead update algorithms presented in Section 5 of this chapter.

1.1 Organization of the chapter

Section 2 provides an overview of previous work on route lookups and a comparative evaluation of the different routing lookup schemes proposed in literature. Section 3 describes the proposed route lookup algorithm and its data structure. Section 4 discusses some variations of the basic algorithm that make more efficient use of memory. Section 5 investigates how route entries can be quickly inserted and removed from the data structure. Finally, Section 6 concludes with a summary of the contributions of this chapter.

2 Background and previous work on route lookup algorithms

This section begins by briefly describing the basic data structures and algorithms for longest prefix matching, followed by a description of some of the more recently proposed schemes and a comparative evaluation (both qualitative and quantitative) of their performance. In each case, we provide only an overview, referring the reader to the original references for more details.

2.1 Background: basic data structures and algorithms

We will use the forwarding table shown in Table 2.1 as an example throughout this subsection. This forwarding table has four prefixes of maximum width 5 bits, assumed to have been added to the table in the sequence P1, P2, P3, P4.

TABLE 2.1. An example forwarding table with four prefixes. The prefixes are written in binary with a '*' denoting one or more trailing wildcard bits — for instance, 10* is a 2-bit prefix.

	Prefix	**Next-hop**
P1	111*	H1
P2	10*	H2
P3	1010*	H3
P4	10101	H4

2.1.1 Linear search

The simplest data structure is a linked-list of all prefixes in the forwarding table. The lookup algorithm traverses the list of prefixes one at a time, and reports the longest matching prefix at the end of the traversal. Insertion and deletion algorithms perform trivial linked-list operations. The storage complexity of this data structure for N prefixes is $O(N)$. The lookup algorithm has time complexity $O(N)$ and is thus too slow for practical purposes when N is large. The insertion and deletion algorithms have time complexity $O(1)$, assuming the location of the prefix to be deleted is known.

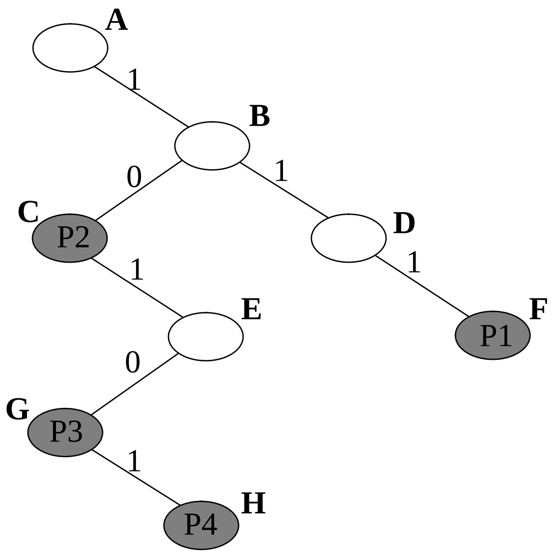
The average lookup time of a linear search algorithm can be made smaller if the prefixes are sorted in order of decreasing length. For example, with this modification, the prefixes of Table 2.1 would be kept in the order P4, P3, P1, P2; and the lookup algorithm would be modified to simply stop traversal of the linked-list the first time it finds a matching prefix.

2.1.2 Caching of recently seen destination addresses

The idea of caching, first used for improving processor performance by keeping frequently accessed data close to the CPU [34], can be applied to routing lookups by keeping recently seen destination addresses and their lookup results in a *route-cache*. A full lookup (using some longest prefix matching algorithm) is now performed only if the incoming destination address is not already found in the cache.

Cache hit rate needs to be high in order to achieve a significant performance improvement. For example, if we assume that a full lookup is 20 times slower than a cache lookup, the hit rate needs to be approximately 95% or higher for a performance improvement by a factor of 10. Early studies [22][24][77] reported high cache hit rates with large parallel caches: for instance, Partridge [77] reports a hit rate of 97% with a cache of size 10,000 entries, and 77% with a cache of size 2000 entries. Reference [77] suggests that the cache size should scale linearly with the increase in the number of hosts or the amount of Internet traffic. This implies the need for exponentially growing cache sizes. Cache hit rates are expected to decrease with the growth of Internet traffic because of decreasing temporal locality [66]. The temporal locality of traffic is decreasing because of an increasing number of concurrent flows at high-speed aggregation points and decreasing duration of a flow, probably because of an increasing number of short web transfers on the Internet.

A cache management scheme must decide which cache entry to replace upon addition of a new entry. For a route cache, there is an additional overhead of flushing the cache on route updates. Hence, low hit rates, together with cache search and management overhead, may even degrade the overall lookup performance. Furthermore, the variability in lookup times of different packets in a caching scheme is undesirable for the purpose of hardware implementation. Because of these reasons, caching has generally fallen out of favor with router vendors in the industry (see Cisco [120], Juniper [126] and Lucent [128]) who tout fast hardware lookup engines that do not use caching.



Trie node

next-hop-ptr (if prefix present)	
left-ptr	right-ptr

Figure 2.1 A binary trie storing the prefixes of Table 2.1. The gray nodes store pointers to next-hops. Note that the actual prefix values are never stored since they are implicit from their position in the trie and can be recovered by the search algorithm. Nodes have been named A, B, ..., H in this figure for ease of reference.

2.1.3 Radix trie

A radix trie, or simply a trie,[1] is a binary tree that has labeled branches, and that is traversed during a search operation using individual bits of the search key. The left branch of a node is labeled '0' and the right-branch is labeled '1.' A node, v, represents a bit-string formed by concatenating the labels of all branches in the path from the root node to v. A prefix, p, is stored in the node that represents the bit-string p. For example, the prefix $0*$ is stored in the left child of the root node.

[1] The name trie comes from re**trie**val, but is pronounced "try". See Section 6.3 on page 492 of Knuth [46] for more details on tries.

A trie for W-bit prefixes has a maximum depth of W nodes. The trie for the example forwarding table of Table 2.1 is shown in Figure 2.1.

The longest prefix search operation on a given destination address proceeds bitwise starting from the root node of the trie. The left (right) branch of the root node is taken if the first bit of the address is ‘0’ (‘1’). The remaining bits of the address determine the path of traversal in a similar manner. The search algorithm keeps track of the prefix encountered most recently on the path. When the search ends at a null pointer, this most recently encountered prefix is the longest prefix matching the key. Therefore, finding the longest matching prefix using a trie takes W memory accesses in the worst case, i.e., has time complexity $O(W)$.

The insertion operation proceeds by using the same bit-by-bit traversal algorithm as above. Branches and internal nodes that do not already exist in the trie are created as the trie is traversed from the root node to the node representing the new prefix. Hence, insertion of a new prefix can lead to the addition of at most W other trie nodes. The storage complexity of a W-bit trie with N prefixes is thus $O(NW)$.[1]

An IPv4 route lookup operation is slow on a trie because it requires up to 32 memory accesses in the worst case. Furthermore, a significant amount of storage space is wasted in a trie in the form of pointers that are null, and that are on *chains* — paths with 1-degree nodes, i.e., that have only one child (e.g., path BCEGH in Figure 2.1).

Example 2.1: Given an incoming 5-bit address 10111 to be looked up in the trie of Figure 2.1, the longest prefix matching algorithm takes the path ABCE before reaching a null pointer. The last prefix encountered on this path, prefix P2 (10*) in node C, is the desired longest matching prefix.

[1] The total amount of space is, in fact, slightly less than NW because prefixes share trie branches near the root node.

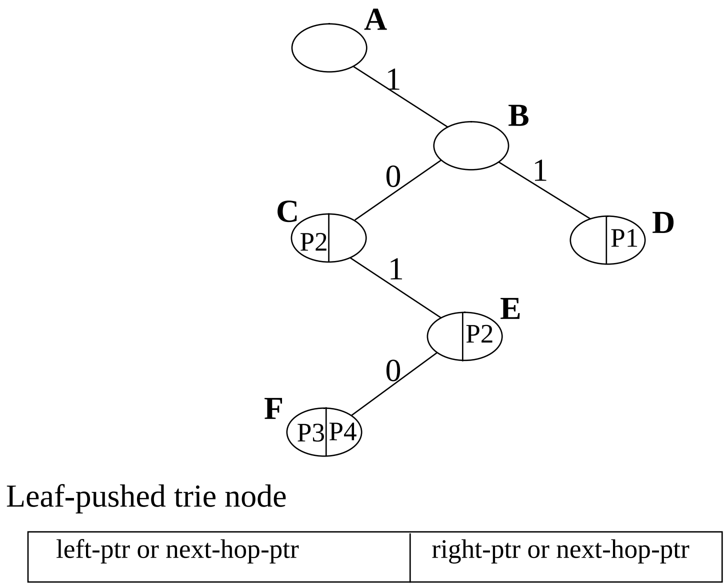


Figure 2.2 A leaf-pushed binary trie storing the prefixes of Table 2.1.

As Figure 2.1 shows, each trie node keeps a pointer each to its children nodes, and if it contains a prefix, also a pointer to the actual forwarding table entry (to recover the next-hop address). Storage space for the pointer can be saved by ‘pushing’ the prefixes to the leaves of the trie so that no internal node of the trie contains a prefix. Such a trie is referred to as a leaf-pushed trie, and is shown in Figure 2.2 for the binary trie of Figure 2.1. Note that this may lead to replication of the same next-hop pointer at several trie nodes.

2.1.4 PATRICIA[1]

A Patricia tree is a variation of a trie data structure, with the difference that it has no 1-degree nodes. Each chain is compressed to a single node in a Patricia tree. Hence, the traversal algorithm may not necessarily inspect all bits of the address consecutively, skipping over bits that formed part of the label of some previous trie chain. Each node now stores an additional field denoting the bit-position in the address that determines the next branch

[1] PATRICIA is an abbreviation for “Practical Algorithm To Retrieve Information Coded In Alphanumeric”. It is simply written as “Patricia” in normal text.

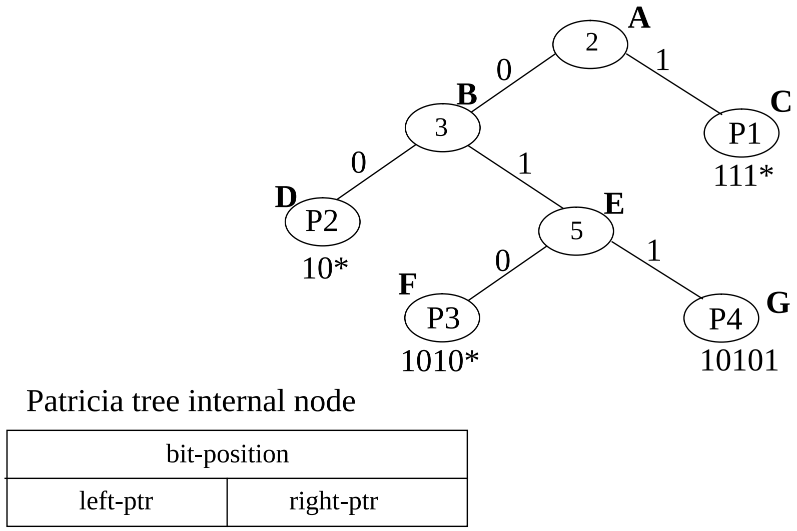


Figure 2.3 The Patricia tree for the example routing table in Table 2.1. The numbers inside the internal nodes denote bit-positions (the most significant bit position is numbered 1). The leaves store the complete key values.

to be taken at this node. The original Patricia tree [64] did not have support for prefixes. However, prefixes can be concatenated with trailing zeroes and added to a Patricia tree. Figure 2.3 shows the Patricia tree for our running example of the routing table. Since a Patricia tree is a complete binary tree (i.e., has nodes of degree either 0 or 2), it has exactly N external nodes (leaves) and $N-1$ internal nodes. The space complexity of a Patricia tree is thus $O(N)$.

Prefixes are stored in the leaves of a Patricia tree. A leaf node may have to keep a linear list of prefixes, because prefixes are concatenated with trailing zeroes. The lookup algorithm descends the tree from the root node to a leaf node similar to that in a trie. At each node, it probes the address for the bit indicated by the bit-position field in the node. The value of this bit determines the branch to be taken out of the node. When the algorithm reaches a leaf, it attempts to match the address with the prefix stored at the leaf. This prefix is the desired answer if a match is found. Otherwise, the algorithm has to recursively backtrack and continue the search in the other branch of this leaf's parent node.

Hence, the lookup complexity in a Patricia tree is quite high, and can reach $O(W^2)$ in the worst case.

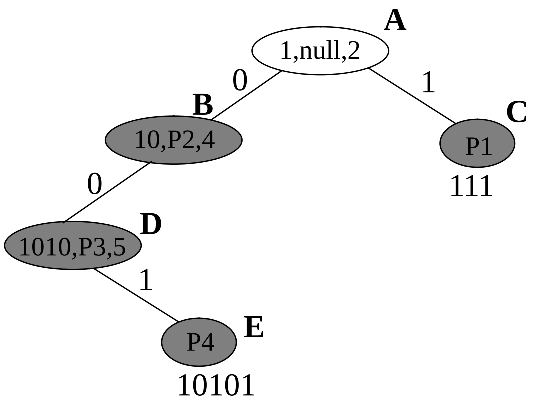
Example 2.2: Give an incoming 5-bit address 10111 to be looked up in the Patricia tree of Figure 2.3, the longest prefix matching algorithm takes the path ABEG, and compares the address to the prefix stored in leaf node G. Since it does not match, the algorithm backtracks to the parent node E and tries to compare the address to the prefix stored in leaf node F. Since it does not match again, the algorithm backtracks to the parent node B and finally matches prefix P2 in node D.

Instead of storing prefixes concatenated with trailing zeros as above, a longest prefix matching algorithm may also form a data structure with W different Patricia trees — one for each of the W prefix lengths. The algorithm searches for an exact match in each of the trees in decreasing order of prefix-lengths. The first match found yields the longest prefix matching the given address. One exact match operation on a Patricia tree takes $O(W)$ time. Hence, a longest prefix matching operation on this data structure will take $O(W^2)$ time and still have $O(N)$ storage complexity.

2.1.5 Path-compressed trie

A Patricia tree loses information while compressing chains because it remembers only the label on the last branch comprising the chain — the bit-string represented by the other branches of the uncompressed chain is lost. Unlike a Patricia trie, a path-compressed trie node stores the complete bit-string that the node would represent in the uncompressed basic trie. The lookup algorithm matches the address with this bit-string before traversing the subtrie rooted at that node. This eliminates the need for backtracking and decreases lookup time to at most W memory accesses. The storage complexity remains $O(N)$. The path-compressed trie for the example forwarding table of Table 2.1 is shown in Figure 2.4.

Example 2.3: Give an incoming 5-bit address 10111 to be looked up in the path-compressed trie of Figure 2.4, the longest prefix matching algorithm takes path AB and encounters a null pointer on the right branch at node B. Hence, the most recently encountered



Path-compressed trie node

<table>
<tr><td colspan="2">variable-length bitstring</td><td colspan="2">next-hop (if prefix present)</td><td>bit-position</td></tr>
<tr><td colspan="3">left-ptr</td><td colspan="2">right-ptr</td></tr>
</table>

Figure 2.4 The path-compressed trie for the example routing table in Table 2.1. Each node is represented by (bitstring,next-hop,bit-position).

prefix P2, stored in node B, yields the desired longest matching prefix for the given address.

2.2 Previous work on route lookups

2.2.1 Early lookup schemes

The route lookup implementation in BSD unix [90][98] uses a Patricia tree and avoids implementing recursion by keeping explicit parent pointers in every node. Reference [90] reports that the expected length of a search on a Patricia tree with N non-prefix entries is $1.44 \log N$. This implies a total of 24 bit tests and 24 memory accesses for $N = 98,000$ prefixes. Doeringer et al [19] propose the *dynamic prefix trie* data structure — a variant of the Patricia data structure that supports non-recursive search and update operations. Each node of this data structure has six fields — five fields contain pointers to other nodes of the data structure and one field stores a bit-index to guide the search algorithm as in a Patricia tree. A lookup operation requires two traversals along the tree, the first traversal descends the tree to a leaf node and the second backtracks to find the longest prefix matching the given address. The insertion and deletion algorithms as reported in [19] need to handle a number of special cases and seem difficult to implement in hardware.

2.2.2 Multi-ary trie and controlled prefix expansion

A binary trie inspects one bit at a time, and potentially has a depth of W for W-bit addresses. The maximum depth can be decreased to W/k by inspecting k bits at a time. This is achieved by increasing the degree of each internal node to 2^k. The resulting trie is called a 2^k-way or 2^k-ary trie, and has a maximum of W/k levels. The number of bits inspected by the lookup algorithm at each trie node, k, is referred to as the stride of the trie. While multi-ary tries have been discussed previously by researchers (e.g., see page 496 of [46], page 408 of [86]), the first detailed exposition in relation to prefixes and routing tables can be found in [97].

Prefixes are stored in a multi-ary trie in the following manner: If the length of a prefix is an integral multiple of k, say mk, the prefix is stored at level m of the trie. Otherwise, a prefix of length that is not a multiple of k needs to be *expanded* to form multiple prefixes, all of whose lengths are integer multiples of k. For example, a prefix of length $k - 1$ needs to be expanded to two prefixes of length k each, that can then be stored in a 2^k-ary trie.

Example 2.4: The 4-ary trie to store the prefixes in the forwarding table of Table 2.1 is shown in Figure 2.5. While prefixes P2 and P3 are stored directly without expansion, the lengths of prefixes P1 and P4 are not multiples of 2 and hence these prefixes need to be expanded. P1 expands to form the prefixes $P1_1$ and $P1_2$, while P4 expands to form prefixes $P4_1$ and $P4_2$. All prefixes are now of lengths either 2 or 4.

Expansion of prefixes increases the storage consumption of the multi-ary trie data structure because of two reasons: (1) The next-hop corresponding to a prefix needs to be stored in multiple trie nodes after expansion; (2) There is a greater number of unused (null) pointers in a node. For example, there are 8 nodes, 7 branches, and $8 \times 2 - 7 = 9$

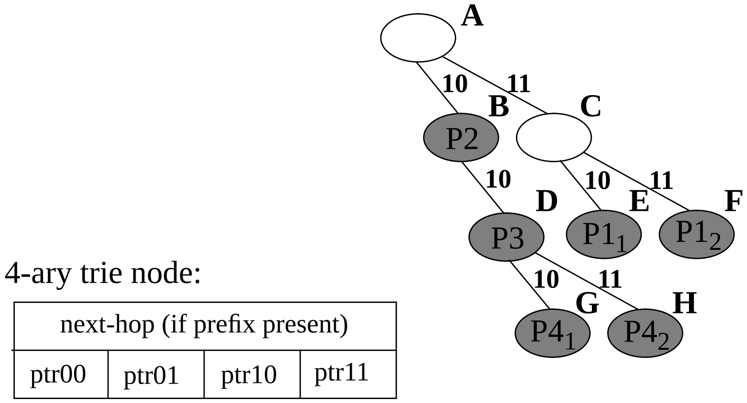


Figure 2.5 A 4-ary trie storing the prefixes of Table 2.1. The gray nodes store pointers to next-hops.

null pointers in the binary trie of Figure 2.1, while there are 8 nodes, 7 branches, and $8 \times 4 - 7 = 25$ null pointers in the 4-ary trie of Figure 2.5. The decreased lookup time therefore comes at the cost of increased storage space requirements. The degree of expansion controls this trade-off of storage versus speed in the multi-ary trie data structure.

Each node of the expanded trie is represented by an array of pointers. This array has size 2^k and the pointer at index j of the array represents the branch numbered j and points to the child node at that branch.

A generalization of this idea is to have different strides at each level of the (expanded) trie. For example, a 32-bit binary trie can be expanded to create a four-level expanded trie with any of the following sequence of strides: 10,10,8,4; or 8,8,8,8, and so on. Srinivasan et al [93][97] discuss these variations in greater detail. They propose an elegant dynamic programming algorithm to compute the optimal sequence of strides that, given a forwarding table and a desired maximum number of levels, minimizes the storage requirements of the expanded trie (called a fixed-stride trie) data structure. The algorithm runs in $O(W^2D)$ time, where D is the desired maximum depth. However, updates to a fixed-stride trie could result in a suboptimal sequence of strides and need costly re-runs of the dynamic programming optimization algorithm. Furthermore, implementation of a trie whose strides depend on the properties of the forwarding table may be too complicated to perform in hardware.

The authors [93][97] extend the idea further by allowing each trie node to have a different stride, and call the resulting trie a variable-stride trie. They propose another dynamic programming algorithm, that, given a forwarding table and a maximum depth D, computes the optimal stride at each trie node to minimize the total storage consumed by the variable-stride trie data structure. The algorithm runs in $O(NW^2D)$ time for a forwarding table with N prefixes.

Measurements in [97] (see page 61) report that the dynamic programming algorithm takes 1 ms on a 300 MHz Pentium-II processor to compute an optimal fixed-stride trie for a forwarding table with 38,816 prefixes. This table is obtained from the MAE-EAST NAP (source [124]). We will call this forwarding table the reference MAE-EAST forwarding table as it will be used for comparison of the different algorithms proposed in this section. This trie has a storage requirement of 49 Mbytes for two levels and 1.8 Mbytes for three levels. The dynamic programming algorithm that computes the optimal variable-stride trie computes a data structure that consumes 1.6 Mbytes for 2 levels in 130 ms, and 0.57 Mbytes for 3 levels in 871 ms.

2.2.3 Level-compressed trie (LC-trie)

We saw earlier that expansion compresses the number of levels in a trie at the cost of increased storage space. Space is especially wasted in the sparsely populated portions of the trie, which are themselves better compressed by the technique of path compression mentioned in Section 2.1.5. Nilsson [69] introduces the LC-trie, a trie structure with combined path and level compression. An LC-trie is created from a binary trie as follows. First, path compression is applied to the binary trie. Second, every node v that is rooted at a complete subtrie of maximum depth k is expanded to create a 2^k-degree node v'. The leaves of the subtrie rooted at node v' in the basic trie become the 2^k children of v'. This expansion is carried out recursively on each subtrie of the basic trie This is done with the motivation of minimizing storage while still having a small number of levels in the trie. An example of an LC-trie is shown in Figure 2.6.

The construction of an LC-trie for N prefixes takes $O(N\log N)$ time [69]. Incremental updates are not supported. Reference [97] notes that an LC-trie is a special case of a variable-stride trie, and the dynamic programming optimization algorithm of [97] would indeed result in the LC-trie if it were the optimal solution for a given set of prefixes. The LC-trie data structure consumes 0.7 Mbytes on the reference MAE-EAST forwarding table consisting of 38,816 prefixes and has 7 levels. This is worse than the 4-level optimal variable-stride trie, which consumes 0.4 Mbytes [97].

2.2.4 The Lulea algorithm

The Lulea algorithm, proposed by Degermark et al [17], is motivated by the objective of minimizing the storage requirements of their data structure, so that it can fit in the L1-cache of a conventional general purpose processor (e.g., Pentium or Alpha processor). Their algorithm expands the 32-bit binary trie to a three-level leaf-pushed trie with the stride sequence of 16, 8 and 8. Each level is optimized separately. We discuss some of the optimizations in this subsection and refer the reader to [17] for more details.

The first optimization reduces the storage consumption of an array when a number of consecutive array elements have the same value; i.e., there are Q distinct elements in the array of size M, with $Q \ll M$. For example, an 8-element array that has values ABBBBCCD could be represented by two arrays: one array, *bitarr*, stores the 8 bits 1100101, and the second array, *valarr*, stores the actual pointer values ABCD. The value of an element at a location j is accessed by first counting the number of bits that are '1' in *bitarr[1..j]*, say p, and then accessing *valarr[p]*.

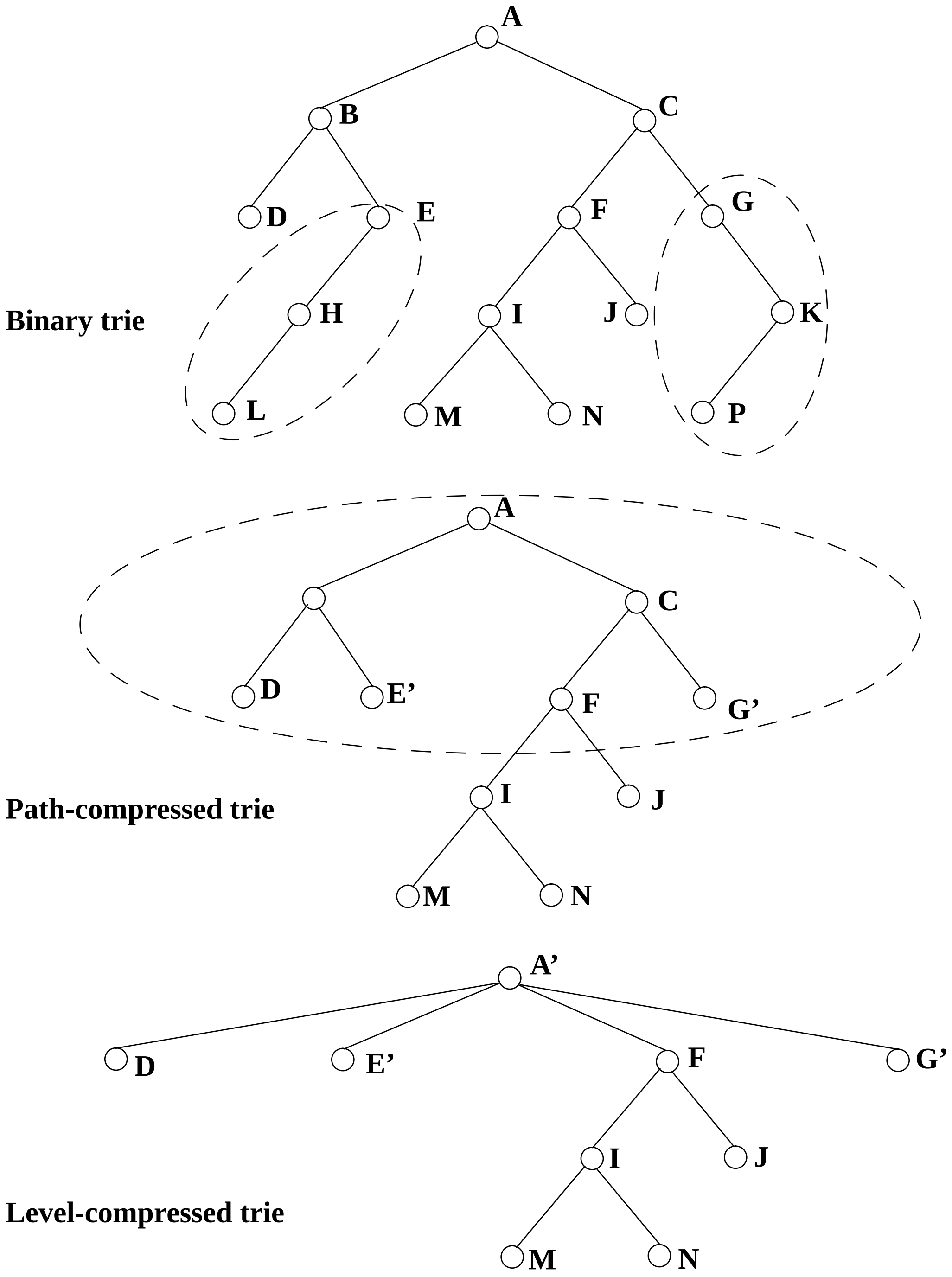


Figure 2.6 An example of an LC-trie. The binary trie is first path-compressed (compressed nodes are circled). Resulting nodes rooted at complete subtries are then expanded. The end result is a trie which has nodes of different degrees.

Hence, an array of M V-bit elements, with Q of them containing distinct values, consumes MV bits when the elements are stored directly in the array, and $M + QV$ bits with this optimization. The optimization, however, comes with two costs incurred at the time the array is accessed: (1) the appropriate number of bits that are '1' need to be counted, and (2) two memory accesses need to be made.

The Lulea algorithm applies this idea to the root node of the trie that contains $2^{16} = 64K$ pointers (either to the next-hop or to a node in the next level). As we saw in Section 2.2.2, pointers at several consecutive locations could have the same value if they are the next-hop pointers of a shorter prefix that has been expanded to 16 bits. Storage space can thus be saved by the optimization mentioned above. In order to decrease the cost of counting the bits in the 64K-wide bitmap, the algorithm divides the bitmap into 16-bit chunks and keeps a precomputed sum of the bits that are '1' in another array, *base_ptr*, of size $(64K) / 16 = 4K$ bits.

The second optimization made by the Lulea algorithm eliminates the need to store the 64K-wide bitmap. They note that the 16-bit bitmap values are not arbitrary. Instead, they are derived from complete binary trees, and hence are much fewer in number (678 [17]) than the maximum possible 2^{16}. This allows them to encode each bitmap by a 10-bit number (called codeword) and use another auxiliary table, called *maptable*, a two-dimensional array of size $10,848 = 678 \times 16$. *maptable[c][j]* gives the precomputed number of bits that are '1' in the 16-bit bitmap corresponding to codeword c before the bit-position j. This has the net effect of replacing the need to count the number of bits that are '1' with an additional memory access into *maptable*.

The Lulea algorithm makes similar optimizations at the second and third levels of the trie. These optimizations decrease the data structure storage requirements to approximately 160 Kbytes for the reference forwarding table with 38,816 prefixes — an average of only 4.2 bytes per prefix. However, the optimizations made by the Lulea algorithm have two disadvantages:

1. It is difficult to support incremental updates in the (heavily-optimized) data structure. For example, an addition of a new prefix may lead to a change in all the entries of the precomputed array *base_ptr.*
2. The benefits of the optimizations are dependent on the structure of the forwarding table. Hence, it is difficult to predict the worst-case storage requirements of the data structure as a function of the number of prefixes.

2.2.5 Binary search on prefix lengths

The longest prefix matching operation can be decomposed into W exact match search operations, one each on prefixes of fixed length. This decomposition can be viewed as a linear search of the space $1\ldots W$ of prefix lengths, or equivalently binary-trie levels. An algorithm that performs a binary search on this space has been proposed by Waldvogel et al [108]. This algorithm uses hashing for an exact match search operation among prefixes of the same length.

Given an incoming address, a linear search on the space of prefix lengths requires probing each of the W hash tables, $H_1\ldots H_W$, — which requires W hash operations and W hashed memory accesses.[1] The binary search algorithm [108] stores in H_j, not only the prefixes of length j, but also the internal trie nodes (called *markers* in [108]) at level j. The algorithm first probes $H_{W/2}$. If a node is found in this hash table, there is no need to probe tables $H_1\ldots H_{W/2-1}$. If no node is found, hash tables $H_{W/2+1}\ldots H_W$ need not be probed. The remaining hash tables are similarly probed in a binary search manner. This requires $O(\log W)$ hashed memory accesses for one lookup operation. This data structure has storage complexity $O(NW)$ since there could be up to W markers for a prefix — each internal node in the trie on the path from the root node to the prefix is a marker. Reference

[1] A hashed memory access takes $O(1)$ time on average. However, the worst case could be $O(N)$ in the pathological case of a collision among all N hashed elements.

[108] notes that not all W markers need actually be kept. Only the $\log W$ markers that would be probed by the binary search algorithm need be stored in the corresponding hash tables — for instance, an IPv4 prefix of length 22 needs markers only for prefix lengths 16 and 20. This decreases the storage complexity to $O(N \log W)$.

The idea of binary search on trie levels can be combined with prefix expansion. For example, binary search on the levels of a 2^k-ary trie can be performed in time $O(\log(W/k))$ and storage $O(N2^k + N\log(W/k))$.

Binary search on trie levels is an elegant idea. The lookup time scales logarithmically with address length. The idea could be used for performing lookups in IPv6 (the next version of IP) which has 128-bit addresses. Measurements on IPv4 routing tables [108], however, do not indicate significant performance improvements over other proposed algorithms, such as trie expansion or the Lulea algorithm. Incremental insertion and deletion operations are also not supported, because of the several optimizations performed by the algorithm to keep the storage requirements of the data structure small [108].

2.2.6 Binary search on intervals represented by prefixes

We saw in Section 1.2 of Chapter 1 that each prefix represents an interval (a contiguous range) of addresses. Because longer prefixes represent shorter intervals, finding the longest prefix matching a given address is equivalent to finding the narrowest enclosing interval of the point represented by the address. Figure 2.7(a) represents the prefixes in the example forwarding table of Table 2.1 on a number line that stretches from address 00000 to 11111. Prefix P3 is the longest prefix matching address 10100 because the interval $[10100\ldots10101]$ represented by P3 encloses the point 10100, and is the narrowest such interval.

The intervals created by the prefixes partition the number line into a set of disjoint intervals (called basic intervals) between consecutive end-points (see Figure 2.7(b)).

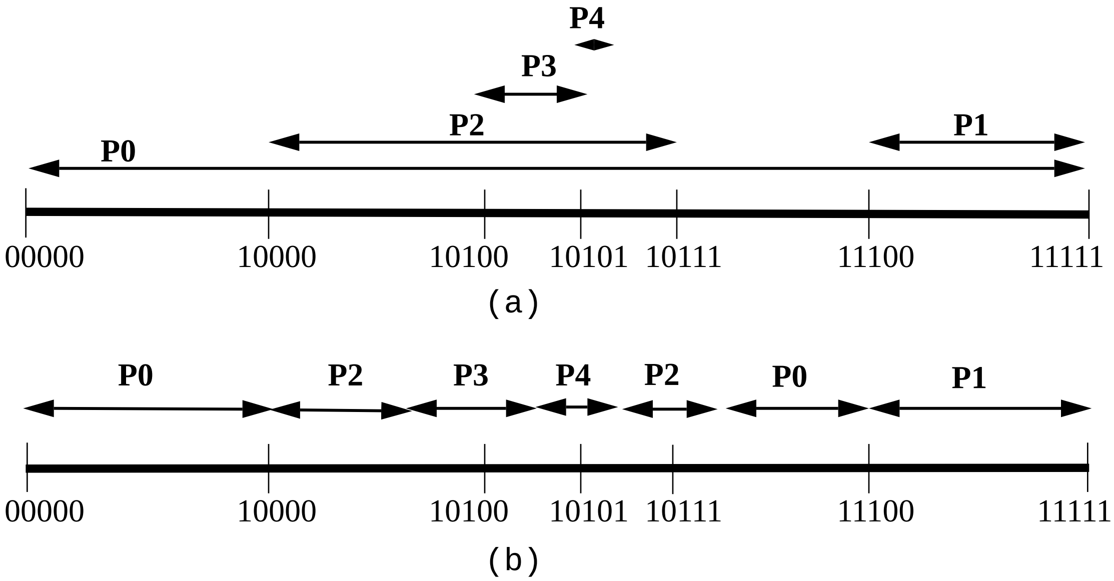


Figure 2.7 (not drawn to scale) (a) shows the intervals represented by prefixes of Table 2.1. Prefix P0 is the "default" prefix. The figure shows that finding the longest matching prefix is equivalent to finding the narrowest enclosing interval. (b) shows the partitioning of the number line into disjoint intervals created from (a). This partition can be represented by a sorted list of end-points.

Lampson et al [49] suggest an algorithm that precomputes the longest prefix for every basic interval in the partition. If we associate every basic interval with its left end-point, the partition could be stored by a sorted list of left-endpoints of the basic intervals. The longest prefix matching problem then reduces to the problem of finding the closest left end-point in this list, i.e., the value in the sorted list that is the largest value not greater than the given address. This can be found by a binary search on the sorted list.

Each prefix contributes two end-points, and hence the size of the sorted list is at most $2N+1$ (including the leftmost point of the number line). One lookup operation therefore takes $O(\log(2N))$ time and $O(N)$ storage space. It is again difficult to support fast incremental updates in the worst case, because insertion or deletion of a (short) prefix can change the longest matching prefixes of several basic intervals in the partition.[1] In our

[1] This should not happen too often in the average case. Also note that the binary search tree itself needs to be updated with up to two new values on the insertion or deletion of a prefix.

simple example of Figure 2.7(b), deletion of prefix P2 requires changing the associated longest matching prefix of two basic intervals to P0.

Reference [49] describes a modified scheme that uses expansion at the root and implements a multiway search (instead of a binary search) on the sorted list in order to (1) decrease the number of memory accesses required and (2) take advantage of the cache-line size of high speed processors. Measurements for a 16-bit expansion at the root and a 6-way search algorithm on the reference MAE-EAST forwarding table with 38,816 entries showed a worst-case lookup time of 490 ns, storage of 0.95 Mbytes, build time of 5.8 s, and insertion time of around 350 ms on a 200 MHz Pentium Pro with 256 Kbytes of L2 cache.

TABLE 2.2. Complexity comparison of the different lookup algorithms. A '-' in the update column denotes that incremental updates are not supported. A '-' in the row corresponding to the Lulea scheme denotes that it is not possible to analyze the complexity of this algorithm because it is dependent on the structure of the forwarding table.

Algorithm	**Lookup complexity**	**Storage complexity**	**Update-time complexity**
Binary trie	W	NW	W
Patricia	W^2	N	W
Path-compressed trie	W	N	W
Multi-ary trie	W/k	$2^k NW/k$	-
LC-trie	W/k	$2^k NW/k$	-
Lulea scheme	-	-	-
Binary search on lengths	$\log W$	$N \log W$	-
Binary search on intervals	$\log(2N)$	N	-
Theoretical lower bound [102]	$\log W$	N	-

2.2.7 Summary of previous algorithms

Table 2.2 gives a summary of the complexities, and Table 2.3 gives a summary of the performance numbers (reproduced from [97], page 42) of the algorithms reviewed in Section 2.2.1 to Section 2.2.6. Note that each algorithm was developed with a software implementation in mind.

TABLE 2.3. Performance comparison of different lookup algorithms.

Algorithm	**Worst-case lookup time on 300 MHz Pentium-II with 15ns 512KB L2 cache (ns).**	**Storage requirements (Kbytes) on the reference MAE-EAST forwarding table consisting of 38,816 prefixes, taken from [124].**
Patricia (BSD)	2500	3262
Multi-way fixed-stride optimal trie (3-levels)	298	1930
Multi-way fixed stride optimal trie (5 levels)	428	660
LC-trie	-	700
Lulea scheme	409	160
Binary search on lengths	650	1600
6-way search on intervals	490	950

2.2.8 Previous work on lookups in hardware: CAMs

The primary motivation for hardware implementation of the lookup function comes from the need for higher packet processing capacity (at OC48c or OC192c speeds) that is typically not obtainable by software implementations. For instance, almost all high speed products from major router vendors today perform route lookups in hardware.[1] A software implementation has the advantage of being more flexible, and can be easily adapted in case of modifications to the protocol. However, it seems that the need for flexibility within

[1] For instance, the OC48c linecards built by Cisco [120], Juniper [126] and Lucent [128] use silicon-based forwarding engines.

the IPv4 route lookup function should be minimal — IPv4 is in such widespread use that changes to either the addressing architecture or the longest prefix matching mechanism seem to be unlikely in the foreseeable future.

A fully associative memory, or content-addressable memory (CAM), can be used to perform an exact match search operation in hardware in a single clock cycle. A CAM takes as input a search key, compares the key in parallel with all the elements stored in its memory array, and gives as output the memory address at which the matching element was stored. If some data is associated with the stored elements, this data can also be returned as output. Now, a longest prefix matching operation on 32-bit IP addresses can be performed by an exact match search in 32 separate CAMs [45][52]. This is clearly an expensive solution: each of the 32 CAMs needs to be big enough to store N prefixes in absence of apriori knowledge of the prefix length distribution (i.e., the number of prefixes of a certain length).

A better solution is to use a ternary-CAM (TCAM), a more flexible type of CAM that enables comparisons of the input key with variable length elements. Assume that each element can be of length from 1 to W bits. A TCAM stores an element as a (*val*, *mask*) pair; where *val* and *mask* are each W-bit numbers. If the element is Y bits wide, $1 \le Y \le W$, the most significant Y bits of the *val* field are made equal to the value of the element, and the most significant Y bits of the *mask* are made '1.' The remaining $(W - Y)$ bits of the *mask* are '0.' The *mask* is thus used to denote the length of an element. The least significant $(W - Y)$ bits of *val* can be set to either '0' or '1,' and are "don't care" (i.e., ignored).[1] For example, if $W = 5$, a prefix 10* will be stored as the pair (10000, 11000). An element matches a given input key by checking if those bits of *val* for which the *mask* bit is '1' are

[1] In effect, a TCAM stores each bit of the element as one of three possible values (0,1,X) where X represents a wildcard, or a don't care bit. This is more powerful than needed for storing prefixes, but we will see the need for this in Chapter 4, when we discuss packet classification.

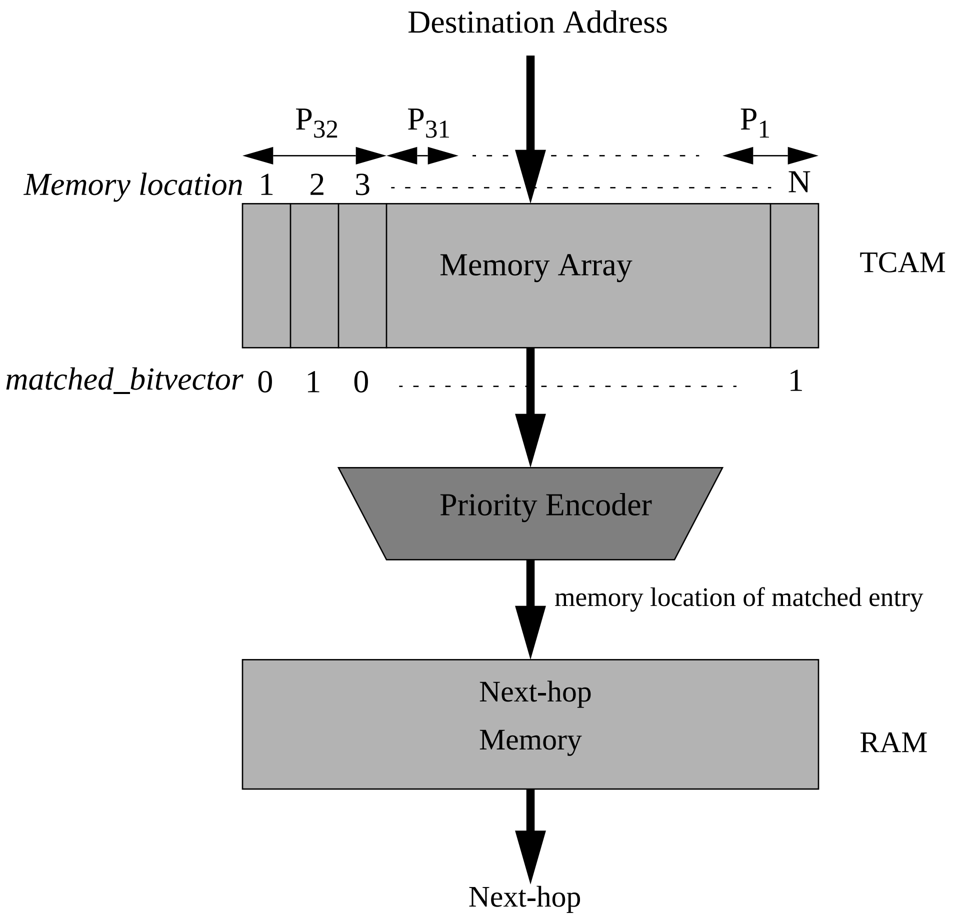


Figure 2.8 Showing the lookup operation using a ternary-CAM. P_i denotes the set of prefixes of length *i*.

identical to those in the key. In other words, (*val*, *mask*) matches an input *key* if (*val* & *m*) equals (*key* & *m*), where & denotes the bitwise-AND operation and *m* denotes the *mask*.

A TCAM is used for longest prefix matching in the manner indicated by Figure 2.8. The TCAM memory array stores prefixes as (*val*, *mask*) pairs in decreasing order of prefix lengths. The memory array compares a given input key with each element. It follows by definition that an element (*val*, *mask*) matches the key if and only if it is a prefix of that key. The memory array indicates the matched elements by setting corresponding bits in the N-bit bitvector, *matched_bv*, to '1.' The location of the longest matching prefix can then be obtained by using an N-bit priority encoder that takes in *matched_bv* as input, and outputs the location of the lowest bit that is ‘1’ in the bitvector. This is then used as an address to a RAM to access the next-hop associated with this prefix.

A TCAM has the advantages of speed and simplicity. However, there are two main disadvantages of TCAMs:

1. A TCAM is more expensive and can store fewer bits in the same chip area as compared to a random access memory (RAM) — one bit in an SRAM typically requires 4-6 transistors, while one bit in a TCAM typically requires 11-15 transistors (two SRAM cells plus at least 3 transistors [87]). A 2 Mb TCAM (biggest TCAM in production at the time of writing) running at 50-100 MHz costs about $60-$70 today, while an 8 Mb SRAM (biggest SRAM commonly available at the time of writing) running at 200 MHz costs about $20-$40. Note that one needs at least $512K \times 32b = 16$ Mb of TCAM to support 512K prefixes. This can be achieved today by *depth-cascading* (a technique to increase the depth of a CAM) eight ternary-CAMs, further increasing the system cost. Newer TCAMs, based on a dynamic cell similar to that used in a DRAM, have also been proposed [130], and are attractive because they can achieve higher densities. One, as yet unsolved, issue with such DRAM-based CAMs is the presence of hard-to-detect soft errors caused by alpha particles in the dynamic memory cells.[1]
2. A TCAM dissipates a large amount of power because the circuitry of a TCAM row (that stores one element) is such that electric current is drawn in every row that has an unmatched prefix. An incoming address matches at most W prefixes, one of each length — hence, most of the elements are unmatched. Because of this reason, a TCAM consumes a lot of power even under the *normal* mode of operation. This is to be contrasted with an SRAM, where the normal mode of operation results in electric current being drawn only by the element accessed at the input memory address. At the time of writing, a 2 Mb TCAM chip running at 50 MHz dissipates about 5-8 watts of power [127][131].

[1] Detection and correction of soft errors is easier in *random access* dynamic memories, because only one row is accessed in one memory operation. Usually, one keeps an error detection/correction code (EC) with each memory row, and verifies the EC upon accessing a row. This does not apply in a CAM because all memory rows are accessed simultaneously, while only one result is made available as output. Hence, it is difficult to verify the EC for all rows in one search operation. One possibility is to include the EC with each element in the CAM and require that a match be indicated only if both the element and its EC match the incoming key and the expected EC. This approach however does not take care of elements that should have been matched, but do not because of memory errors. Also, this mechanism does not work for ternary CAM elements because of the presence of wildcarded bits.

An important issue concerns fast incremental updates in a TCAM. As elements need to be sorted in decreasing order of prefix lengths, the addition of a prefix may require a large number of elements to be shifted. This can be avoided by keeping unused elements between the set of prefixes of length i and $i + 1$. However, that wastes space and only improves the average case update time. An optimal algorithm for managing the empty space in a TCAM has been proposed in [88].

In summary, TCAMs have become denser and faster over the years, but still remain a costly solution for the IPv4 route lookup problem.

3 Proposed algorithm

The algorithm proposed in this section is motivated by the need for an inexpensive and fast lookup solution that can be implemented in pipelined hardware, and that can handle updates with low overhead to the central processor. This section first discusses the assumptions and the key observations that form the basis of the algorithm, followed by the details of the algorithm.

3.1 Assumptions

The algorithm proposed in this section is specific to IPv4 and does not scale to IPv6, the next version of IP. It is based on the assumption that a hardware solution optimized for IPv4 will be useful for a number of years because of the continued popularity of IPv4 and delayed widespread use of IPv6 in the Internet. IPv6 was introduced in 1995 to eliminate the impending problem of IPv4 address space exhaustion and uses 128-bit addresses instead of 32-bit IPv4 addresses. Our assumption is supported by the observation that IPv6 has seen only limited deployment to date, probably because of a combination of the following reasons:

1. ISPs are reluctant to convert their network to use an untested technology, particularly a completely new Internet protocol.
2. The industry has meanwhile developed other techniques (such as network address translation, or NAT [132]) that alleviate the address space exhaustion problem by enabling reuse of IPv4 addresses inside administrative domains (for instance, large portions of the networks in China and Microsoft are behind network elements performing NAT).
3. The addressing and routing architecture in IPv6 has led to new technical issues in areas such as multicast and multi-homing. We do not discuss these issues in detail here, but refer the reader to [20][125].

3.2 Observations

The route lookup scheme presented here is based on the following two key observations:

1. Because of route-aggregation at intermediate routers (mentioned in Chapter 1), routing tables at higher speed backbone routers contain *few entries with prefixes longer than 24-bits*. This is verified by a plot of prefix length distribution of the backbone routing tables taken from the PAIX NAP on April 11, 2000 [124], as shown in Figure 2.9 (note the logarithmic scale on the y-axis). In this example, 99.93% of the prefixes are 24-bits or less. A similar prefix length distribution is seen in the routing tables at other backbone routers. Also, this distribution has hardly changed over time.
2. *DRAM memory is cheap,* and continues to get cheaper by a factor of approximately two every year. 64 Mbytes of SDRAM (synchronous DRAM) cost around $50 in April 2000 [129]. Memory densities are following Moore's law and doubling every eighteen months. The net result is that a large amount of memory is available at low cost. This observation provides the motivation for trading off large amounts of memory for lookup speed. This is in contrast to most of the previous work (mentioned in Section 2.2) that seeks to minimize the storage requirements of the data structure.

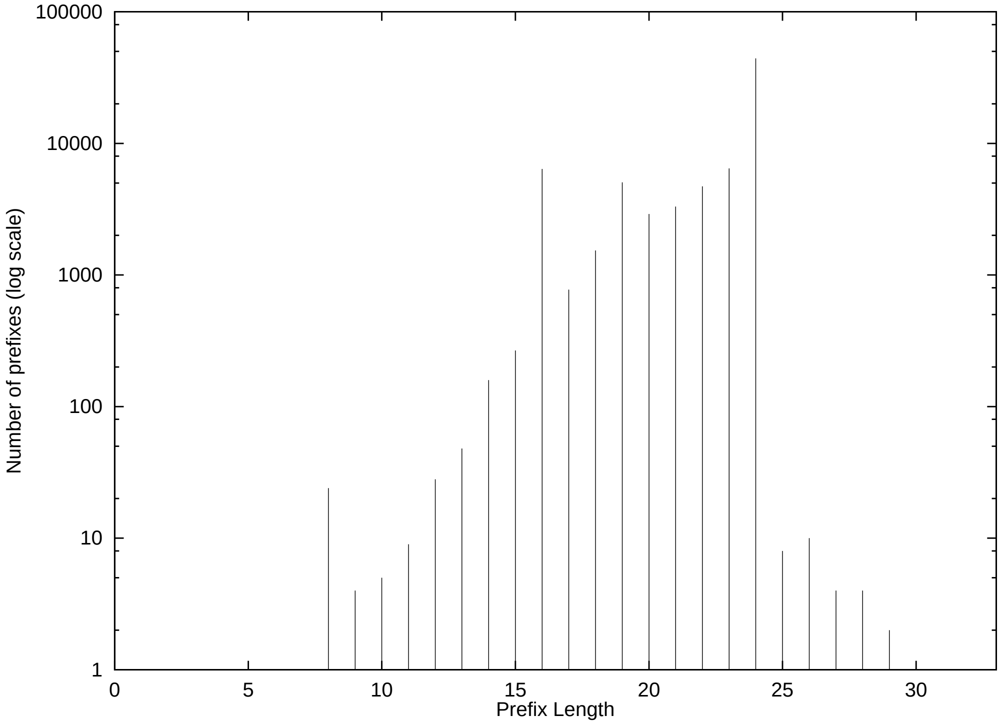


Figure 2.9 The distribution of prefix lengths in the PAIX routing table on April 11, 2000. (Source: [124]). The number of prefixes longer than 24 bits is less than 0.07%.

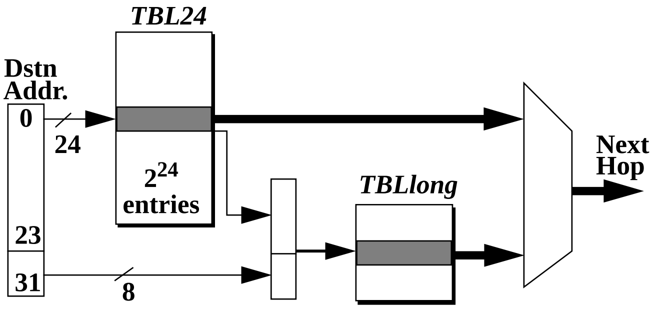


Figure 2.10 Proposed *DIR-24-8-BASIC* architecture. The next-hop result comes from either *TBL24* or *TBLlong*.

3.3 Basic scheme

The basic scheme, called *DIR-24-8-BASIC*, makes use of the two tables shown in Figure 2.10. The first table (called *TBL24*) stores all possible route-prefixes that are up to, and

If longest prefix with this 24-bit prefix is < 25 bits long:

0	**Next-hop**
1 bit	**15 bits**

If longest prefix with this 24 bits prefix is > 24 bits long:

1	**Index into 2nd table TBLlong**
1 bit	**15 bits**

Figure 2.11 *TBL24* entry format

including, 24-bits long. This table has 2^{24} entries, addressed from 0 (corresponding to the 24-bits being 0.0.0) to $2^{24}-1$ (255.255.255). Each entry in *TBL24* has the format shown in Figure 2.11. The second table (*TBLlong*) stores all route-prefixes in the routing table that are longer than 24-bits. This scheme can be viewed as a fixed-stride trie with two levels: the first level with a stride of 24, and the second level with a stride of 8. We will refer to this as a (24,8) split of the 32-bit binary trie. In this sense, the scheme can be viewed as a special case of the general scheme of expanding tries [93].

A prefix, *X*, is stored in the following manner: if *X* is less than or equal to 24 bits long, it need only be stored in *TBL24*: the first bit of such an entry is set to zero to indicate that the remaining 15 bits designate the next-hop. If, on the other hand, prefix *X* is longer than 24 bits, the first bit of the entry indexed by the first 24 bits of *X* in *TBL24* is set to one to indicate that the remaining 15 bits contain a pointer to a set of entries in *TBLlong*.

In effect, route-prefixes shorter than 24-bits are expanded; e.g. the route-prefix 128.23.0.0/16 will have $2^{24-16} = 256$ entries associated with it in *TBL24*, ranging from the memory address 128.23.0 through 128.23.255. All 256 entries will have exactly the same contents (the next-hop corresponding to the route-prefix 128.23.0.0/16). By using memory inefficiently, we can find the next-hop information within one memory access.

TBLlong contains all route-prefixes that are longer than 24 bits. Each 24-bit prefix that has at least one route longer than 24 bits is allocated $2^8 = 256$ entries in *TBLlong*. Each entry in *TBLlong* corresponds to one of the 256 possible longer prefixes that share the single 24-bit prefix in *TBL24*. Note that because only the next-hop is stored in each entry of the second table, it need be only 1 byte wide (under the assumption that there are fewer than 255 next-hop routers – this assumption could be relaxed for wider memory.

Given an incoming destination address, the following steps are taken by the algorithm:

1. Using the first 24-bits of the address as an index into the first table *TBL24*, the algorithm performs a single memory read, yielding 2 bytes.
2. If the first bit equals zero, then the remaining 15 bits describe the next-hop.
3. Otherwise (i.e., if the first bit equals one), the algorithm multiplies the remaining 15 bits by 256, adds the product to the last 8 bits of the original destination address (achieved by shifting and concatenation), and uses this value as a direct index into *TBLlong*, which contains the next-hop.

3.3.1 Examples

Consider the following examples of how route lookups are performed using the table in Figure 2.12.

Example 2.5: Assume that the following routes are already in the table: 10.54.0.0/16, 10.54.34.0/24, 10.54.34.192/26. The first route requires entries in *TBL24* that correspond to the 24-bit prefixes 10.54.0 through 10.54.255 (except for 10.54.34). The second and third routes require that the second table be used (because both of them have the same first 24-bits and one of them is more than 24-bits long). So, in *TBL24*, the algorithm inserts a '1' bit, followed by an index (in the example, the index equals 123) into the entry corresponding to the 10.54.34 prefix. In the second table, 256 entries are allocated starting with memory location 123×256. Most of these locations are filled in with the next-hop corresponding to the 10.54.34 route, but 64 of them (those from $(123 \times 256) + 192$ to $(123 \times 256) + 255$) are filled in with the next-hop corresponding to the route-prefix 10.54.34.192.

We now consider some examples of packet lookups.

Example 2.6: If a packet arrives with the destination address 10.54.22.147, the first 24 bits are used as an index into *TBL24*, and will return an entry with the correct next-hop (A).

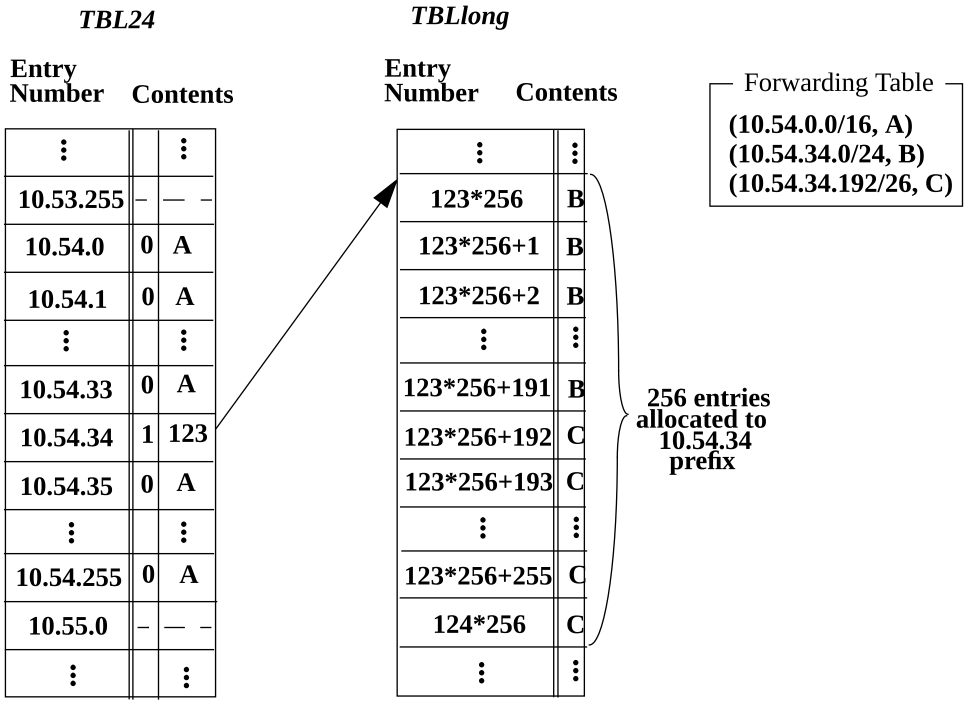


Figure 2.12 Example with three prefixes.

Example 2.7: If a packet arrives with the destination address 10.54.34.14, the first 24 bits are used as an index into the first table, which indicates that the second table must be consulted. The lower 15 bits of the *TBL24* entry (123 in this example) are combined with the lower 8 bits of the destination address and used as an index into the second table. After two memory accesses, the table returns the next-hop (B).

Example 2.8: If a packet arrives with the destination address 10.54.34.194, *TBL24* indicates that *TBLlong* must be consulted, and the lower 15 bits of the *TBL24* entry are combined with the lower 8 bits of the address to form an index into the second table. This time the next-hop (C) associated with the prefix 10.54.34.192/26 (C) is returned.

The size of second memory that stores the table *TBLlong* depends on the number of routes longer than 24 bits required to be supported. For example, the second memory needs to be 1 Mbyte in size for 4096 routes longer than 24 bits (to be precise, 4096 routes that are longer than 24 bits and have distinct 24-bit prefixes). We see from Figure 2.9 that the number of routes with length above 24 is much smaller than 4096 (only 31 for this router). Because 15 bits are used to index into *TBLlong*, 32K distinct 24-bit-prefixed long routes with prefixes longer than 24 bits can be supported with enough memory.

As a summary, we now review some of the pros and cons associated with the *DIR-24-8-BASIC* scheme.

Pros

1. Except for the limit on the number of distinct 24-bit-prefixed routes with length greater than 24 bits, this infrastructure will support an unlimited number of route-prefixes.
2. The design is well suited to hardware implementation. A reference implementation could, for example, store *TBL24* in either off-chip, or embedded SDRAM and *TBLlong* in on-chip SRAM or embedded-DRAM. Although (in general) two memory accesses are required, these accesses are in separate memories, allowing the scheme to be pipelined. When pipelined, 20 million packets per second can be processed with 50ns DRAM. The lookup time is thus equal to one memory access time.
3. The total cost of memory in this scheme is the cost of 33 Mbytes of DRAM (32 Mbytes for *TBL24* and 1 Mbyte for *TBLlong*), assuming *TBLlong* is also kept in DRAM. No special memory architectures are required.

Cons

1. Memory is used inefficiently.
2. Insertion and deletion of routes from this table may require many memory accesses, and a large overhead to the central processor. This is discussed in detail in Section 5.

4 Variations of the basic scheme

The basic scheme, *DIR-24-8-BASIC*, consumes a large amount of memory. This section proposes variations of the basic scheme with lower storage requirements, and explores the trade-off between storage requirements and the number of pipelined memory accesses.

4.1 Scheme *DIR-24-8-INT*: adding an intermediate "length" table

This variation is based on the observation that very few prefixes in a forwarding table that are longer than 24 bits are a full 32 bits long. For example, there are no 32-bit prefixes in the prefix-length distribution shown in Figure 2.9. The basic scheme, *DIR-24-8-BASIC*, allocates an entire block of 256 entries in table *TBLlong* for each prefix longer than 24 bits. This could waste memory — for example, a 26-bit prefix requires only $2^{26-24} = 4$ entries, but is allocated 256 *TBLlong* entries in the basic scheme.

The storage efficiency (amount of memory required per prefix) can be improved by using an additional level of indirection. This variation of the basic scheme, called *DIR-24-8-INT*, maintains an additional "intermediate" table, *TBLint*, as shown in Figure 2.13. An entry in *TBL24* that pointed to an entry in *TBLlong* in the basic scheme now points to an entry in *TBLint*. Each entry in *TBLint* corresponds to the unique 24-bit prefix represented by the *TBL24* entry that points to it. Therefore, *TBLint* needs to be M entries deep to support M prefixes that are longer than 24 bits and have distinct 24-bit prefixes.

Assume that an entry, e, of *TBLint* corresponds to the 24-bit prefix q. As shown in Figure 2.14, entry e contains a 21-bit index field into table *TBLlong*, and a 3-bit *prefix-length* field. The index field stores an absolute memory address in *TBLlong* at which the set of *TBLlong* entries associated with q begins. This set of *TBLlong* entries was always of size 256 in the basic scheme, but could be smaller in this scheme *DIR-24-8-INT*. The size of this set is encoded in the *prefix-length* field of entry e. The *prefix-length* field indicates the longest prefix in the forwarding table among the set of prefixes that have the first 24-bits identical to q. Three bits are sufficient because the length of this prefix must be in the range 25-32. The *prefix-length* field thus indicates how many entries in *TBLlong* are allocated to this 24-bit prefix q. For example, if the longest prefix is 30 bits long, then the *pre-*

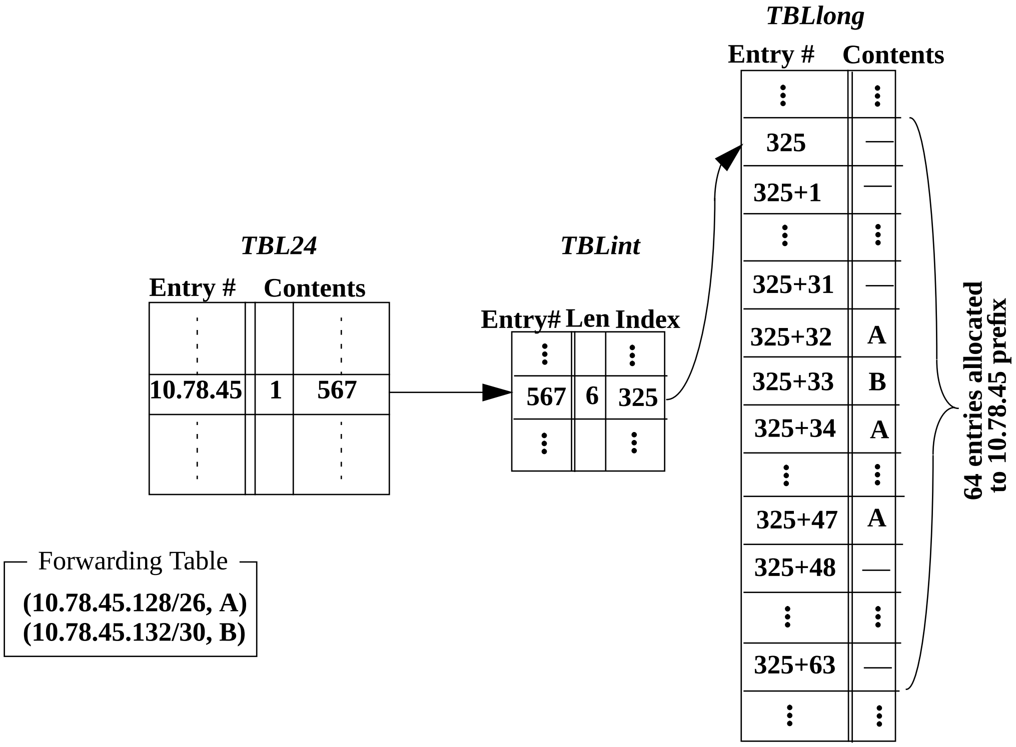


Figure 2.13 Scheme *DIR-24-8-INT*

index into 2nd table	max length
21 bits	3 bits

Figure 2.14 *TBLint* entry format.

fix-length field will store $30 - 24 = 6$, and *TBLlong* will have $2^6 = 64$ entries allocated to the 24-bit prefix q.

Example 2.9: (see Figure 2.13) Assume that two prefixes 10.78.45.128/26 and 10.78.45.132/30 are stored in the table. The entry in table *TBL24* corresponding to 10.78.45 will contain an index to an entry in *TBLint* (the index equals 567 in this example). Entry 567 in *TBLint* indicates a length of 6, and an index into *TBLlong* (the index equals 325 in the example) pointing to 64 entries. One of these entries, the 33rd (bits numbered 25 to 30 of prefix 10.78.45.132/30 are 100001, i.e., 33), contains the next-hop for the 10.78.45.132/30 route-prefix. Entry 32 and entries 34 through 47 (i.e., entries indicated by 10**** except 100001) contain the next-hop for the

10.78.45.128/26 route. The other entries contain the next-hop value for the default route.

The scheme *DIR-24-8-INT* improves utilization of table *TBLlong*, by an amount that depends on the distribution of the length of prefixes that are longer than 24-bits. For example, if the lengths of such prefixes were uniformly distributed in the range 25 to 32, 16K such prefixes could be stored in a total of 1.05 Mbytes of memory. This is because *TBLint* would require $16K \times 3B \approx 0.05MB$, and *TBLlong* would require $(16K) \times \left(\sum_{1\ldots8} 2^i \right) / 8 \times 1byte \approx 1MB$ of memory. In contrast, the basic scheme would require $16K \times 2^8 \times 1byte = 4MB$ to store the same number of prefixes. However, the modification to the basic scheme comes at the cost of an additional memory access, extending the pipeline to three stages.

4.2 Multiple table scheme

The modifications that we consider next split the 32-bit space into smaller subspaces so as to decrease the storage requirements. This can be viewed as a special case of the generalized technique of trie expansion discussed in Section 2.2.2. However, the objective here is to focus on a hardware implementation, and hence on the constraints posed by the worst-case scenarios, as opposed to generating an optimal sequence of strides that minimizes the storage consumption for a given forwarding table.

The first scheme, called *DIR-21-3*, extends the basic scheme *DIR-24-8-BASIC* to use three smaller tables instead of one large table (*TBL24*) and one small table (*TBLlong*). As an example, tables *TBL24* and *TBLlong* in scheme *DIR-24-8-BASIC* are replaced by a 2^{21} entry table (the “first” table, *TBLfirst21*), another 2^{21} entry table (the “second” table, *TBLsec21*), and a 2^{20} entry table (the “third” table, *TBLthird20*). The first 21 bits of the packet’s destination address are used to index into *TBLfirst21*, which has entries of width

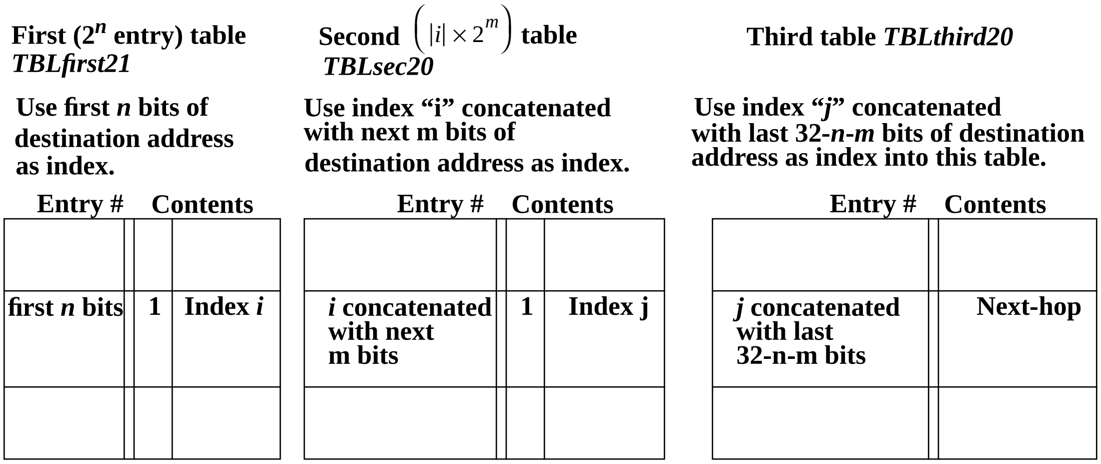

First (2^n entry) table
TBLfirst21
Use first *n* bits of destination address as index.
Entry #
Contents
first *n* bits
1
Index *i*
Second $\left(|i| \times 2^m\right)$ table
TBLsec20
Use index "i" concatenated with next m bits of destination address as index.
Entry #
Contents
i concatenated with next m bits
1
Index j
Third table *TBLthird20*
Use index "*j*" concatenated with last 32-*n*-*m* bits of destination address as index into this table.
Entry #
Contents
j concatenated with last 32-n-m bits
Next-hop

Figure 2.15 Three table scheme in the worst case, where the prefix is longer than (n+m) bits long. In this case, all three levels must be used, as shown.

19 bits.[1] As before, the first bit of the entry will indicate whether the rest of the entry is used as the next-hop identifier or as an index into another table (*TBLsec21* in this scheme).

If the rest of the entry in *TBLfirst21* is used as an index into another table, this 18-bit index is concatenated with the next 3 bits (bit numbers 22 through 24) of the packet's destination address, and is used as an index into *TBLsec21*. *TBLsec21* has entries of width 13 bits. As before, the first bit indicates whether the remaining 12-bits can be considered as a next-hop identifier, or as an index into the third table (*TBLthird20*). If used as an index, the 12 bits are concatenated with the last 8 bits of the packet's destination address, to index into *TBLthird20*. *TBLthird20*, like *TBLlong*, contains entries of width 8 bits, storing the next-hop identifier.

The scheme *DIR-21-3* corresponds to a (21,3,8) split of the trie. It could be generalized to the *DIR-n-m* scheme which corresponds to a $(n, m, 32 - n - m)$ split of the trie for general n and m. The three tables in *DIR-n-m* are shown in Figure 2.15.

[1] Word-lengths, such as those which are not multiples of 4, 8, or 16, are not commonly available in off-chip memories. We will ignore this issue in our examples.

DIR-21-3 has the advantage of requiring a smaller amount of memory: $(2^{21} \cdot 19) + (2^{21} \cdot 13) + (2^{20} \cdot 8) = 9MB$. One disadvantage of this scheme is an increase in the number of pipeline stages, and hence the pipeline complexity. Another disadvantage is that this scheme puts another constraint on the number of prefixes — in addition to only supporting 4096 routes of length 25 or greater with distinct 24-bit prefixes, the scheme supports only 2^{18} prefixes of length 22 or greater with distinct 21-bit prefixes. It is to be noted, however, that the decreased storage requirements enable *DIR-21-3* to be readily implemented using on-chip embedded-DRAM.[1]

The scheme can be extended to an arbitrary number of table levels between 1 and 32 at the cost of an additional constraint per table level. This is shown in Table 2.4, where we assume that at each level, only 2^{18} prefixes can be accommodated by the next higher level memory table, except the last table, which we assume supports only 4096 prefixes. Although not shown in the table, memory requirements vary significantly (for the same number of levels) with the choice of the actual number of bits to use per level. Table 2.4 shows only the *lowest* memory requirement for a given number of levels. For example, a three level (16,8,8) split would require 105 Mbytes with the same constraints. As Table

TABLE 2.4. Memory required as a function of the number of levels.

Number of levels	Bits used per level	Minimum memory requirement (Mbytes)
3	21, 3 and 8	9
4	20, 2, 2 and 8	7
5	20, 1, 1, 2 and 8	7
6	19, 1, 1, 1, 2 and 8	7

[1] IBM offers 128 Mb embedded DRAM of total size 113 mm^2 using 0.18 u semiconductor process technology [122] at the time of writing.

2.4 shows, increasing the number of levels achieves diminishing memory savings, coupled with increased hardware logic complexity to manage the deeper pipeline.

5 Routing table updates

Recall from Section 1.1 of Chapter 1 that as the topology of the network changes, new routing information is disseminated among the routers, leading to changes in routing tables. As a result, one or more entries must be added, updated, or deleted from the forwarding table. The action of modifying the table can interfere with the process of forwarding packets – hence, we need to consider the frequency and overhead caused by changes to the table. This section proposes several techniques for updating the forwarding table and evaluates them on the basis of (1) overhead to the central processor, and (2) number of memory accesses required per routing table update.

Measurements and anecdotal evidence suggest that routing tables change frequently [47]. Trace data collected from a major ISP backbone router[1] indicates that a few hundred updates can occur per second. A potential drawback of the 16-million entry *DIR-24-8-BASIC* scheme is that changing a single prefix can affect a large number of entries in the table. For instance, inserting an 8-bit prefix in an empty forwarding table may require changes to 2^{16} consecutive memory entries. With the trace data, if every routing table change affected 2^{16} entries, it would lead to millions of entry changes per second![2]

Because longer prefixes create “holes” in shorter prefixes, the memory entries required to be changed on a prefix update may not be at consecutive memory locations. This is

[1] The router is part of the Sprint network running BGP-4. The trace had a total of 3737 BGP routing updates, with an average of 1.04 updates per second and a maximum of 291 updates per second.

[2] In practice, of course, the number of 8-bit prefixes is limited to just 256, and it is extremely unlikely that they will all change at the same time.

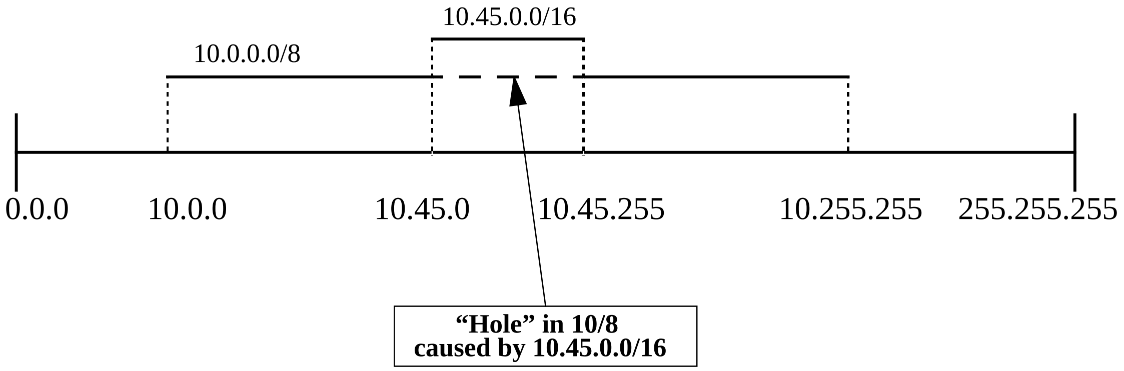


Figure 2.16 Holes created by longer prefixes require the update algorithm to be careful to avoid them while updating a shorter prefix.

illustrated in Figure 2.16 where a route-prefix of 10.45.0.0/16 exists in the forwarding table. If the new route-prefix 10.0.0.0/8 is added to the table, we need to modify only a portion of the 2^{16} entries described by the 10.0.0.0/8 route, and leave the 10.45.0.0/16 “hole” unmodified.

We will only focus on techniques to update the large *TBL24* table in the *DIR-24-8-BASIC* scheme. The smaller *TBLlong* table requires less frequent updates and is ignored in this discussion.

5.1 Dual memory banks

This technique uses two distinct ‘banks’ of memory – resulting in a simple but expensive solution. Periodically, the processor creates and downloads a new forwarding table to one bank of memory. During this time (which in general will take much longer than one lookup time), the other bank of memory is used for forwarding. Banks are switched when the new bank is ready. This provides a mechanism for the processor to update the tables in a simple and timely manner, and has been used in at least one high-performance router [76].

5.2 Single memory bank

It is possible to avoid doubling the memory by making the central processor do more work. This is typically achieved as follows: the processor keeps a software copy of the hardware memory contents and calculates the hardware memory locations that need to be modified on a prefix update. The processor then sends appropriate instructions to the hardware to change memory contents at the identified locations. An important issue to consider is the number of instructions that must flow from the processor to the hardware for every prefix update. If the number of instructions is too high, performance will become limited by the processor. We now describe three different update techniques, and compare their performance when measured by the number of update instructions that the processor must generate.

5.2.1 Update mechanism 1: *Row-update*

In this technique, the processor sends one instruction for each modified memory location. For example, if a prefix of 10/8 is added to a table that already has a prefix of 10.45.0.0/16 installed, the processor will send $65536 - 256 = 65280$ separate instructions, each instructing the hardware to change the contents of the corresponding memory locations.

While this technique is simple to implement in hardware, it places a huge burden on the processor, as experimental results described later in this section show.

5.2.2 Update mechanism 2: *Subrange-update*

The presence of “holes” partitions the range of updated entries into a series of intervals, which we call subranges. Instead of sending one instruction per memory entry, the processor can find the bounds of each subrange, and send one instruction per subrange. The instructions from the processor to the linecards are now of the form: “*change X memory entries starting at memory address Y to have the new contents Z*” where X is the number of entries in the subrange, Y is the starting entry number, and Z is the new next-hop identifier. In our example above, the updates caused by the addition of a new route-prefix in this technique are performed with just two instructions: the first instruction updating entries 10.0.0 through 10.44.255, and the second 10.46.0 through 10.255.255.

This update technique works well when entries have few “holes”. However, many instructions are still required in the worst case: it is possible (though unlikely) in the pathological case that every other entry needs to be updated. Hence, an 8-bit prefix would require up to 32,768 update instructions in the worst case.

5.2.3 Update mechanism 3: *One-instruction-update*

This technique requires only one instruction from the processor for each updated prefix, regardless of the number of holes. This is achieved by simply including an additional 5-bit length field in every memory entry indicating the length of the prefix to which the entry belongs. The hardware now uses this information to decide whether a memory entry needs to be modified on an update instruction from the processor.

Consider again the example of a routing table containing the prefixes 10.45.0.0/16 and 10.0.0.0/8. The entries in the “hole” created by the 10.45.0.0/16 prefix contain the value 16 in the 5-bit length field; the other entries associated with the 10.0.0.0/8 prefix contain the value 8. Hence, the processor only needs to send a single instruction for each prefix update. This instruction is of the form: “*insert a Y-bit long prefix starting in memory at X to have the new contents Z*”; or “*delete the Y-bit long prefix starting in memory at X* .” The hardware then examines 2^{24-Y} entries beginning with entry X. On an insertion, each entry whose length field is less than or equal to Y is updated to contain the value Z. Those entries with length field greater than Y are left unchanged. As a result, “holes” are skipped within the updated range. A delete operation proceeds similarly.

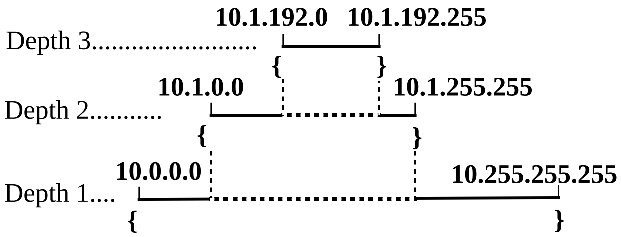


Figure 2.17 Example of the balanced parentheses property of prefixes.

This update technique reduces overhead at the cost of an additional 5-bit field that needs to be added to all 16 million entries in the table, which is an additional 10 Mbyte (about 30%) of memory. Also, unlike the Row- and Subrange-update techniques, this technique requires a read-modify-write operation for each scanned entry. This can be reduced to a parallel read and write if the marker field is stored in a separate physical memory.

5.2.4 Update mechanism 4: *Optimized One-instruction-update*

This update mechanism eliminates the need to store a length field in each memory entry, and still requires the processor to send only one instruction to the hardware. It does so by utilizing structural properties of prefixes, as explained below.

First note that for any two distinct prefixes, either one is completely contained in the other, or the two prefixes have no entries in common. This structure is very similar to that of parenthetical expressions where the scope of an expression is delimited by balanced opening and closing parentheses: for example, the characters “{” and “}” used to delimit expressions in the ‘C’ programming language. Figure 2.17 shows an example with three “nested” prefixes.

The hardware needs to know the length of the prefix that a memory entry belongs to when deciding whether or not the memory entry needs to be modified. In the previous

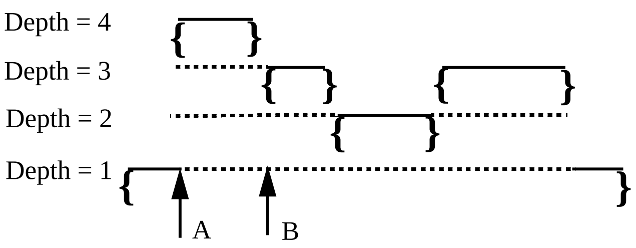


Figure 2.18 This figure shows five prefixes, one each at nesting depths 1,2 and 4; and two prefixes at depth 3. The dotted lines show those portions of ranges represented by prefixes that are also occupied by ranges of longer prefixes. Prefixes at depths 2, 3 and 4 start at the same memory entry A, and the corresponding parenthesis markers are moved appropriately.

One-instruction-update mechanism, the length is explicitly stored in each memory entry. However, the balanced parentheses property of prefixes allows the calculation of the nesting depth of a memory entry as follows. The central processor provides the hardware with the location of the *first* memory entry to be updated. Assume that this entry is at a nesting depth d. The hardware performs a sequential scan of the memory, and keeps track of the number of opening and closing parentheses seen so far in the scan. Since each opening parenthesis increases the nesting depth, and each closing parenthesis decreases the nesting depth, the hardware can calculate the nesting depth of each memory entry, and modify it if the depth is d. The sequential scan stops when the hardware encounters the closing parenthesis at nesting depth d.

Under this technique, each entry in *TBL24* is categorized as one of the following types: an opening parenthesis (start of prefix), a closing parenthesis (end of prefix), no parenthesis (middle of prefix), or both an opening and closing parenthesis (if the prefix contains only a single entry). This information is represented by a 2-bit marker field in each entry.

Care must be taken when a single entry in *TBL24* corresponds to the start or end of multiple prefixes, as shown in Figure 2.18. A 2-bit encoding is not sufficient to describe all

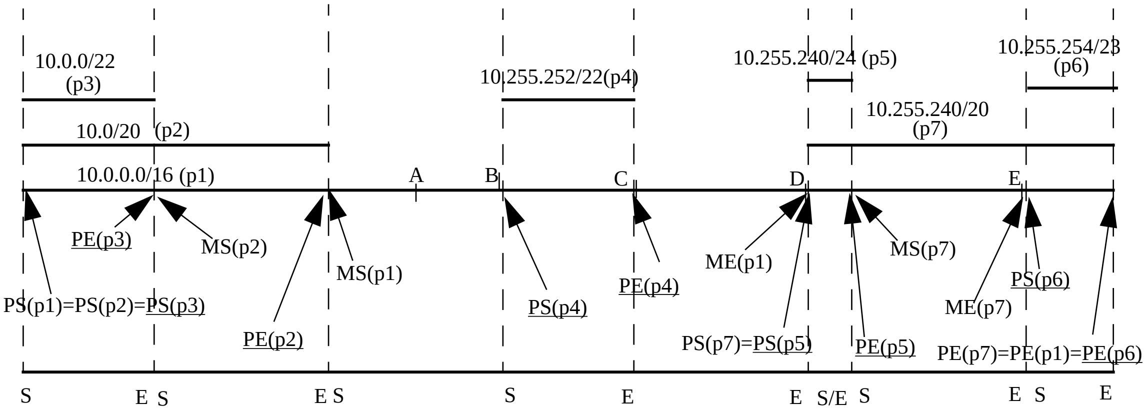


Figure 2.19 Definition of the prefix and memory start and end of prefixes. Underlined PS (PE) indicates that this prefix-start (prefix-end) is also the memory-start (memory-end) marker.

the prefixes that begin and end at a memory location ‘A.’ The problem is readily fixed by shifting the opening and closing markers to the start (end) of the first (last) entry in memory that the prefix affects. The update algorithm is described in detail below.

We first define two terms – *prefix-start (PS)* and *memory-start (MS)* of a prefix. $PS(p)$, the prefix-start of a prefix p, is defined to be the memory entry where the prefix is *supposed* to start in memory (for example, both 10.0.0.0/8 and 10.0.0.0/24 are supposed to start at 10.0.0). $MS(p)$, the memory start of a prefix p, is the first memory entry which *actually* has the entry corresponding to prefix p in memory. $MS(p)$ may or may not be the same as $PS(p)$. These two entries are different for a prefix p if and only if a longer prefix than p starts at $PS(p)$. In the same way, we define the prefix- and memory-ends (*PE* and *ME*) of a prefix. Hence, $MS(p)$ is the first memory entry which has p as the deepest (longest) prefix covering it, and $ME(p)$ is the last.

Example 2.10: If we have prefixes p1(10/8) and p2(10.0.0.0/24); $PS(p1) = PS(p2) = 10.0.0$; $MS(p1) = 10.0.1$; $MS(p2) = 10.0.0$; $ME(p1) = PE(p1) = 10.255.255.255$, $ME(p2) = PE(p2) = 10.0.0.255$.

1. Initialize a depth-counter (DC) to zero.
2. Write the start-marker on m.
3. Scan each memory entry starting with m, until either DC reaches zero, or, $PE(p)$ is reached (i.e., the memory entry just scanned has a ‘1’ in its last (24-Y) bits). At each location, perform in order: (a) If entry has start marker, increment DC by 1.(b) If DC equals 1, update this entry to denote the next-hop Z. (c) If entry has an end-marker, decrement DC by 1.
4. After completion of (3), put an end marker on the last memory entry scanned. If a total of only one memory entry (m) was scanned, put a start-and-end marker on m.

Figure 2.20 The optimized One-instruction-update algorithm executing *Update(m,Y,Z)*.

Example 2.11: Figure 2.19 shows another detailed example with several route-prefixes, along with their prefix and memory starts and ends.

Now, instead of putting the start/end markers on the prefix start/end entries, this update mechanism puts the *markers on the memory start/end entries*. Thus when the hardware encounters a marker, it can uniquely determine that exactly *one* prefix has started or ended. This takes care of the problem that multiple prefixes may start or end at the same memory location. The exact algorithm can now be formalized:

Assume that the new update is to be carried out starting with memory entry X for a Y-bit prefix, p, with new next-hop Z. First, the processor determines the first memory entry, say m, after X whose next-hop should change to Z as a result of this update. The processor then issues *one* instruction *Update(m,Y,Z)* to the hardware, which then executes the steps shown in Figure 2.20.

The algorithm can be intuitively understood as follows: if the hardware encounters a start-marker while scanning the memory in order to add a new prefix, it knows that it is entering a deeper prefix and stops updating memory until it again reaches the prefix at which it started. The end condition guarantees that any start-marker it sees will mark the start of a deeper prefix than Y (and is hence not to be updated). A formal proof of correctness of this algorithm is provided in Appendix A.

If a prefix is updated, the start and end markers may need to be changed. For instance, if the entry 10.255.240/20 (p7) is deleted in Figure 2.19, the end-marker for p1 has to be moved from point D to point E. Again this can be achieved in one instruction if the processor sends an indication of whether to move the start/end marker in conjunction with the relevant update instruction. Note that at most one start/end marker of any other prefix (apart from the one which is being updated) needs to be changed. This observation enables the algorithm to achieve all updates (additions/deletions/modifications) in only one processor instruction.

5.3 Simulation results

The behavior of each update technique was simulated with the same sequence of routing updates collected from a backbone router. The trace had a total of 3737 BGP routing updates, with an average of 1.04 updates per second and a maximum of 291 updates per second. The simulation results are shown in Table 2.5.[1]

TABLE 2.5. Simulation results of different routing table update techniques.

Update Technique	Number of instructions from processor per second (avg/max)	Number of memory accesses per second (avg/max)
Row	43.4/17545	43.4/17545
Subrange	1.14/303	43.4/17545
One-instruction	1.04/291	115.3/40415

[1] For the one-instruction-update (optimized technique) we assume that the extra 2-bits to store the opening/closing marker fields mentioned above are *not* stored in a separate memory.

The results corroborate the intuition that the row-update technique puts a large burden on the processor. At the other extreme, the one-instruction-update technique is optimal in terms of the number of instructions required to be sent by the processor. But unless a separate marker memory is used, the one-instruction technique requires more than twice as many memory accesses as the other update techniques. However, this still represents less than 0.2% of the routing lookup capacity achievable by the lookup algorithm. This simulation suggests that the subrange-update technique performs well by both measures. The small number of instructions from the processor can be attributed to the fact that the routing table contained few holes. This is to be expected for most routing tables in the near term. But it is too early to tell whether routing tables will become more fragmented and contain more holes in the future.

6 Conclusions and summary of contributions

The main contribution of this chapter is an algorithm to perform one IPv4 routing lookup operation in dedicated hardware in the time that it takes to execute a single memory access (when pipelined), and no more than two memory accesses. With the throughput of one memory access rate, approximately 20 million lookups can be completed per second with 50 ns DRAMs (or even faster with upcoming embedded-DRAM technology).

Furthermore, this is the only algorithm that we know of that supports an unlimited number of prefixes that are less than or equal to 24 bits long. Since a very small proportion (typically less than 0.1%) of all prefixes in a routing table are longer than 24 bits (see Section 3.2), this algorithm supports, practically speaking, routing tables of unlimited size. The algorithm operates by expanding the prefixes and trading-off cheap memory for speed. Yet, the total memory cost today is less than $25, and will (presumably) continue to halve each year. For those applications where low cost is paramount, this chapter described several multi-level variations on the basic scheme that utilize memory more efficiently.

Another contribution of this chapter is the design of several hardware update mechanisms. The chapter proposed and evaluated two update mechanisms (Subrange-update and One-instruction-update) that perform efficiently and quickly in hardware, with little burden on the routing processor and low interference to the normal forwarding function.

CHAPTER 3

Minimum average and bounded worst-case routing lookup time on binary search trees

1 Introduction

Most work on routing lookups [17][31][69][93] has focused on the development of data structures and algorithms for minimizing the worst-case lookup time, given a forwarding table and some storage space constraints. Minimizing the worst-case lookup time is attractive because it does not require packets to be queued before lookup. This enables simplicity and helps bound the delay of the packet through the router. However, it suffices to minimize the average lookup time for some types of traffic, such as “best-effort” traffic.[1] This presents opportunities for higher overall lookup performance because an average case constraint is less stringent than the worst-case constraint. This chapter presents two such algorithms for minimizing the average lookup time — in particular, lookup algorithms that adapt their binary search tree data structure based on the observed statistical

[1] Best-effort traffic comprises the highest proportion of Internet traffic today. This is generally expected to continue to remain true in the near future.

properties of recent lookup results in order to achieve higher performance. The exact amount of performance improvement obtained using the proposed algorithms depends on the forwarding table and the traffic patterns. For example, experiments using one set of parameters show a reduction of 42% in the average number of memory accesses per lookup than those obtained by worst-case lookup time minimization algorithms. Another benefit of these algorithms is the “near-perfect” load balancing property of the resulting tree data structures. This enables, for example, doubling the lookup speed by replicating only the root node of the tree, and assigning one lookup engine each to the left and right subtrees.

As we saw in Chapter 2, most lookup algorithms use a tree-based data structure. A natural question to ask is: “What is the best tree data structure for a given forwarding table?”. This chapter considers this question in the context of binary search trees as constructed by the lookup algorithm discussed in Section 2.2.6 of Chapter 2. The two algorithms proposed in this chapter adapt the shape of the binary search tree constructed by the lookup algorithm of Section 2.2.6 of Chapter 2. The tree is redrawn based on the statistics gathered on the number of accesses to prefixes in the forwarding table, with the aim of minimizing the average lookup time. However, the use of a binary search tree data structure brings up a problem — depending on the distribution of prefix access probabilities, it is possible for the worst-case depth of a redrawn binary search tree to be as large as $2m - 1$, where m is the total number of forwarding table entries, and is close to 98,000 [136] at the time of writing. The worst-case lookup time can not be completely neglected — if it takes very long to lookup even one incoming packet, a large number of packets arriving shortly thereafter must be queued until the packet has completed its lookup. Practical router design considerations (such as silicon and board real-estate resources) limit the maximum size of this queue, and hence make bounding the worst-case lookup time highly desirable. Bounding the worst-case performance also enables bounding packet delay in the router and hence in the network. It is the objective of this chapter to devise algorithms on binary search trees that *minimize the average* lookup time while *keeping the worst-case* lookup time *smaller* than a pre-specified maximum.

The approach taken in this chapter has a limitation that it cannot be used in some hardware-based designs where the designer desires a fixed routing lookup time for all packets. The approach of this chapter can only be used when the router designer wants to minimize the average, subject to a maximum lookup time. Thus, the designer should be willing to buffer incoming packets before sending them to the lookup engine in order to absorb the variability in the lookup times of different packets.

1.1 Organization of the chapter

Section 2 sets up the formal minimization problem. Sections 3 and 4 describe the two proposed algorithms and analyze their performance. Section 5 discusses the load balancing characteristics of these algorithms, and Section 6 provides experimental results on publicly available routing tables and a packet trace. Section 7 discusses related work, and Section 8 concludes with a summary and contributions of this chapter.

2 Problem statement

Recall that the binary search algorithm [49], discussed in Section 2.2.6 of Chapter 2, views each prefix as an interval on the IP number line. The union of the end points of these intervals partitions the number line into a set of disjoint intervals, called basic intervals (see, for example, Figure 2.7 of Chapter 2). The algorithm precomputes the longest prefix for every basic interval in the partition, and associates every basic interval with its left end-point. The distinct number of end-points for m prefixes is at most $n = 2m$. These end-points are kept in a sorted list. Given a point, P, on the number line representing an incoming packet, the longest prefix matching problem is solved by using binary search on

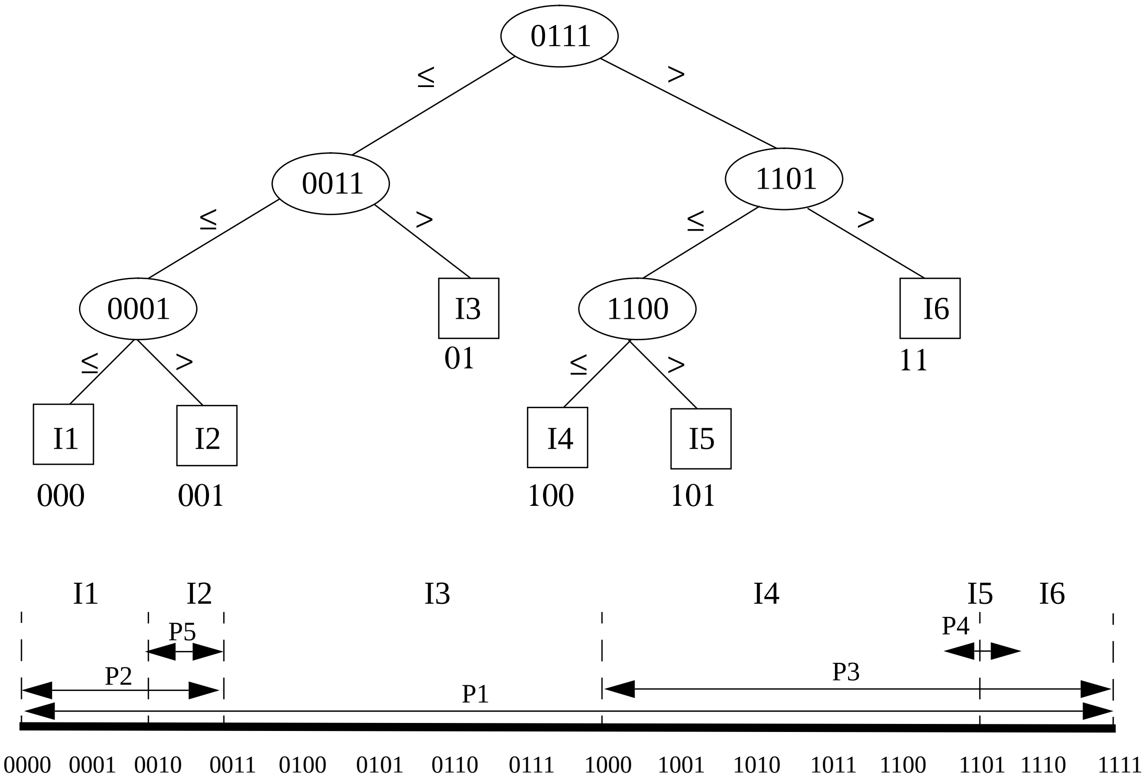


Figure 3.1 The binary search tree corresponding to the forwarding table in Table 3.1. The bit-strings in bold are the binary codes of the leaves.

the sorted list to find the end-point in the list that is closest to, but not greater than P. Binary search is performed by the following binary tree data structure: the leaves (external nodes) of the tree store the left end-points in order from left to right, and the internal nodes of the tree contain suitably chosen values to guide the search process to the appropriate child node. This binary search tree for m prefixes takes $O(m)$ storage space and has a maximum depth of $O(\log(2m))$.[1]

Example 3.1: An example of a forwarding table with 4-bit prefixes is shown in Table 3.1, and the corresponding partition of the IP number line and the binary search tree is shown in Figure 3.1.

[1] All logarithms in this chapter are to the base 2.

TABLE 3.1. An example forwarding table.

	Prefix	**Interval start-point**	**Interval end-point**
P1	*	0000	1111
P2	00*	0000	0011
P3	1*	1000	1111
P4	1101	1101	1101
P5	001*	0010	0011

The key idea used in this chapter is that the average lookup time in the binary search tree data structure can be decreased by making use of the frequency with which a certain forwarding table entry is accessed in the router. We note that most routers already maintain such per-entry statistics. Hence, minimizing routing lookup times by making use of this information comes at no extra data collection cost. A natural question to ask is: 'Given the frequency with which the leaves of a tree are accessed, what is the best binary search tree — i.e., the tree with the minimum average depth?' Viewing it this way, the problem is readily recognized to be one of minimizing the average weighted depth of a binary tree whose leaves are weighted by the probabilities associated with the basic intervals represented by the leaves. The minimization is to be carried over all possible binary trees that can be constructed with the given number and weights of the leaves.

This problem is analogous to the design of efficient codes (see Chapter 5 of Cover and Thomas [14]), and so we briefly explain here the relationship between the two problems. A binary search tree is referred to as an *alphabetic tree*, and the leaves of the tree the *letters* of that alphabet. Each leaf is assigned a binary codeword depending on its position in the tree. The length of the codeword of a symbol is equal to the depth of the corresponding leaf in the tree. For the example in Figure 3.1, the codeword associated with interval *I1* is

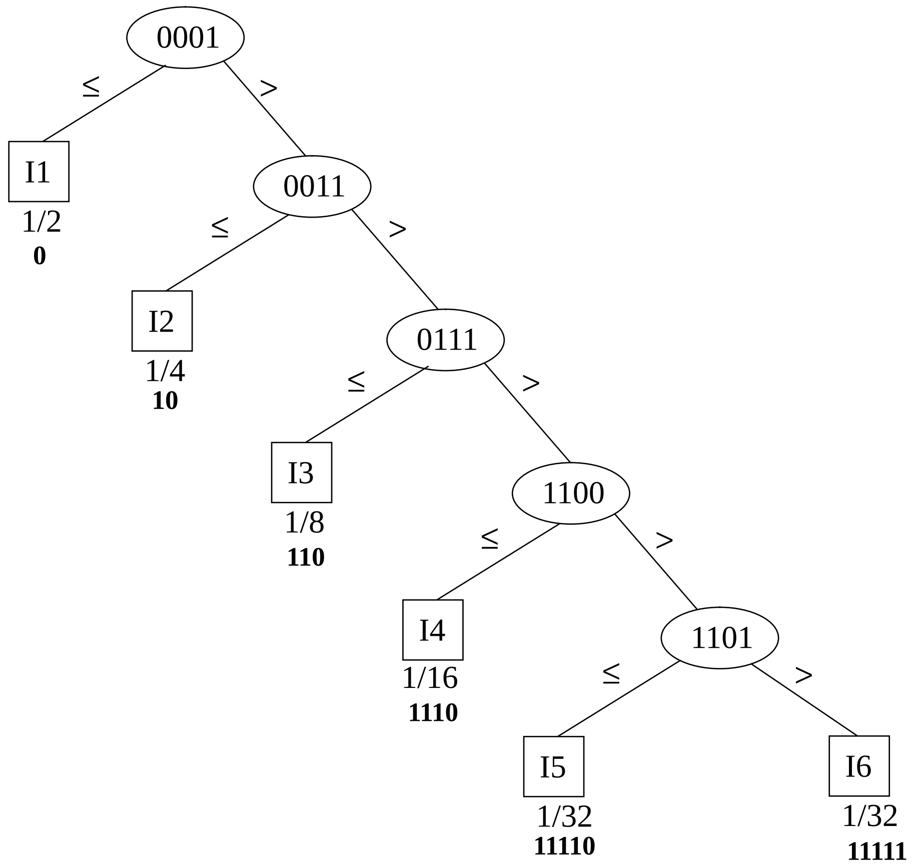


Figure 3.2 The optimal binary search tree (i.e., one with the minimum average weighted depth) corresponding to the tree in Figure 3.1 when leaf probabilities are as shown. The binary codewords are shown in bold.

000 and that associated with interval *I5* is 101, where a bit in the codeword is 0 (respectively 1) for the left (respectively right) branch at the corresponding node.

A *prefix* code satisfies the property that no two codes are prefixes of each other. An *alphabetic* code is a prefix code in which the n letters are ordered lexicographically on the leaves of the resulting binary tree. In other words, if letter A appears before letter B in the alphabet, then the codeword associated with letter A has a value of smaller magnitude than the codeword associated with letter B. Designing a code for an alphabet is equivalent to constructing a tree for the letters of the alphabet. With a letter corresponding to an interval, the lookup problem translates to: “Find a minimum average length alphabetic prefix code (or tree) for an n -letter alphabet.”

Example 3.2: If the intervals *I1* through *I6* in Figure 3.1 are accessed with probabilities 1/2, 1/4, 1/8, 1/16, 1/32 and 1/32 respectively, then the best (i.e., optimal) alphabetic tree corresponding to these probabilities (or weights) is shown in Figure 3.2. The codeword for *I1* is now 0 and that of *I5* is 11110. Since *I1* is accessed with a greater probability than *I5*, it has been placed higher up in the tree, and thus has a shorter codeword

The average length of a general prefix code for a given set of probabilities can be minimized using the Huffman coding algorithm [39]. However, Huffman’s algorithm does not necessarily maintain the alphabetic order of the input data set. This causes implementational problems, as simple comparison queries are not possible at internal nodes to guide the binary search algorithm. Instead, at an internal node of a Huffman tree, one needs to ask for memberships in arbitrary subsets of the alphabet to proceed to the next level. Because this is as hard as the original search problem, it is not feasible to use Huffman’s algorithm.

As mentioned previously, we wish to bound the maximum codeword length (i.e., the maximum depth of the tree) to make the solution useful in practice. This can now be better understood: an optimal alphabetic tree for n letters can have a maximum depth (the root is assumed to be at depth 0) of $n-1$ (see, for instance, Figure 3.2 with $n = 6$). This is unacceptable in practice because we have seen that $n = 2m$, and the value of m, the size of the forwarding table, could be as high as 98,000 [136]. Furthermore, any change in the network topology or in the distribution of incoming packet addresses can lead to a large increase in the access frequency of a deep leaf. It is therefore highly desirable to have a small upper bound on the maximum depth of the alphabetic tree. Therefore, well-known algorithms for finding an optimal alphabetic tree such as those in [27][36][37] which do

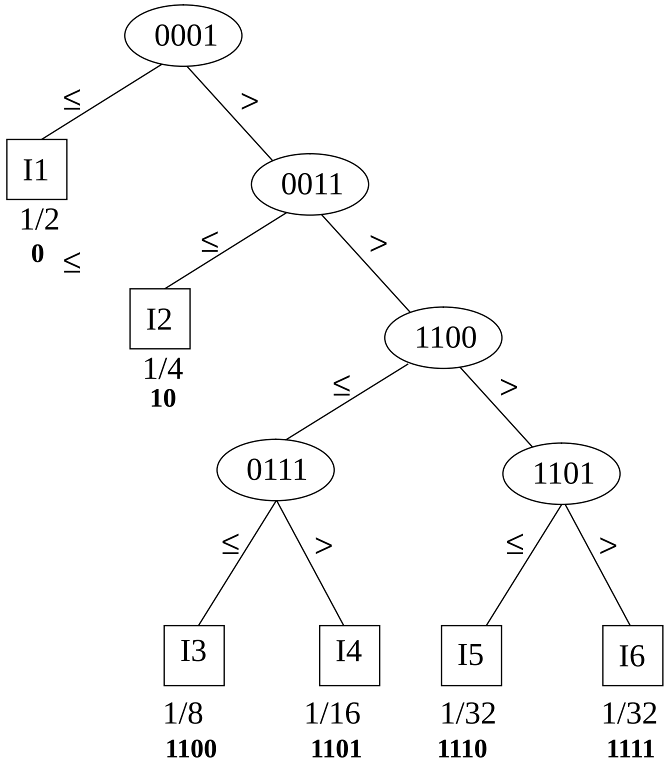


Figure 3.3 The optimal binary search tree with a depth-constraint of 4, corresponding to the tree in Figure 3.1.

not incorporate a maximum depth-constraint cannot be used in this chapter's setting. Here is an example to understand this last point better.

Example 3.3: The alphabetic tree in Figure 3.2 is optimal if the intervals I1 through I6 shown in the binary tree of Figure 3.1 are accessed with probabilities {1/2, 1/4, 1/8, 1/16, 1/32, 1/32} respectively. For these probabilities, the average lookup time is 1.9375,[1] while the maximum depth is 5. If we impose a maximum depth-constraint of 4, then we need to redraw the tree to obtain the optimal tree that has minimum average weighted depth and has maximum depth no greater than 4. This tree is shown in Figure 3.3 where the average lookup time is calculated to be 2.

The general minimization problem can now be stated as follows:

[1] $1.9375 = 1 \cdot (1/2) + 2 \cdot (1/4) + 3 \cdot (1/8) + 4 \cdot (1/16) + 5 \cdot (1/32) + 5 \cdot (1/32)$

Choose $\{l_i\}_{i=1}^{n}$ in order to minimize $C = \sum_{i=1}^{n} l_i \cdot p_i$, such that $l_i \le D \ \forall i$, and $\{l_i\}_{i=1}^{n}$ gives rise to an alphabetic tree for n intervals where p_i is the access probability of the i^{th} interval, and l_i is the length of its codeword, i.e., the number of comparisons required to lookup a packet in the i^{th} interval.

The smallest possible value of C is the entropy [14], $H(p)$, of the set of probabilities $\{p_i\}$, where $H(p) = -\sum_i p_i \log p_i$. It is usually the case that C is larger than $H(p)$ for depth-constrained alphabetic trees.[1] Finding fast algorithms for computing optimal depth-constrained binary trees (without the alphabetic constraint) is known to be a hard problem, and good approximate solutions are appearing only now [59][60][61], almost 40 years after the original Huffman algorithm [39]. Imposing the alphabetic constraint renders the problem harder [27][28][35][109]. Still, an optimal algorithm, proposed by Larmore and Przytycka [50], finds the best depth-constrained alphabetic tree in $O(nD \log n)$ time. Despite its optimality, the algorithm is complicated and difficult to implement.[2]

In light of this, our goal is to find a practical and provably good approximate solution to the problem of computing optimal depth-constrained alphabetic trees. Such a solution should be simpler to find than an optimal solution. More importantly, it should be much simpler to implement. Also, as the probabilities associated with the intervals induced by routing prefixes change and are not known exactly, it does not seem to make much sense to solve the problem exactly for an optimal solution. As we will see later, one of the two near-optimal algorithms proposed in this chapter can be analytically proved to be requiring no more than two extra comparisons per lookup when compared to the optimal solution. In practice, this discrepancy has been found to be less than two (for both of the approximate algorithms). Hence, we refer to them as algorithms for *near-optimal depth-constrained alphabetic trees*, and describe them next.

[1] The lower bound of entropy is achieved in general when there are no alphabetic or maximum depth-constraints.

[2] The complexity formula $O(nD \log n)$ has large constant factors, as the implementation requires using a list of mergeable priority queues with priority queue operations such as *delete_min, merge, find* etc.

3 Algorithm MINDPQ

We first state two results from Yeung [114] as lemmas that we will use to develop algorithm MINDPQ. The first lemma states a necessary and sufficient condition for the existence of an alphabetic code with specified codeword lengths, and the second prescribes a method for constructing good, near-optimal trees (which are not depth-constrained).

Lemma 3.1 *(The Characteristic Inequality)*: There exists an alphabetic code with codeword lengths l_k if and only if $s_n \le 1$, where $s_k = c(s_{k-1}, 2^{-l_k}) + 2^{-l_k}$, $s_0 = 0$, and c is defined by $c(a,b) = \lceil a/b \rceil b$.

Proof: For a complete proof, see [114]. The basic idea is to construct a *canonical* coding tree, a tree in which the codewords are chosen lexicographically using the lengths l_i. For instance, suppose that $l_i = 4$ for some i, and in drawing the canonical tree we find the codeword corresponding to letter i to be 0010. If $l_{i+1} = 4$, then the codeword for letter $i+1$ will be chosen to be 0011; if $l_{i+1} = 3$, the codeword for letter $i+1$ is chosen to be 010; and if $l_{i+1} = 5$, the codeword for letter $i+1$ is chosen to be 00110. Clearly, the resulting tree will be alphabetic and Yeung's result verifies that this is possible if and only if the characteristic inequality defined above is satisfied by the lengths l_i.

The next lemma (also from [114]) considers the construction of good, near-optimal codes. Note that it does not produce alphabetic trees with prescribed maximum depths. That is the subject of this chapter.

Lemma 3.2 The minimum average length, C_{min}, of an alphabetic code on n letters, where the i^{th} letter occurs with probability p_i satisfies: $H(p) \le C_{min} \le H(p) + 2 - p_1 - p_{min}$. Therefore, there exists an alphabetic tree on n letters with average code length within 2 bits of the entropy of the probability distribution of the letters.

Proof: The lower bound, $H(p)$, is obvious. For the upper bound, the code length l_k of the k^{th} letter occurring with probability p_k is chosen to be:

$$l_k = \begin{cases} \lceil -\log p_k \rceil & k = 1,n \\ \lceil -\log p_k \rceil + 1 & 2 \le k \le n-1 \end{cases}$$

The proof in [114] verifies that these lengths satisfy the characteristic inequality of Lemma 3.1, and shows that a canonical coding tree constructed with these lengths has an average depth satisfying the upper bound.

We now return to our original problem of finding near-optimal depth-constrained alphabetic trees. Let D be the maximum allowed depth. Since the given set of probabilities $\{p_k\}$ might be such that $p_{min} = min_k\{p_k\} < 2^{-D}$, a direct application of Lemma 3.2 could yield a tree where the maximum depth is higher than D. To work around this problem, we transform the given probabilities p_k into another set of probabilities q_k such that $q_{min} = min_k\{q_k\} \ge 2^{-D}$. This allows us to apply the following variant of the scheme in Lemma 3.2 to obtain a near-optimal depth-constrained alphabetic tree with leaf probabilities q_k.

Given a probability vector q_k such that $q_{min} \ge 2^{-D}$, we construct a canonical alphabetic coding tree with the codeword length assignment to the k^{th} letter given by:

$$l_k^* = \begin{cases} min(\lceil -\log q_k \rceil, D) & k = 1, n \\ min(\lceil -\log q_k \rceil + 1, D) & 2 \le k \le n-1 \end{cases} \tag{3.1}$$

Each codeword is clearly at most D bits long and the tree thus generated has a maximum depth of D. It remains to be shown that these codeword lengths yield an alphabetic tree. By Lemma 3.1 it suffices to show that the $\{l_k^*\}$ satisfy the characteristic inequality. This verification is deferred to Appendix B later in the thesis.

Proceeding, if the codeword lengths are given by $\{l_k^*\}$, the resulting alphabetic tree has an average length of $\sum_k p_k l_k^*$. Now,

$$\sum_k p_k l_k^* \le \sum_k p_k \log \frac{1}{q_k} + 2 = \sum_k p_k \log \frac{p_k}{q_k} - \sum_k p_k \log p_k + 2 = D(p \parallel q) + H(p) + 2 \qquad (3.2)$$

where $D(p \parallel q)$ is the 'relative entropy' (see page 22 of [14]) between the probability distributions p and q, and $H(p)$ is the entropy of the probability distribution p. In order to minimize $\sum_k p_k l_k^*$, we must therefore choose $\{q_i\}$, given $\{p_i\}$, so as to minimize $D(p \parallel q)$.

3.1 The minimization problem

We are thus led to the following optimization problem:

Given $\{p_i\}$, choose $\{q_i\}$ in order to minimize $DPQ = D(p \parallel q) = \sum_i p_i \log (p_i / q_i)$ subject to $\sum_i q_i = 1, q_i \ge Q = 2^{-D} \forall i$.

Observe that the cost function $D(p \parallel q)$ is convex in (p, q) (see page 30 of [14]). Further, the constraint set is convex and compact. In fact, we note that the constraint set is defined by *linear* inequalities. Minimizing convex cost functions with linear constraints is a standard problem in optimization theory and is easily solved by using Lagrange multiplier methods (see, for example, Section 3.4 of Bertsekas [5]).

Accordingly, define the Lagrangean

$$L(q, \bar{\lambda}, \mu) = \sum p_i \log (p_i / q_i) + \sum \lambda_i (Q - q_i) + \mu\left(\sum q_i - 1\right)$$

Setting the partial derivatives with respect to q_i to zero at q_i^*, we get:

$$\left(\frac{\partial L}{\partial q_i} = 0\right) \Rightarrow q_i^* = \frac{p_i}{\mu - \lambda_i} \qquad (3.3)$$

Putting this back in $L(q, \bar{\lambda}, \mu)$, we get the dual: $G(\bar{\lambda}, \mu) = \sum_i (p_i \log(\mu - \lambda_i) + \lambda_i Q) + (1 - \mu)$. Now minimizing $G(\bar{\lambda}, \mu)$ subject to $\lambda_i \geq 0$ and $\mu > \lambda_i \forall i$ gives:

$$\frac{\partial G}{\partial \mu} = 0 \Rightarrow \sum_i \frac{p_i}{\mu - \lambda_i} = 1$$

$$\frac{\partial G}{\partial \lambda_i} = 0 \forall i \Rightarrow Q = \frac{p_i}{\mu - \lambda_i},$$

which combined with the constraint that $\lambda_i \geq 0$ gives us $\lambda_i^* = max(0, \mu - p_i / Q)$. Substituting this in Equation 3.3, we get

$$q_i^* = max(p_i / \mu, Q) \tag{3.4}$$

To finish, we need to solve Equation 3.4 for $\mu = \mu^*$ under the constraint that $\sum_{i=1}^{n} q_i^* = 1$. The desired probability distribution is then $\{q_i^*\}$. It turns out that we can find an explicit solution for μ^*, using which we can solve Equation 3.4 by an algorithm that takes $O(n \log n)$ time and $O(n)$ storage space. This algorithm first sorts the original probabilities $\{p_i\}$ to get $\{\hat{p}_i\}$ such that $\{\hat{p}_1\}$ is the largest and $\{\hat{p}_n\}$ the smallest probability. Call the transformed (sorted) probability distribution $\{\hat{q}_i^*\}$. Then the algorithm solves for μ^* such that $F(\mu^*) = 0$ where:

$$F(\mu) = \sum_{i=1}^{n} \hat{q}_i^* - 1 = \sum_{i=1}^{k_\mu} \frac{\hat{p}_i}{\mu} + (n - k_\mu) Q - 1 \tag{3.5}$$

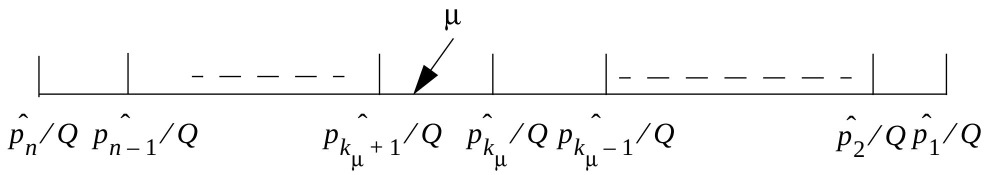


Figure 3.4 Showing the position of μ and k_μ.

Here, k_μ is the number of letters with probability greater than $(\mu \cdot Q)$, and the second equality follows from Equation 3.4. Figure 3.4 shows the relationship between μ and k_μ. For all letters to the left of μ in Figure 3.4, $q_i^* = Q$ and for others, $q_i^* = \hat{p}_i/\mu$.

Lemma 3.3 $F(\mu)$ is a monotonically decreasing function of μ.

Proof: First, it is easy to see that if μ increases in the interval $\left[\hat{p}_{r+1}/Q, \hat{p}_r/Q\right)$, i.e., such that k_μ does not change, $F(\mu)$ decreases monotonically. Similarly, if μ increases from $\hat{p}_r/Q - \varepsilon$ to $\hat{p}_r/Q + \varepsilon$ so that k_μ decreases by 1, it is easy to verify that $F(\mu)$ decreases.

The algorithm uses Lemma 3.3 to do a binary search (in $O(\log n)$ time) for finding the half-closed interval that contains μ, i.e., a suitable value of r such that $\mu \in \left[\hat{p}_r/Q, \hat{p}_{r-1}/Q\right)$ and $F(\hat{p}_r/Q) \geq 0$ and $F(\hat{p}_{r-1}/Q) < 0$.[1] The algorithm then knows the exact value of $k_\mu = K$ and can directly solve for μ^* using Equation 3.5 to get an explicit formula to calculate $\mu^* = \left(\sum_{i=1}^{K} \hat{p}_i\right) / (1 - (n-K)Q)$. Putting this value of μ^* in Equation 3.4 then gives the transformed set of probabilities $\{\hat{q}_i^*\}$. Given such $\{\hat{q}_i\}$, the algorithm then constructs a canonical alphabetic coding tree as in [114] with the codeword lengths l_k^* as chosen in Equation 3.1. This tree clearly has a maximum depth of no more than D, and its average weighted depth is worse than the optimal algorithm by no more

[1] Note that $O(n)$ time is spent by the algorithm in the calculation of $\sum_{i=1}^{k_\mu} \hat{p}_i$ anyway, so a simple linear search can be implemented to find the interval $\left[\hat{p}_r/Q, \hat{p}_{r-1}/Q\right)$.

than 2 bits. To see this, let us refer to the codeword lengths in the optimal tree as $\{l_k^{opt}\}$. Then $C_{opt} = \sum_k p_k l_k = H(p) + D(p \| 2^{-l_k^{opt}})$. As q^* has been chosen to be such that $D(p \| q^*) \le D(p \| q)$ for all probability distributions q in the set $\{q_i : q_i \ge Q\}$, it follows from Equation 3.2 that $C_{mindpq} \le H(p) + D(p \| q^*) + 2 \le C_{opt} + 2$. This proves the following main theorem of this chapter:

Theorem 3.1 Given a set of n probabilities $\{p_i\}$ in a specified order, an alphabetic tree with a depth-constraint D can be constructed in $O(n \log n)$ time and $O(n)$ space such that the average codeword length is at most 2 bits more than that of the optimal depth-constrained alphabetic tree. Further, if the probabilities are given in sorted order, such a tree can be constructed in linear time.

4 Depth-constrained weight balanced tree (DCWBT)

This section presents a heuristic algorithm to generate near-optimal depth-constrained alphabetic trees. This heuristic is similar to the weight balancing heuristic proposed by Horibe [35] with the modification that the maximum depth-constraint is never violated. The trees generated by this heuristic algorithm have been observed to have even lower average weighted depth than those generated by algorithm MINDPQ. Also, the implementation of this algorithm turns out to be even simpler. Despite its simplicity, it is unfortunately hard to prove optimality properties of this algorithm.

We proceed to describe the normal weight balancing heuristic of Horibe, and then describe the modification needed to incorporate the constraint of maximum depth. First, we need some terminology. In a tree, suppose the leaves of a particular subtree correspond to letters numbered r through t — we say that the weight of the subtree is $\sum_{i=r}^{t} p_i$. The root node of this subtree is said to represent the probabilities $\{p_r, p_{r+1}, ..., p_t\}$; denoted by $\{p_i\}_{i=r}^{t}$. Thus, the root node of an alphabetic tree has weight 1 and represents the probability distribution $\{p_i\}_{i=1}^{n}$.

In the normal weight balancing heuristic of Horibe [35], one constructs a tree such that the weight of the root node is split into two parts representing the weights of its two children in the most balanced manner possible. The weights of the two children nodes are then split recursively in a similar manner. In general, at an internal node representing the probabilities $\{p_r \dots p_t\}$, the left and right children are taken as representing the probabilities $\{p_r \dots p_s\}$ and $\{p_{s+1} \dots p_t\}$, $r \le s < t$, if s is such that

$$\Delta(r, t) = \left| \sum_{i=r}^{s} p_i - \sum_{i=s+1}^{t} p_i \right| = min_{\forall u\,(r \le u < t)} \left| \sum_{i=r}^{u} p_i - \sum_{i=u+1}^{t} p_i \right|$$

This ‘top-down’ algorithm clearly produces an alphabetic tree. As an example, the weight-balanced tree corresponding to Figure 3.1 is the tree shown in Figure 3.2. Horibe [35] proves that the average depth of such a weight-balanced tree is greater than the entropy of the underlying probability distribution $\{p_i\}$ by no more than $2 - (n+2)p_{min}$, where p_{min} is the minimum probability in the distribution.

Again this simple weight balancing heuristic can produce a tree of unbounded maximum depth. For instance, a distribution $\{p_i\}$ such that $p_n = 2^{-(n-1)}$ and $p_i = 2^{-i}$ $\forall 1 \le i \le n-1$, will produce a highly skewed tree of maximum depth $n-1$. Figure 3.2 is an instance of a highly skewed tree on such a distribution. We now propose a simple modification to account for the depth constraint. The modified algorithm follows Horibe's weight balancing heuristic, constructing the tree in the normal top-down weight balancing manner until it reaches a node such that if the algorithm were to split the weight of the node further in the most balanced manner, the depth-constraint would be violated. Instead, the algorithm splits the node maintaining as much balance as it can while respecting the depth-constraint. In other words, if this node is at depth d representing the probabilities $\{p_r \ldots p_t\}$, the algorithm takes the left and right children as representing the probabilities $\{p_r \ldots p_s\}$ and $\{p_{s+1} \ldots p_t\}$, $a \le s < b$, if s is such that

$$\Delta(r, t) = \left|\sum_{i=r}^{s} p_i - \sum_{i=s+1}^{t} p_i\right| = min_{\forall u\,(a \le u < b)} \left|\sum_{i=r}^{u} p_i - \sum_{i=u+1}^{t} p_i\right|,$$

and $a = t - 2^{D-d-1}$ and $b = r + 2^{D-d-1}$. Therefore, the idea is to use the weight balancing heuristic as far down into the tree as possible. This implies that any node where the modified algorithm is unable to use the original heuristic would be deep down in the tree. Hence, the total weight of this node would be small enough so that approximating the weight balancing heuristic does not cause any substantial effect to the average path length. For instance, Figure 3.5 shows the depth-constrained weight balanced tree for a maximum depth-constraint of 4 for the tree in Figure 3.1.

As mentioned above, we have been unable to come up with a provably good bound on the distance of this heuristic from the optimal solution, but its conceptual and implementational simplicity along with the experimental results (see next section) suggest its usefulness.

Lemma 3.4 A depth-constrained weight balanced tree (DCWBT) for n leaves can be constructed in $O(n\log n)$ time and $O(n)$ space.

Proof: At an internal node, the signed difference in the weights between its two subtrees is a monotonically increasing function of the difference in the number of nodes in the left and right subtrees. Thus a suitable split may be found by binary search in $O(\log n)$ time at every internal node.[1] Since there are $n-1$ internal nodes in a binary tree with n leaves, the total time complexity is $O(n\log n)$. The space complexity is the complexity of storing the binary tree and is thus linear.

[1] Note that we may need access to $\sum_{i=r}^{s} p_i, \forall 1 \le r, s \le n$. This can be obtained by precomputing $\sum_{i=1}^{s} p_i, \forall 1 \le s \le n$ in linear time and space.

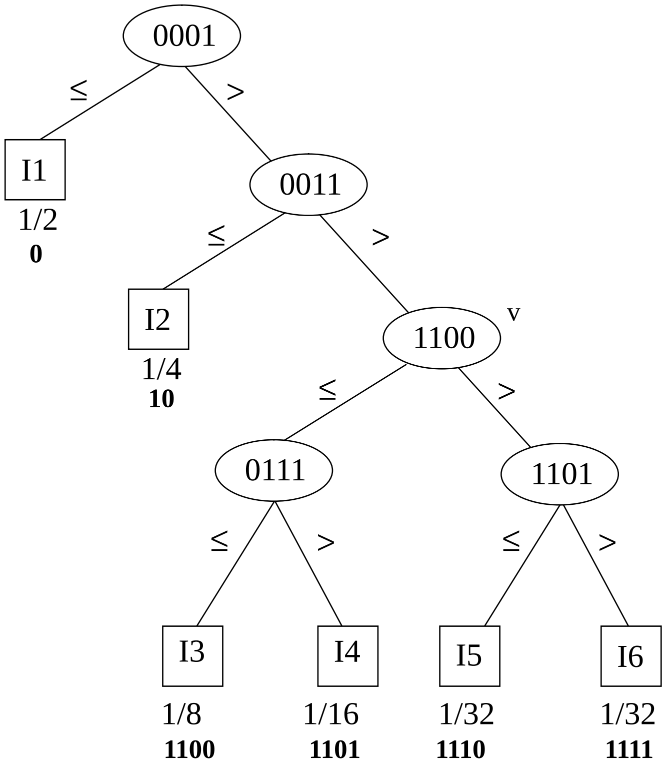


Figure 3.5 Weight balanced tree for Figure 3.1 with a depth-constraint of 4. The DCWBT heuristic is applied in this example at node v (labeled 1100).

5 Load balancing

Both of the algorithms MINDPQ and DCWBT produce a binary search tree that is fairly weight-balanced. This implies that such a tree data structure can be efficiently parallelized. For instance, if two separate lookup engines for traversing a binary tree were available, one engine can be assigned to the left-subtree of the root node and the second to the right-subtree. Since the work load is expected to be balanced among the two engines, we can get twice the average lookup rate that is possible with one engine. This 'near-perfect load-balancing' helps achieve speedup linear in the number of lookup engines, a feature attractive in parallelizable designs. The scalability property can be extended — for instance, the average lookup rate could be made 8 times higher by having 8 subtrees, each being traversed by a separate lookup engine running at 1/8th the aggregate lookup rate

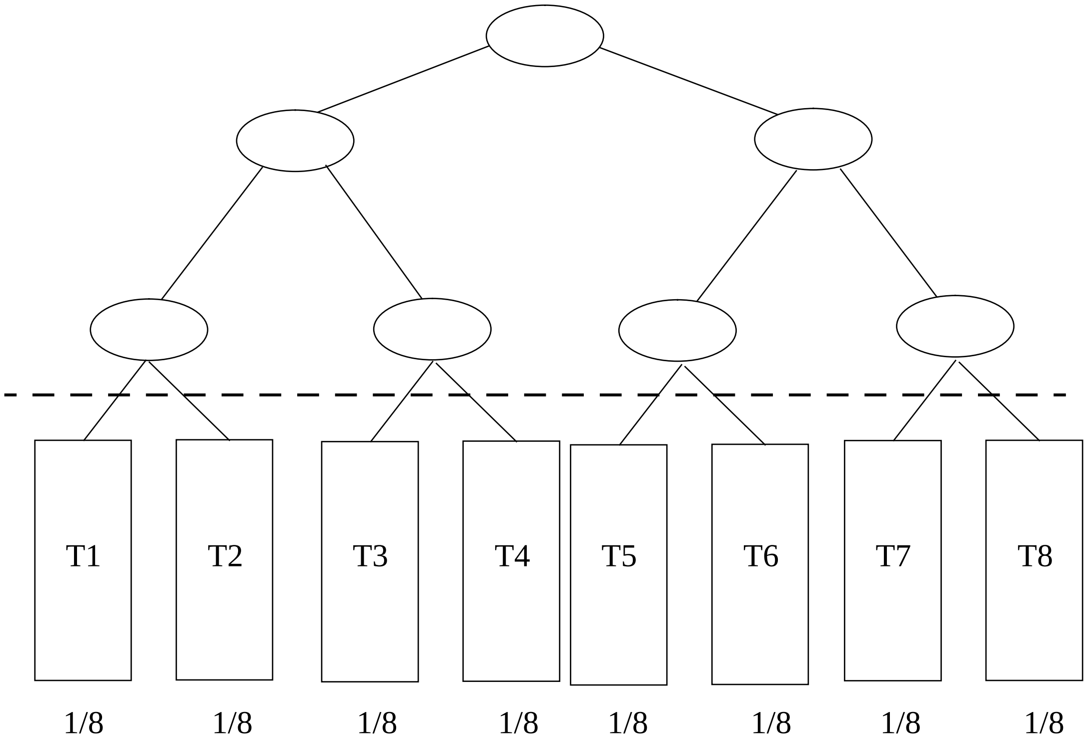


Figure 3.6 Showing 8-way parallelism achievable in an alphabetic tree constructed using algorithm MINDPQ or DCWBT.

(see Figure 3.6). It is to be remembered, however, that only the *average* lookup rate is balanced among the different engines, and hence, a buffer is required to absorb short-term bursts to one particular engine in such a parallel architecture.

6 Experimental results

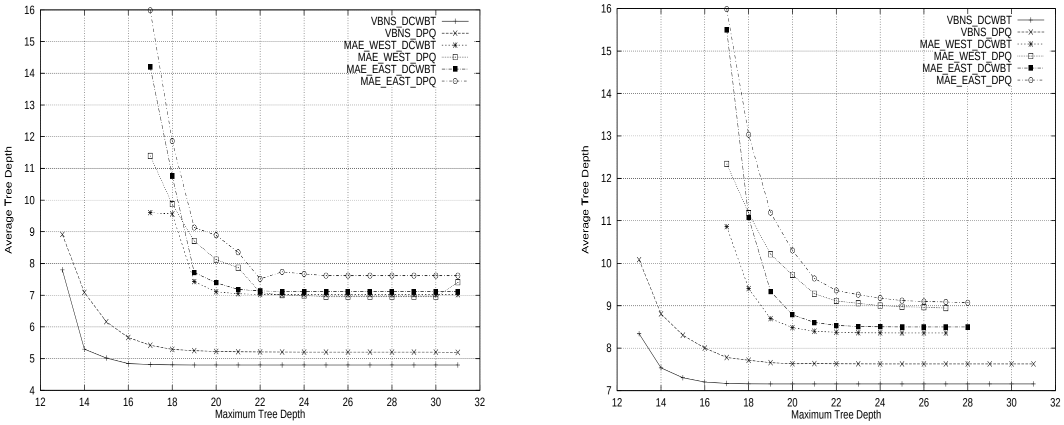
A plot at CAIDA [12] shows that over 80% of the traffic is destined to less than 10% of the autonomous systems — hence, the amount of traffic is very non-uniformly distributed over prefixes. This provides some real-life evidence of the possible benefits to be gained by optimizing the routing table lookup data structure based on the access frequency of the table entries. To demonstrate this claim, we performed experiments using two large default-free routing tables that are publicly available at IPMA [124], and another smaller table available at VBNS [118].

A knowledge of the access probabilities of the routing table entries is crucial to make an accurate evaluation of the advantages of the optimization algorithms proposed in this chapter. However, there are no publicly available packet traffic traces with non-encrypted destination addresses that access these tables. Fortunately, we were able to find one trace of about 2.14 million packet destination addresses at NLANR [134]. This trace has been taken from a different network location (*Fix-West)* and thus does not access the same routing tables as obtained from IPMA. Still, as the default-free routing tables should not be too different from each other, the use of this trace should give us valuable insights into the advantages of the proposed algorithms. In addition, we also consider the 'uniform' distribution in our experiments, where the probability of accessing a particular prefix is proportional to the size of its interval, i.e., an 8-bit long prefix has a probability of access twice that of a 9-bit long prefix.

Table 3.2 shows the sizes of the three routing tables considered in our experiments, along with the entropy values of the uniform probability distribution and the probability distribution obtained from the trace. Also shown is the number of memory accesses required in an unoptimized binary search (denoted as "Unopt_srch"), which simply is $\lceil \log (\#\text{Intervals}) \rceil$.

TABLE 3.2. Routing tables considered in experiments. Unopt_srch is the number of memory accesses required in a naive, unoptimized binary search tree.

Routing table	**Number of prefixes**	**Number of intervals**	**Entropy (uniform)**	**Entropy (trace)**	**Unopt_s rch**
VBNS [118]	1307	2243	4.41	6.63	12
MAE_WEST [124]	24681	39277	6.61	7.89	16
MAE_EAST [124]	43435	65330	6.89	8.02	16



(a) (b)

Figure 3.7 Showing how the average lookup time decreases when the worst-case depth-constraint is relaxed: (a) for the "uniform" probability distribution, (b) for the probability distribution derived by the 2.14 million packet trace available from NLANR. X_Y in the legend means that the plot relates to algorithm Y when applied to routing table X.

Figure 3.7 plots the average lookup query time (measured in terms of the number of memory accesses) versus the maximum depth-constraint value for the two different probability distributions. These figures show that as the maximum depth-constraint is relaxed from $\lceil \log m \rceil$ to higher values, the average lookup time falls quickly, and approaches the entropy of the corresponding distribution (see Table 3.2). An interesting observation from the plots (that we have not been able to explain) is that the simple weight-balancing heuristic DCWBT almost always performs better than the near-optimal MINDPQ algorithm, especially at higher values of maximum depth-constraint.

6.1 Tree reconfigurability

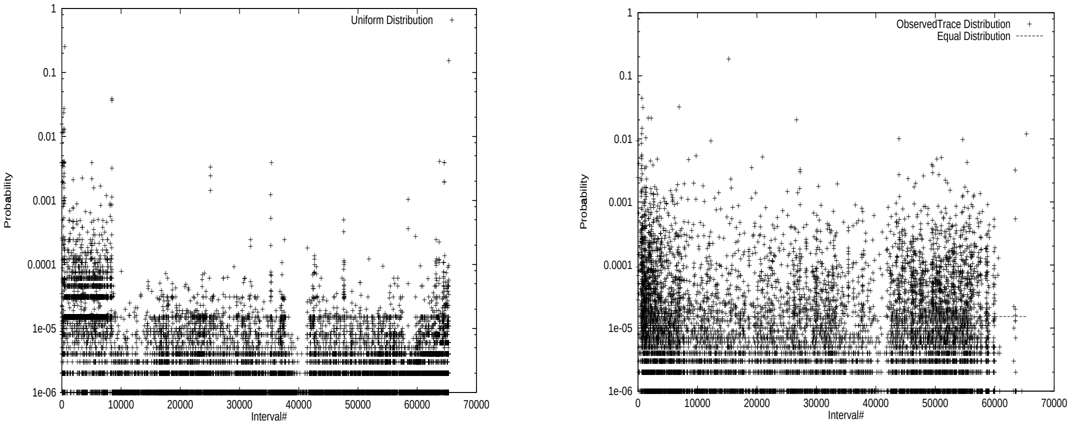
Because routing tables and prefix access patterns are not static, the data-structure build time is an important consideration. This is the amount of time required to compute the optimized tree data structure. Our experiments show that even for the bigger routing table at MAE_EAST, the MINDPQ algorithm takes about 0.96 seconds to compute a new tree, while the DCWBT algorithm takes about 0.40 seconds.[1] The build times for the smaller

VBNS routing table are only 0.033 and 0.011 seconds for the MINDPQ and DCWBT algorithms respectively.

Computation of a new tree could be needed because of two reasons: (1) change in the routing table, or (2) change in the access pattern of the routing table entries. As mentioned in Chapter 2, the average frequency of routing updates in the Internet today is of the order of a few updates per second, even though the peak value can be up to a few hundred updates per second. Changes in the routing table structure can be managed by batching several updates to the routing table and running the tree computation algorithm periodically. The change in access patterns is harder to predict, but there is no reason to believe that it should happen at a very high rate. Indeed, if it does, there is no benefit to optimizing the tree anyway. In practice, we expect that the long term access pattern will not change a lot, while a small change in the probability distribution is expected over shorter time scales. Hence, an obvious way for updating the tree would be to keep track of the current average lookup time as measured by the last few packet lookups in the router, and do a new tree computation whenever this differs from the tree's average weighted depth (which is the expected value of the average lookup time if the packets were obeying the probability distribution) by more than some configurable threshold amount. The tree could also be recomputed at fixed intervals regardless of the changes.

To investigate tree reconfigurability in more detail, the packet trace was simulated with the MAE_EAST routing table. For simplicity, we divided the 2.14 million packet destination addresses in the trace into groups, each group consisting of 0.5M packets. The addresses were fed one at a time to the simulation and the effects of updating the tree simulated after seeing the last packet in every group. The assumed initial condition was the ‘equal’ distribution, i.e., every tree leaf, which corresponds to a prefix interval, is equally

[1] These experiments were carried out by implementing the algorithms in C and running as a user-level process under Linux on a 333 MHz Pentium-II processor with 96 Mbytes of memory.



(a) (b)

Figure 3.8 Showing the probability distribution on the MAE_EAST routing table: (a) "Uniform" probability distribution, i.e., the probability of accessing an interval is proportional to its length, (b) As derived from the packet trace. Note that the "Equal" Distribution corresponds to a horizontal line at y=1.5e-5.

likely to be accessed by an incoming packet. Thus the initial tree is simply the complete tree of depth $\lceil \log m \rceil$. The tree statistics for the MINDPQ trees computed for every group are shown in Table 3.3 for (an arbitrarily chosen) maximum lookup time constraint of 22 memory accesses.

TABLE 3.3. Statistics for the MINDPQ tree constructed at the end of every 0.5 million packets in the 2.14 million packet trace for the MAE_EAST routing table. All times/lengths are specified in terms of the number of memory accesses to reach the leaf of the tree storing the interval. The worst-case lookup time is denoted by luWorst, the average look up time by luAvg, the standard deviation by luSd. and the average weighted depth of the tree by WtDepth.

PktNum	luWorst	luAvg	luSd	Entropy	WtDepth
0-0.5M	16	15.94	0.54	15.99	15.99
0.5-1.0M	22	9.26	4.09	7.88	9.07
1.0-1.5M	22	9.24	4.11	7.88	9.11
1.5-2.0M	22	9.55	4.29	7.89	9.37
2.0-2.14M	22	9.38	4.14	7.92	9.31

The table shows how computing a new tree at the end of the first group brings down the average lookup time from 15.94 to 9.26 memory accesses providing an improvement in the lookup rate by a factor of 1.72. This improvement is expected to be greater if the depth-constraint were to be relaxed further. The statistics show that once the first tree update (at the end of the last packet of the first group) is done, the average lookup time decreases significantly and the other subsequent tree updates do not considerably alter this lookup time. In other words, the access pattern changes only slightly across groups. Figure 3.8(b) shows the probability distribution derived from the trace, and also plots the ‘equal’ distribution (which is just a straight line parallel to the x-axis). Also shown for comparison is the ‘uniform’ distribution in Figure 3.8(a). Experimental results showed that the distribution derived from the trace was relatively unchanging from one group to another, and therefore only one of the groups is shown in Figure 3.8(b).

7 Related work

Early attempts at using statistical properties comprised caching recently seen destination addresses and their lookup results (discussed in Chapter 2). The algorithms considered in this chapter adapt the lookup data structure based on statistical properties of the forwarding table itself, i.e., the frequency with which each forwarding table entry has been accessed in the past. Intuitively, we expect that these algorithms should perform better than caching recently looked up addresses because of two reasons. First, the statistical properties of accesses on a forwarding table are relatively more static (as we saw in Section 6.1) because these properties relate to prefixes that are aggregates of destination addresses, rather than the addresses themselves. Second, caching provides only two discrete levels of performance (good or bad) for all packets depending on whether they take the slow or the fast path. Hence, caching performs poorly when only a few packets take the fast path. In contrast, an algorithm that adapts the lookup data structure itself provides a more continuous level of performance for incoming packets, from the fastest to the slowest, and hence can provide a higher average lookup rate.

Since most previous work on routing lookups has focussed on minimizing worst-case lookup time, the only paper with a similar formulation as ours is by Cheung and McCanne [10]. Their paper also considers the frequency with which a certain prefix is accessed to improve the average time taken to lookup an address. However, reference [10] uses a trie data structure answering the question of "how to redraw a trie to minimize the average lookup time under a given storage space constraint," while the algorithms described here use a binary search tree data structure based on the binary search algorithm [49] discussed in Section 2.2.6 of Chapter 2. Thus, the methods and the constraints imposed in this chapter are different. For example, redrawing a trie typically entails compressing it by increasing the degree of some of its internal nodes. As seen in Chapter 2, this can alter its space consumption. In contrast, it is possible to redraw a binary search tree without changing the amount of space consumed by it, and hence space consumption is not a constraint in this chapter's formulation.

While it is not possible to make a direct comparison with [10] because of the different nature of the problems being solved, we can make a comparison of the complexity of computation of the data structures. The complexity of the algorithm in [10] is stated to be $O(DnB)$ where B is a constant around 10, and $D = 32$, which makes it about $320n$. In contrast, both the MINDPQ and the DCWBT algorithms are of complexity $O(n\log n)$ for n prefixes, which, strictly speaking, is worse than $O(n)$. However, including the constants in calculations, these algorithms have complexity $O(Cn\log n)$, where the constant factor C is no more than 3. Thus even for very large values of n, say $2^{17} = 128K$, the complexity of these algorithms is no more than $96n$.

8 Conclusions and summary of contributions

This chapter motivates and defines a new problem — that of minimizing the average routing lookup time while still keeping the worst case lookup time bounded — and proposes two near-optimal algorithms for this problem using a binary search tree data structure. This chapter explores the complexity, performance and optimality properties of the algorithms. Experiments performed on data taken from routing tables in the backbone of the Internet show that the algorithms provide a performance gain up to a factor of about 1.7. Higher lookup rates can be achieved with low overhead by parallelizing the data structure using its “near-perfect” load balancing property.

Finding good depth-constrained alphabetic and Huffman trees are problems of independent interest, e.g., in computationally efficient compression and coding. The general approach of this chapter, although developed for alphabetic trees for the application of routing lookups, turns out to be equally applicable for solving related problems of independent interest in Information theory — such as finding depth-constrained Huffman trees, and compares favorably to recent work on this topic (for example, Mildiu and Laber [59][60][61] and Schieber [85]). Since this extension is tangential to the subject of this chapter, it is not discussed here.

CHAPTER 4

Recursive Flow Classification: An Algorithm for Packet Classification on Multiple Fields

1 Introduction

Chapters 2 and 3 described algorithms for routing lookups. In this chapter and the next we consider algorithms for multi-field packet classification.[1]

This chapter presents an algorithm for fast packet classification on multiple header fields. The algorithm, though designed with a hardware realization in mind, is suitable for implementation in software as well. As we will see from the overview of previous work on packet classification algorithms in Section 2, the packet classification problem is expensive to solve in the worst-case — theoretical bounds state that solutions to multi-field classification either require storage that is geometric, or a number of memory accesses that is polylogarithmic, in the number of classification rules. Hence, most classi-

[1] The packet classification problem was introduced in Chapter 1: its motivation described in Section 2.1, problem definition in Section 2.3 and the metrics for classification algorithms in Section 3.

fication algorithms proposed in the literature [7][23][96] are designed to work well for two dimensions (i.e., with two header fields), but do not perform as well in multiple dimensions. This is explained in detail in Section 2.

This chapter makes the observation that classifiers in real networks have considerable structure and redundancy that can be exploited by a practical algorithm. Hence, this chapter takes a pragmatic approach, and proposes a heuristic algorithm, called RFC[1] (Recursive Flow Classification), that seems to work well with a selection of classifiers in use today. With current technology, it appears practical to use the proposed classification algorithm for OC192c line rates in hardware and OC48c rates in software. However, the storage space and preprocessing time requirements become large for classifiers with more than approximately 6000 four-field rules. For this, an optimization of the basic RFC algorithm is described which decreases the storage requirements of a classifier containing 15,000 four-field rules to below 4 Mbytes.

1.1 Organization of the chapter

Section 2 overviews previous work on classification algorithms. Section 3 describes the proposed algorithm, RFC, and Section 4 discusses experimental results of RFC on the classifiers in our dataset. Section 5 describes variations of RFC to handle larger classifiers. Section 6 compares RFC with previous work described in Section 2, and finally, Section 7 concludes with a summary and contributions of this chapter.

2 Previous work on classification algorithms

Recall from Section 3 of Chapter 1 that a classification algorithm preprocesses a given classifier to build a data structure, that is then used to find the highest priority matching rule for every incoming packet. We will assume throughout this chapter that rules do not

[1] This is not to be confused with “Request for Comments”.

carry an explicit priority field, and that the matching rule closest to the top of the list of rules in the classifier is the highest priority matching rule. We will work with the following example classifier in this section.

Example 4.1: The classifier C shown in Table 4.1 consists of six rules in two fields (dimensions) labeled $F1$ and $F2$. All field specifications are prefixes of maximum length 3 bits. As per convention, rule priorities are ordered in decreasing order from top to bottom of the classifier.

TABLE 4.1. An example classifier.

Rule	F1	F2
R1	00*	00*
R2	0*	01*
R3	1*	0*
R4	00*	0*
R5	0*	1*
R6	*	1*

2.1 Range lookups

Algorithms that perform classification in multiple dimensions often use a one-dimensional lookup algorithm as a primitive. If the field specifications in a particular dimension are all prefixes, a lookup in this dimension usually involves either finding all matching prefixes or the longest matching prefix — this could be performed using any of the algorithms discussed in Chapters 2 and 3. However, as we will see in Section 3.2, field specifications can be arbitrary ranges. Hence, it will be useful to define the following range lookup problem for a dimension of width W bits.

Definition 1.1: *Given a set of N disjoint ranges $G = \{G_i = [l_i, u_i]\}$ that form a partition of the number line $[0, 2^W - 1]$, i.e., l_i and u_i are such that $l_1 = 0, l_i \le u_i, l_{i+1} = u_i + 1, u_N = 2^W - 1$; the range lookup problem is to find* *the range G_P (and any associated information) that contains an incoming point P.*

We have already seen one algorithm to solve the range lookup problem — the binary search algorithm of Section 2.2.6 in Chapter 2 builds a binary search tree on the endpoints of the set of ranges. We could also solve the range lookup problem by first converting each range to a set of maximal prefixes, and then solving the prefix matching problem on the union of the prefixes thus created. The conversion of a range to prefixes uses the observation that a prefix of length s corresponds to a range $[l, u]$ where the $(W-s)$ least significant bits of l are all 0 and those of u are all 1. Hence, if we split a given range into the minimum number of subranges satisfying this property, we arrive at a set of maximal prefixes equivalent to the original range. Table 4.2 lists examples of some range to prefix conversions for 4-bit fields.

TABLE 4.2. Examples of range to prefix conversions for 4-bit fields.

Range	**Constituent maximal prefixes**
[4,7]	01**
[3,8]	0011, 01**, 1000
[1,14]	0001, 001*, 01**, 10**, 110*, 1110

It can be seen that a range on a W-bit dimension can be split into a maximum of $2W-2$ maximal prefixes.[1] Hence, the range lookup problem can be solved using a prefix matching algorithm, but with the storage complexity increased by a factor of $2W$. Feldmann and Muthukrishnan [23] show a reduction of the range lookup problem to the prefix matching problem with an increase in storage complexity by only a constant factor of 2. However, as we will see later, this reduction cannot be used in all multi-dimensional classification schemes.

[1] For example, the range $[1, 2^W-2]$ is split into $2W-2$ prefixes. An example of this is the last row in Table 4.2 with $W = 4$.

2.2 Bounds from Computational Geometry

There is a simple geometric interpretation of the packet classification problem. We have seen that a prefix represents a contiguous interval on the number line. Similarly, a two-dimensional rule represents an axes-parallel rectangle in the two-dimensional euclidean space of size $2^{W_1} \times 2^{W_2}$, where W_1 and W_2 are the respective widths of the two dimensions. Generalizing, a rule in d dimensions represents a d-dimensional hyperrectangle in d-dimensional space. A classifier is therefore a collection of rectangles, each of which is labeled with a priority. An incoming packet header represents a point with coordinates equal to the values of the header fields corresponding to the d dimensions. For example, Figure 4.1 shows the geometric representation of the classifier in Table 4.1. Rules of higher priority overlay those of lower priority in the figure.

Given this geometric representation, classifying an arriving packet is equivalent to finding the highest priority rectangle among all rectangles that contain the point representing the packet. If higher priority rectangles are drawn on top of lower priority rectangles (as in Figure 4.1), this is equivalent to finding the topmost visible rectangle containing a given point. For example, the packet represented by the point P(011,110) in Figure 4.1 would be classified by rule R_5.

There are several standard problems in the field of computational geometry [4][79][84], such as ray-shooting, point location and rectangle enclosure, that resemble packet classification. Point location in a multi-dimensional space requires finding the enclosing region of a point, given a set of *non-overlapping* regions. Since the hyperrectangles in packet classification could be overlapping, packet classification is at least as hard as point location. The best bounds for point location in N rectangular regions and d dimensions in the worst-case, for $d > 3$, are $O(\log N)$ time with $O(N^d)$ space;[1] or $O((\log N)^{d-1})$ time and $O(N)$ space [73][79]. Clearly this is impracticably slow for classi-

[1] The time bound for $d \leq 3$ is $O(\log\log N)$ [73] but has large constant factors.

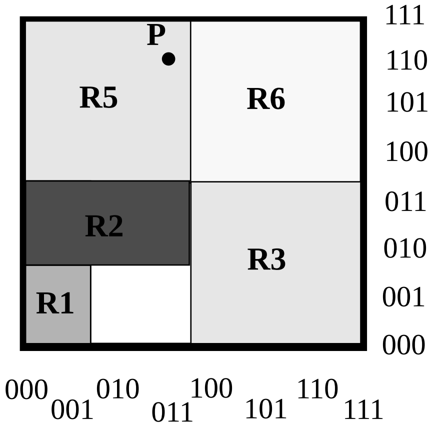


Figure 4.1 Geometric representation of the two-dimensional classifier of Table 4.1. An incoming packet represents a point in the two dimensional space, for instance, P(011,110). Note that R4 is completely hidden by R1 and R2.

fication in a high speed router — with just 100 rules and 4 fields, N^d space is about 100 Mbytes; and $(\log N)^{d-1}$ is about 350 memory accesses.

2.3 Linear search

As in the routing lookup problem, the simplest data structure is a linked-list of all the classification rules, possibly stored in sorted order of decreasing priorities. For every arriving packet, each rule is evaluated sequentially until a rule is found that matches all the relevant fields in the packet header. While simple and storage-efficient, this algorithm clearly has poor scaling properties: the time to classify a packet grows linearly with the number of rules.[1]

2.4 Ternary CAMs

We saw in Section 2.2.8 of Chapter 2 how ternary CAMs (TCAMs) can be used for performing longest prefix matching operations in dedicated hardware. TCAMs can similarly be used for multi-dimensional classification with the modification that each row of the TCAM memory array needs to be wider than 32 bits — the required width depends on

[1] Practical evidence suggests that this data structure can support a performance between 10,000 and 30,000 packets per second using a 200 MHz CPU with a few hundred 4-dimensional classification rules.

the number of fields used for classification, and usually varies between 128 and 256 bits depending on the application. An increasing number of TCAMs are being used in the industry at the time of writing (for at least some applications) because of their simplicity, speed (the promise of classification in a single clock-cycle), improving density, and possibly absence of competitive algorithmic solutions. While the same advantages and disadvantages as discussed in Chapter 2 hold for a classification TCAM, we look again at a few issues specifically raised by classification.

- **Density**: The requirement of a wider TCAM further decreases its depth for a given density. Hence, for a 2 Mb 256-bit wide TCAM, at most 8K classification rules can be supported. As a TCAM row stores a (value, mask) pair, range specifications need to be split into mask specifications, further bringing down the number of usable TCAM entries by a factor of $(2W-2)^d$ in the worst case for d-dimensional classification. Even if only two 16-bit dimensions specify ranges (which is quiet common in practice with the transport-layer source and destination port number fields), this is a multiplicative factor of 900.
- **Power**: Power dissipated in one TCAM row increases proportionally to its width.

In summary, classification makes worse the disadvantages of existing TCAMs. Because of these reasons, TCAMs will probably still remain unsuitable in the near future for the following situations: (1) Large classifiers (256K-1M rules) used for microflow recognition at the edge of the network, (2) Large classifiers (128-256K rules) used at edge routers that manage thousands of subscribers (with a few rules per subscriber), (3) Extremely high speed (greater than 200-250 Mpps) classification, and (4) Software-based classification that may be required for a large number of dimensions, for instance, more than 8.

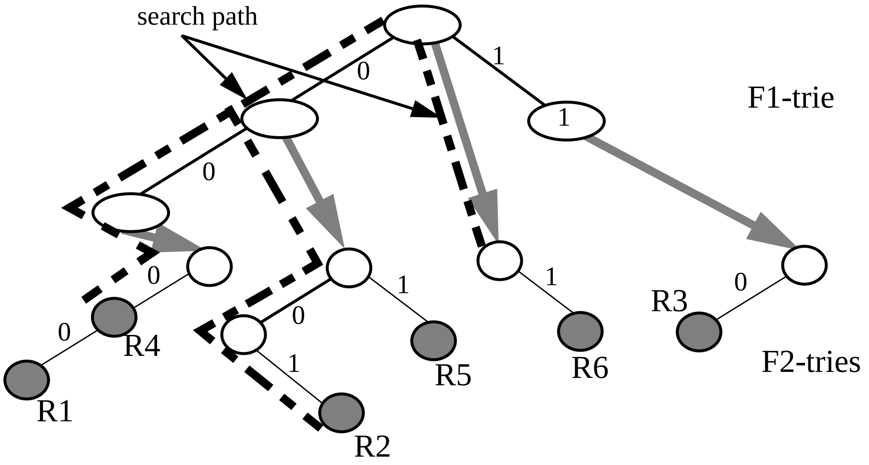


Figure 4.2 The hierarchical trie data structure built on the rules of the example classifier of Table 4.1. The gray pointers are the “next-trie” pointers. The path traversed by the query algorithm on an incoming packet (000, 010) is also shown.

2.5 Hierarchical tries

A d-dimensional hierarchical radix trie is a simple extension of the radix trie data structure in one dimension (henceforth called a 1-dimensional trie), and is constructed recursively as follows. If d equals 1, the hierarchical trie is identical to the 1-dimensional radix trie studied before in Section 2.1.3 of Chapter 2. If d is greater than 1, we first construct a 1-dimensional trie on say dimension $F1$, called the $F1$-trie. Hence, the $F1$-trie is a ‘trie’ on the set of prefixes $\{R_{j1}\}$, belonging to dimension $F1$ of all rules in the classifier, $C = \{R_j\}$, where $R_j = \{R_{j1}, R_{j2}\}$. For each prefix, p, in the 1-dimensional $F1$-trie, we recursively construct a $(d-1)$-dimensional hierarchical trie, T_p, on those rules which exactly specify p in dimension $F1$, in other words, on the set of rules $\{R_j : R_{j1} = p\}$. Prefix p is linked to the trie T_p using another pointer called the next-trie pointer. For instance, the data structure in two dimensions is comprised of one $F1$-trie and several $F2$-tries linked to nodes in the $F1$-trie. The storage complexity of the data structure for an N-rule classifier is $O(NdW)$. The hierarchical trie data structure for the example classifier of

Table 4.1 is shown in Figure 4.2. Hierarchical tries are sometimes called “multi-level tries,” “backtracking-search tries,” or “trie-of-tries.”

A classification query on an incoming packet $(v_1, v_2, \ldots, v_d)$ proceeds recursively on each dimension as follows. The query algorithm first traverses the 1-dimensional $F1$-trie based on the bits in v_1 in the usual manner. At each $F1$-trie node encountered during this traversal, the algorithm follows the next-trie pointer (if non-null) and recursively traverses the $(d-1)$-dimensional hierarchical trie stored at that node. Hence, this query algorithm encounters a rule in its traversal if and only if that rule matches the incoming packet, and it need only keep track of the highest priority rule encountered. Because of its recursive nature, the query algorithm is sometimes referred to as a backtracking search algorithm. The query time complexity for d-dimensions is $O(W^d)$. Incremental updates can be carried out in $O(d^2W)$ time since each of the d-prefix components of the updated rule is stored in exactly one location at maximum depth $O(dW)$ in the data structure. As an example, the path traversed by the classification query algorithm for an incoming packet (000,010) is also shown in Figure 4.2.

2.6 Set-pruning tries

A set-pruning trie [106] is similar to a hierarchical trie but with reduced data structure query time obtained by eliminating the need for doing recursive traversals. This is achieved by replicating rules at several nodes in the data structure as follows. Consider a d-dimensional hierarchical trie consisting of an $F1$-trie and several $(d-1)$-dimensional hierarchical tries. Let S be the set of nodes representing prefixes longer than a prefix p in the $F1$-trie. A set-pruning trie is similar to this hierarchical trie except that the rules in the $(d-1)$-dimensional hierarchical trie linked to a prefix p in the $F1$-trie are “pushed down,” i.e., replicated in the $(d-1)$-dimensional hierarchical tries linked to all the nodes in S. This “pushing-down” of prefixes is carried out recursively (during preprocessing) on the remaining $(d-1)$ dimensions in the set-pruning trie data structure.

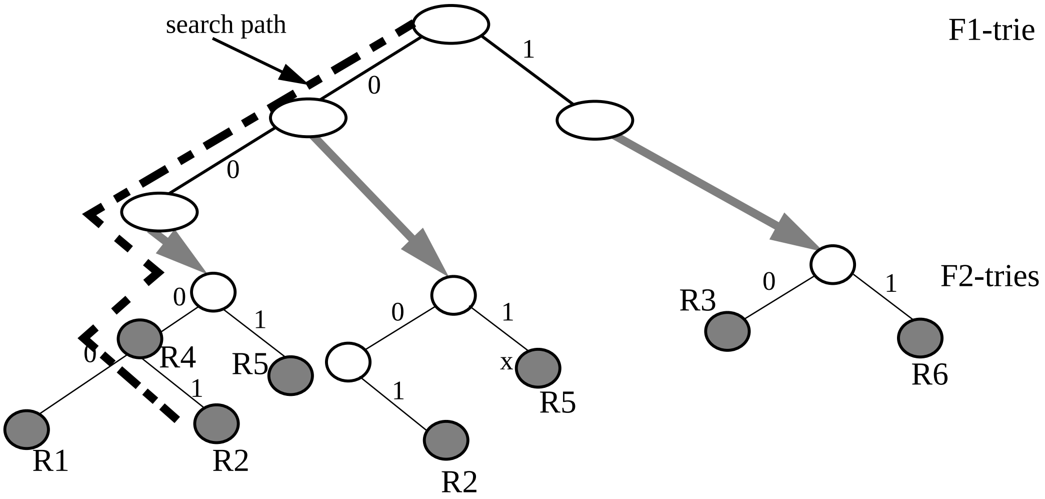


Figure 4.3 The set-pruning trie data structure built on the rules of example classifier of Table 4.1. The gray pointers are the "next-trie" pointers. The path traversed by the query algorithm on an incoming packet (000, 010) is also shown.

The query algorithm for an incoming packet $(v_1, v_2, \ldots, v_d)$ now need only traverse the $F1$-trie to find the longest matching prefix of v_1, follow its next-trie pointer (if non-null), traverse the $F2$-trie to find the longest matching prefix of v_1, and so on for all dimensions. The manner of replication of rules ensures that every matching rule will be encountered in this path. The query time complexity reduces to $O(dW)$ at the expense of an increased storage complexity of $O(N^d dW)$ since a rule may need to be replicated $O(N^d)$ times — for every dimension k, the k^{th} prefix component of a rule may be longer than $O(N)$ other k^{th} prefix components of other rules in the classifier. Update complexity is $O(N^d)$, and hence, this data structure is, practically speaking, static.

The set-pruning trie for the example classifier of Table 4.1 is shown in Figure 4.3. The path traversed by the query algorithm on an incoming packet (000,010) is also shown. Note that replication may lead to prefix components of different rules being allocated to the same trie node. When this happens, only the highest priority rule need be stored at that

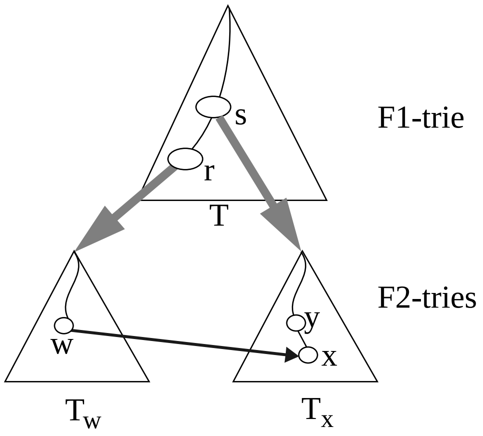


Figure 4.4 Showing the conditions under which a switch pointer is drawn from node w to node x. The pointers out of nodes s and r to tries T_x and T_w respectively are next-trie pointers.

node — for instance, both R5 and R6 are allocated to node x in the $F2$-trie of Figure 4.4, but the node x stores only the higher priority rule R5.

2.7 Grid-of-tries

The grid-of-tries data structure, proposed by Srinivasan et al [95], is an optimization of the hierarchical trie data structure for two dimensions. This data structure avoids the memory blowup of set-pruning tries by allocating a rule to only one trie node as in hierarchical tries. However, it still achieves $O(W)$ query time by using pre-computation and storing a *switch pointer* in some trie nodes. A switch pointer is labeled ‘0’ or ‘1’ and guides the search process in the manner described below. The conditions which must be satisfied for a switch pointer labeled b (b = ’0’ or ‘1’) to exist from a node w in the trie T_w to a node x of another trie T_x are (see Figure 4.4):

1. T_x and T_w are distinct tries built on the prefix components of dimension $F2$. Furthermore, T_x and T_w are respectively pointed to by the next-trie pointers of two distinct nodes, say r and s of the same trie, T, built on prefix components of dimension $F1$.

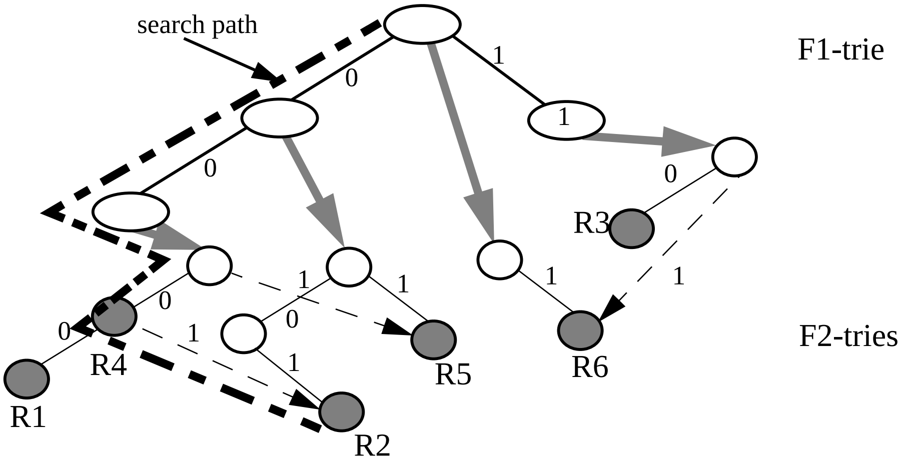


Figure 4.5 The grid-of-tries data structure built on the rules of example classifier in Table 4.1. The gray pointers are the “next-trie” pointers, and the dashed pointers are the switch pointers. The path traversed by the query algorithm on an incoming packet (000, 010) is also shown.

2. The bit-string that denotes the path from the root node to node w in trie T_w concatenated with the bit b is identical to the bit-string that denotes the path from the root node to node x in the trie T_x.
3. Node w does not have a child pointer labeled b, and
4. Node s in trie T is the closest ancestor of node r that satisfies the above conditions.

If the query algorithm traverses paths $U1(s, root(T_x), y, x)$ and $U2(r, root(T_w), w)$ for an incoming packet on the hierarchical trie, the query algorithm need only traverse the path $U(s, r, root(T_w), w, x)$ on a grid-of-tries data structure. This is because paths $U1$ and $U2$ are identical (by condition 2 above) till $U1$ terminates at node w because w does not have a child branch labeled b (by condition 3). The use of another pointer, called a “switch pointer,” from node w directly to node x allows the grid-of-tries query algorithm to traverse all branches that would have been traversed by the hierarchical trie query algorithm without the need to ever backtrack. This new algorithm examines each bit of the incoming packet header at most once. Hence, the time complexity reduces to $O(W)$, while storage complexity of $O(NW)$ remains identical to that of 2-dimensional hierarchical tries.

However, adding switch pointers to the hierarchical trie data structure makes incremental updates difficult to support, so the authors recommend rebuilding the data structure (in time $O(NW)$) in order to carry out updates [95]. The grid-of-tries data structure for the example classifier of Table 4.1 is shown in Figure 4.5, along with an example path traversed by the query algorithm.

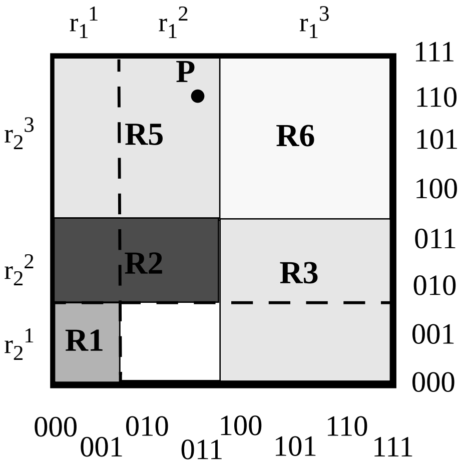
Reference [95] reports a memory usage of 2 Mbytes on a classifier containing 20,000 rules in two dimensions comprising destination and source IP prefixes, when the stride of the destination prefix trie is 8 bits and that of the source prefix tries is 5 bits. The worst case number of memory accesses is therefore 9. The classifier was constructed by using a publicly available routing table for the destination IP dimension and choosing prefixes from this routing table randomly to form the source IP dimension.

Grid-of-tries is a good data structure for two dimensional classification occupying reasonable amount of memory and requiring a few memory accesses. It can be used as an optimization for the last two dimensions of a multi-dimensional hierarchical trie, hence decreasing the classification time complexity by a factor of W to $O(NW^{d-1})$ in d dimensions, in the same amount of storage $O(NdW)$. As with hierarchical and set-pruning tries, grid-of-tries requires range specifications to be split into prefixes before the data structure is constructed.

2.8 Crossproducting

Crossproducting [95] is a packet classification solution suitable for an arbitrary number of dimensions. The idea is to classify an incoming packet in d dimensions by composing the results of separate 1-dimensional range lookups in each dimension as follows.

The preprocessing step to construct the data structure comprises computing the set of ranges, G_k, of size $s_k = |G_k|$, projected by rule specifications in each dimension $k, 1 \le k \le d$. Let r_k^j, $1 \le j \le s_k$, denote the j^{th} range in G_k. A crossproduct table C_T of size



Crossproduct Table

(r_1^1,r_2^1)	**R1**
(r_1^1,r_2^2)	**R2**
(r_1^1,r_2^3)	**R5**
(r_1^2,r_2^1)	—
(r_1^2,r_2^2)	**R2**
(r_1^2,r_2^2)	**R5**
(r_1^3,r_2^1)	**R3**
(r_1^3,r_2^2)	**R3**
(r_1^3,r_2^3)	**R5**

Figure 4.6 The table produced by the crossproducting algorithm and its geometric representation of the two-dimensional classifier of Table 4.1.

$\prod_{k=1}^{d} s_k$ is then constructed, and the best matching rule for each entry $\left(r_1^{i_1}, r_2^{i_2}, \ldots, r_d^{i_d}\right)$, $1 \le i_k \le s_k$, $1 \le k \le d$ in this table is precomputed and stored.

Classification query on an incoming packet $(v_1, v_2, \ldots, v_d)$ first performs a range lookup in each dimension k to identify the range $r_k^{i_k}$ containing point v_k. The tuple $\langle r_1^{i_1}, r_2^{i_2}, \ldots, r_d^{i_d} \rangle$ is then directly looked up in the crossproduct table C_T to access the precomputed best matching rule.

Example 4.5: The crossproduct table for the example classifier of Table 4.1 is shown in Figure 4.6. The figure also illustrates the geometric interpretation of crossproducting. There is one entry in the crossproduct table for each rectangular cell in the grid created by extending the sides of each original rectangle representing a rule. The query algorithm for an example incoming packet P(011,110) accesses table entry with the address (r_1^2, r_2^3) accessing rule R5.

We have seen that N prefixes give rise to at most $2N$ ranges, hence, $s_k \le 2N$, and C_T is of size $O(N^d)$. The lookup time is $O(dt_{RL})$ where t_{RL} is the time complexity of doing a range lookup in one dimension. Crossproducting is a suitable solution for very small classifiers only because of its high worst case storage complexity. Reference [95] proposes using an on-demand crossproducting scheme together with caching for classifiers bigger than 50 rules in five dimensions. Crossproducting is a static solution since addition of a rule could change the set of projected ranges and necessitate re-computing the crossproduct table.

2.9 Bitmap-intersection

The bitmap-intersection classification scheme, proposed by Lakshman and Stiliadis [48], is based on the observation that the set of rules, S, that match a packet header, is the intersection of d sets, S_i, where S_i is the set of rules that match the packet in the i^{th} dimension alone. While crossproducting precomputes S and stores the best matching rule in S, this scheme computes S and the best matching rule on the fly, i.e., during each classification operation.

In order to compute intersection of sets efficiently in hardware, each set is encoded as an N-bit bitmap with one bit corresponding to each of the N rules. The set of matching rules is then the set of rules whose corresponding bits are ‘1’ in the bitmap. A classification query on a packet, P, proceeds in a fashion similar to crossproducting by first performing separate range lookups in each of the d dimensions. Each range lookup returns a bitmap encoding the set of matching rules (precomputed for each range) in that dimension. The d sets are intersected (by a simple hardware boolean AND operation) to give the set of rules that match P. The best matching rule is then computed from this set. See Figure 4.7 for the bitmaps corresponding to the example classifier of Table 4.1.

Since each bitmap is N bits wide, and there are $O(N)$ of ranges in each of the d dimensions, the total amount of storage space consumed is $O(dN^2)$. The classification time complexity is $O(dt_{RL} + dN/w)$ where t_{RL} is the time to do one range lookup and w is the memory width so that it takes N/w memory operations to access one bitmap. Time com-

Dimension 1

r_1^1	{R1,R2,R4,R5,R6}	110111
r_1^2	{R2,R5,R6}	010011
r_1^3	{R3,R6}	001001

Dimension 2

r_2^1	{R1,R3,R4}	101100
r_2^2	{R2,R3,R4}	011000
r_2^3	{R5,R6}	000111

Query on P(011,010):

010011 Dimension-1 bitmap
000111 Dimension-2 bitmap

000011 Intersected bitmap
R5 Best matching rule

Figure 4.7 The bitmap tables used in the “bitmap-intersection” classification scheme for the example classifier of Table 4.1. See Figure 4.6 for a description of the ranges. Also shown is classification query on an example packet P(011, 110).

plexity can be brought down by a factor of d by using parallelism in hardware to lookup each dimension independently in parallel. Incremental updates are not supported. The same scheme can be implemented in software, but the classification time is expected to be higher because of the unavailability of hardware-specific features, such as parallelism and bitmap-intersection.

Reference [48] reports that the scheme could support up to 512 rules with a 33 MHz FPGA device and five 1 Mbit SRAMs, classifying one million packets per second. The scheme works well for a small number of rules in multiple dimensions, but suffers from a quadratic increase in storage space and a linear increase in memory bandwidth requirements (and hence in classification time) with the size of the classifier. A variation is described in [48] that decreases the storage requirement at the expense of increased classification time.

2.10 Tuple space search

The idea of the basic tuple space search algorithm (Suri et al [96]) is to decompose a classification query into a number of exact match queries. The algorithm first maps each d-dimensional rule into a d-tuple whose i^{th} component stores the length of the prefix

Rule	Specification	Tuple
R1	(00*,00*)	(2,2)
R2	(0**,01*)	(1,2)
R3	(1**,0**)	(1,1)
R4	(00*,0**)	(2,1)
R5	(0**,1**)	(1,1)
R6	(***,1**)	(0,1)

Tuple	Hash Table Entries
(0,1)	{R6}
(1,1)	{R3,R5}
(1,2)	{R2}
(2,1)	{R4}
(2,2)	{R1}

Figure 4.8 The tuples and associated hash tables in the tuple space search scheme for the example classifier of Table 4.1.

specified in the i^{th} dimension (the scheme supports only prefix specifications). Hence, the set of rules mapped to the same tuple are of a fixed and known length, and thus stored in a hash table for exact match query operations. A classification query is carried out by performing exact match operations on each of the hash tables corresponding to all possible tuples in the classifier. The tuples and their corresponding hash tables for the example classifier of Table 4.1 are shown in Figure 4.8. A variation of the basic algorithm uses heuristics to avoid searching all hash tables using ideas similar to those used in the “binary search on prefix lengths” lookup scheme mentioned in Section 2.2.5 of Chapter 2 (see [96] for details).

Classification time in the tuple space search scheme is equal to the time needed for M hashed memory accesses, where M is the number of tuples in the classifier. The scheme uses $O(N)$ storage since each rule is stored in exactly one hash table. Incremental updates are supported and require just one hashed memory access to the hash table associated with the tuple of the modified rule. In summary, the tuple space search algorithm performs well for multiple dimensions in the average case if the number of tuples is small. However, the use of hashing makes the time complexity of searches and updates non-deterministic. Also, the number of tuples could be very large, up to $O(W^d)$, in the worst case. Furthermore, since the scheme supports only prefixes, the storage complexity increases by a factor of $O(W^d)$ for generic rules as each range could be split into $O(W)$ prefixes in the

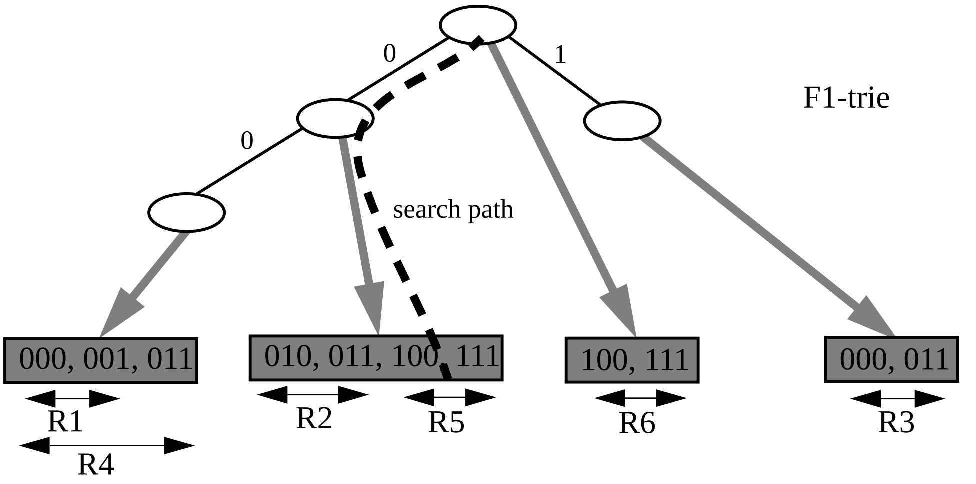


Figure 4.9 The data structure of Section 2.11 for the example classifier of Table 4.1 The search path for example packet P(011, 110) resulting in R5 is also shown.

manner explained in Section 2.1. This is one example where the range-to-prefix transformation technique of [23] cannot be applied because all fields are looked up simultaneously.

2.11 A 2-dimensional classification scheme from Lakshman and Stiliadis [48]

Lakshman and Stiliadis [48] propose a 2-dimensional classification algorithm where one dimension, say $F1$, is restricted to having prefix specifications, while the second dimension, $F2$, is allowed to have arbitrary range specifications. The data structure first builds an $F1$-trie on the prefixes of dimension $F1$, and then associates a set G_w of non-overlapping ranges to each trie node, w, that represents prefix p. These ranges are created by the end-points of possibly overlapping projections on dimension $F2$ of those rules, S_w, that specify exactly p in dimension $F1$. A range lookup data structure (e.g., an array or a binary search tree) is then constructed on G_w and associated with trie node w. The data structure for the example classifier of Table 4.1 is shown in Figure 4.9.

Given a point $P(v_1,v_2)$, the query algorithm proceeds downwards from the root of the trie according to the bits of v_1 in the usual manner. At every trie node, w, encountered during this traversal, a range lookup is performed on the associated data structure G_w. The range lookup operation returns the range in G_w containing v_2, and hence the best matching rule, say R_w, within the set S_w that matches point P. The highest priority rule among the rules $\{R_w\}$ for all trie nodes w encountered during the traversal is the desired highest priority matching rule in the classifier.

The query algorithm takes time $O(W \log N)$ because a range lookup needs to be performed (in $O(\log N)$ time) at every trie node in the path from the root to a null node in the $F1$-trie. This can be improved to $O(W + \log N)$ using a technique called *fractional cascading* borrowed from Computational Geometry [4]. This technique augments the data structure such that the problem of searching for the same point in several sorted lists is reduced to searching in only one sorted list plus accessing a constant number of elements in the remaining lists. The storage complexity is $O(NW)$ because each rule is stored only once in the data structure. However, the use of fractional cascading renders the data structure static.

2.12 Area-based quadtree

The Area-based Quadtree (AQT) data structure proposed by Buddhikot et al [7] for classification in two dimensions supports incremental updates that can be traded off with classification time by a tunable parameter. The preprocessing algorithm first builds a quadtree [4], a tree in which each internal node has four children. The parent node of a quadtree represents a two dimensional space that is decomposed into four equal sized quadrants, each of which is represented by a child of that node. The original two dimensional space is thus recursively decomposed into four equal-sized quadrants till each quadrant has less than or equal to one rule in it (see Figure 4.10 for an example of the

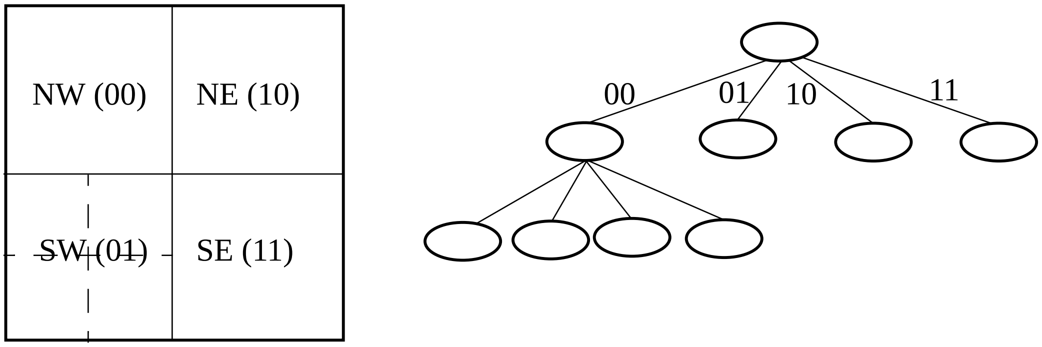


Figure 4.10 An example quadtree constructed by spatial decomposition of two-dimensional space. Each decomposition results in four quadrants.

decomposition process). A set of rules is then allocated to each node of the quadtree in the manner described next.

A rule is said to cross a quadrant in dimension j if it completely spans the dimension-j of the area represented by that quadrant. For instance, rule R6 spans in both dimensions the quadrant represented by the root node (the complete 2-dimensional space) of Figure 4.11, while rule R5 does not. If we divide the 2-dimensional space into four quadrants, rule R5 crosses the north-west quadrant in both dimensions while rule R2 crosses the south-west quadrant in dimension-$F1$. The set of rules crossing the quadrant represented by a node in dimension k is called the "k-crossing filter set (k-CFS)" of that node.

Two instances of the same data structure are associated with each quadtree node — one each for storing the rules in k-CFS ($k = 1, 2$). Since rules in crossing filter sets span at least one of the two dimensions, only the range specified in the other dimension need be stored in the data structure. The classification query proceeds by traversing the quadtree according to the bits in the given packet — looking at two bits at a time, formed by transposing one bit from each dimension. The query algorithm does two 1-dimensional look-

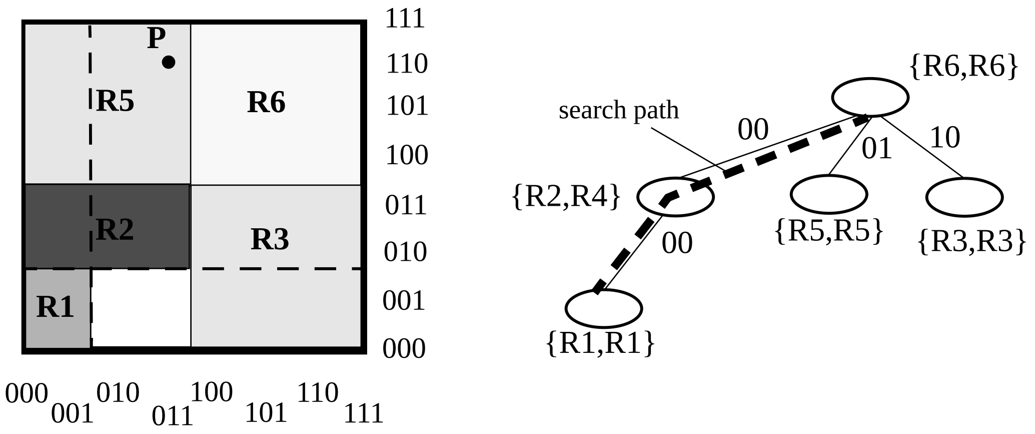


Figure 4.11 The AQT data structure for the classifier of Table 4.1. The label of each node denotes {1-CFS, 2-CFS}. Also shown is the path traversed by the query algorithm for an incoming packet P(001, 010), yielding R1 as the best matching rule.

ups (one for each dimension on k-CFS) at each quadtree node traversed. Figure 4.11 shows the AQT data structure for the example classifier of Table 4.1.

Reference [7] also proposes an efficient incremental update algorithm that enables AQT to achieve the following bounds for N two-dimensional rules: $O(NW)$ space complexity, $O(\alpha W)$ search time and $O(\alpha\sqrt[\alpha]{N})$ update time for a tunable integral parameter α.

2.13 Fat Inverted Segment Tree (FIS-tree)

Feldmann and Muthukrishnan [23] propose the FIS-tree data structure for two dimensional classification as a modification of the segment tree data structure. We first describe the segment tree data structure, and then the FIS-tree data structure.

A segment tree [4] stores a set S of line segments (possibly overlapping) to answer queries such as finding the highest priority line segment containing a given point efficiently. It consists of a balanced binary search tree on the end points of the line segments in S. Each node, w, of a segment tree represents a range G_w — leaves represent the original line segments in S, and parent nodes represent the union of the ranges represented by

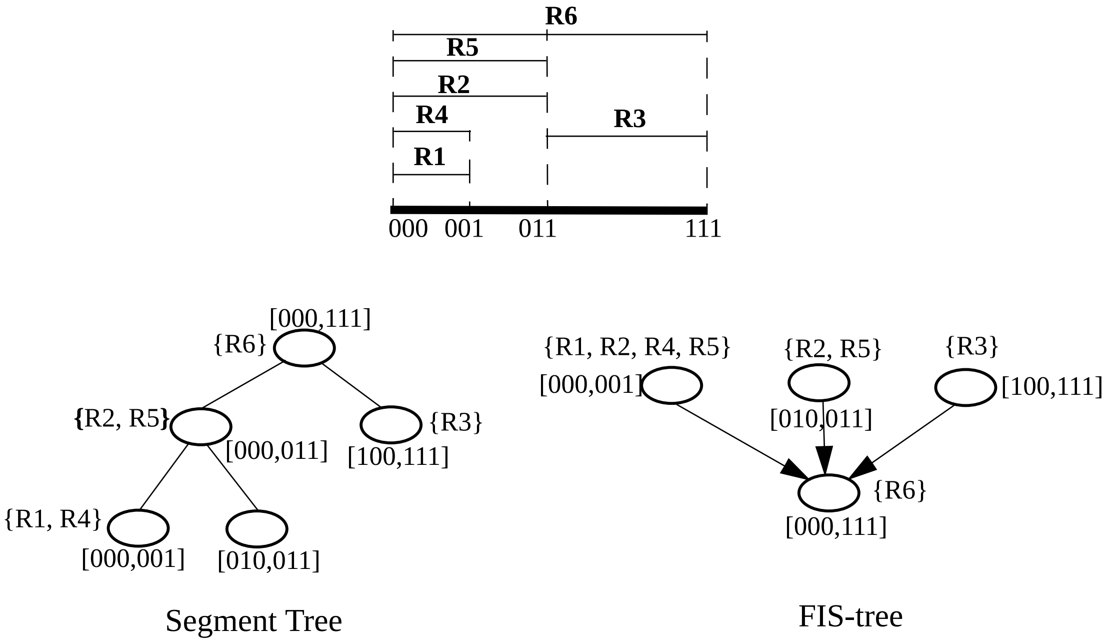


Figure 4.12 The segment tree and the 2-level FIS-tree for the classifier of Table 4.1.

their children. A line segment is allocated to a node w if it contains G_w but does not contain $G_{parent(w)}$. The highest priority line segment among all the line segments allocated to a node is precomputed and stored at the node. The search algorithm for finding the highest priority line segment containing a given point traverses the segment tree downwards from the root, and calculates the highest priority of all the precomputed segments encountered at each node during its traversal. Figure 4.12 shows the segment tree for the line segments created by the $F1$-projections of the rules of classifier in Table 4.1.

An FIS-tree is a segment tree with two modifications: (1) The segment tree is compressed (made "fat" by increasing the degree to more than two) in order to decrease its depth so that it occupies a given number of levels l. (2) Pointers are set up inverted, i.e., go from child nodes to the parent to help the search process described below. The classification data structure for 2-dimensional classifiers consists of an FIS-tree on dimension $F1$, and a range lookup data structure associated with each node of the FIS-tree. An instance of the range lookup data structure associated with node w of the FIS-tree stores the ranges formed by the $F2$-projections of those classifier rules whose $F1$-projections were allocated to w.

A classification query on a given point $P(v_1,v_2)$ first solves the range lookup problem in dimension $F1$. This returns a leaf node of the FIS-tree representing the range containing the point v_1. The query algorithm then follows the parent pointers from this leaf node up towards the root node, carrying out 1-dimensional range lookups in the associated range lookup data structures at each node traversed. The algorithm finally computes the highest priority rule containing the given point at the end of the traversal.

The search time complexity for an l-level FIS-tree is $O((l+1)\,t_{RL})$ with a storage space complexity of $O(ln^{1+1/l})$, where t_{RL} is the time taken to carry out a 1-dimensional range lookup. Storage space can be traded off with search time by suitably tuning the parameter l. Several variations to the FIS-tree are needed in order to support incremental updates — even then, it is easier to support inserts than deletes [23]. The static FIS-tree can be extended to multiple dimensions by building hierarchical FIS-trees, but the bounds obtained are similar to other data structures studied earlier. (Please see [23] for details on supporting updates in FIS trees and multi-dimensional static FIS trees).

Extensive measurements on real-life 2-dimensional classifiers are reported in [23] using the static FIS-tree data structure. These measurements indicate that two levels suffice in the FIS tree for 4-60K rules with a storage consumption of less than 5 Mbytes. One classification operation requires fewer than 15 memory accesses. For larger classifiers containing up to one million 2-dimensional rules, at least 3 levels are required with a storage consumption of approximately 100 Mbytes, while one classification operation requires fewer than 18 memory accesses.

2.14 Summary of previous work

Table 4.3 gives a summary of the complexities of the multi-dimensional classification algorithms reviewed in this chapter. Most proposed algorithms work well for two dimensions, but do not extend to multiple dimensions. Others have either non-deterministic search time (e.g., tuple space search), or do not scale to classifiers larger than a few hundred rules (e.g., crossproducting or bitmap-intersection). This is not surprising since theoretical bounds tell us that multi-dimensional classification has poor worst-case performance, in either storage or time complexity.

TABLE 4.3. Comparison of the complexities of previously proposed multi-dimensional classification algorithms on a classifier with N rules and d W-bit wide dimensions. The results assume that each rule is stored in $O(1)$ space and takes $O(1)$ time to determine whether it matches a packet. This table ignores the multiplicative factor of $(2W-2)^d$ in the storage complexity caused by splitting of ranges to prefixes.

Algorithm	**Worst-case time complexity**	**Worst-case storage complexity**
Linear Search	N	N
Hierarchical tries	W^d	NdW
Set-pruning tries	dW	$N^d dW$
Grid-of-tries	W^{d-1}	NdW
Crossproducting	dW	N^d
Bitmap-intersection	$(W + N/memwidth)\, d$	dN^2
Tuple space search	N	N
FIS-tree	$(l+1)\,W$	$l \times N^{1+1/l}$
Ternary CAM	1	N

3 Proposed algorithm RFC (Recursive Flow Classification)

3.1 Background

The RFC algorithm is motivated by the observation that real-life classifiers contain a large amount of structure and redundancy that can be exploited by a pragmatic classification algorithm. RFC works well for a selection of multi-dimensional real-life classifiers available to us. We proceed to describe the observed characteristics of these real-life classifiers and a description of the structure present in them.

3.2 Characteristics of real-life classifiers

We collected 793 packet classifiers from 101 different ISP and enterprise networks with a total of 41,505 rules. For privacy reasons, sensitive information such as IP addresses were sanitized while preserving the relative structure in the classifiers.[1] Each network provided up to ten separate classifiers for different services.[2] We found the classifiers to have the following characteristics:

1. The classifiers do not contain a large number of rules. Only 0.7% of the classifiers contain more than 1000 rules, with a mean of 50 rules. The distribution of the number of rules in a classifier is shown in Figure 4.13. The relatively small number of rules per classifier should not come as a surprise: in most networks today, rules are configured manually by network operators, and it is a non-trivial task to ensure correct behavior if the classifier becomes large.
2. The syntax of these classifiers allows a maximum of 8 header fields to be specified: source/destination network-layer address (32-bits), source/destination transport-layer port numbers (16-bits for TCP and UDP), type-of-service (TOS) field

[1] We wanted to preserve the properties of set relationship, e.g. inclusion, among the rules, or their fields. A 32-bit IP address _p0.p1.p2.p3_ is sanitized as follows: (a) A random 32-bit number _c0.c1.c2.c3_ is first chosen, (b) a random permutation of the 256 numbers 0...255 is then generated to get _perm[0..255]_ (c) Another random number _S_ between 0 and 255 is generated: these randomly generated numbers are common for all the rules in the classifier, (d) The IP address with bytes: _perm[(p0 ^ c0 + 0 * s) % 256]_, _perm[(p1 ^ c1 + 1 * s) % 256]_, _perm[(p2 ^ c2 + 2 * s) % 256]_ and _perm[(p3 ^ c3 + 3 * s) % 256]_ is then returned as the sanitized transformation of the original IP address, where ^ denotes the exclusive-or operation. This transformation preserves set relationship across bytes but not necessarily within a byte. Hence, some structure present in the original classifier may be lost. However, we have since had access to some of the original classifiers, with results similar to those shown in this chapter.

[2] In the collected dataset, classifiers for different services are made up of one or more ACLs (access control lists). An ACL rule can have one of two actions, “deny” or “permit”. In this discussion, we will assume that each ACL is a separate classifier, a common case in practice.

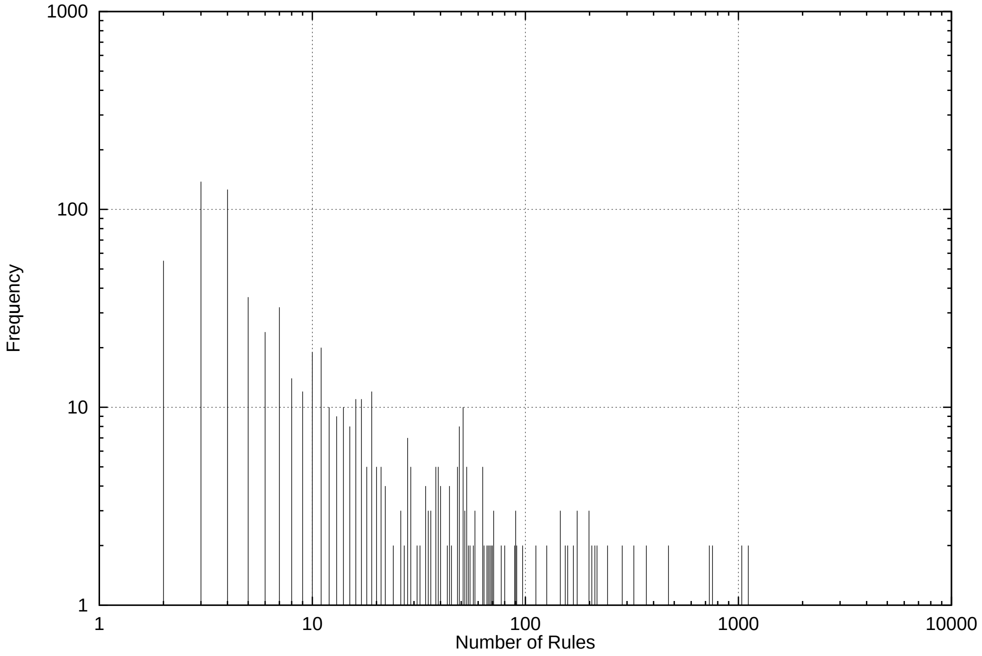


Figure 4.13 The distribution of the total number of rules per classifier. Note the logarithmic scale on both axes.

(8-bits), protocol field (8-bits), and transport-layer protocol flags (8-bits) with a total of 120 bits. 17% of all rules in the dataset have 1 field specified, 23% have 3 fields specified and 60% have 4 fields specified.[1]

3. The transport-layer protocol field is restricted to a small set of values: in our dataset, this field contained only the following values: TCP, UDP, ICMP, IGMP, (E)IGRP, GRE and IPINIP, or the wildcard ‘*’ (i.e., the set of all transport protocols).

4. The transport-layer address fields have a wide variety of specifications. Many (10.2%) of them are *range* specifications — such as ‘range 20-24’ or ‘gt 1023,’ which means all values greater than 1023. In particular, the specification ‘gt 1023’ occurs in about 9% of the rules. Splitting this range into prefixes results in six constituent maximal prefixes: 1024-2047, 2048-4095, 4096-8191, 8192-16383, 16384-32767, 32768-65535. Thus, converting all range specifications to prefix specifications could result in a large increase in the size of a classifier.

[1] If a field is not specified, the wildcard specification is assumed. Note that this is determined by the syntax of the rule specification language.

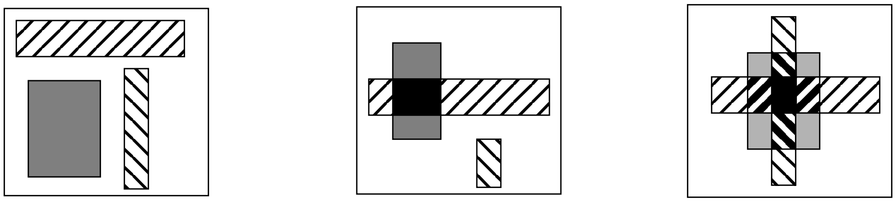
5. Approximately 14% of all classifiers had at least one rule with a non-contiguous mask, and 10.2% of all rules had non-contiguous masks. A non-contiguous mask means that the bits that are '1' in the mask are not contiguous. For example, a specification of 137.98.217.0/8.22.160.80 has a non-contiguous mask, which is surprising. One suggested reason for this is that some network operators choose a specific numbering/addressing scheme for their routers. This observation indicates that a packet classification algorithm cannot always rely on a network-layer address specification to be a prefix.

6. It is common for different rules in the same classifier to share a number of field specifications. Sharing occurs because a network operator frequently wants to specify the same policy for a pair of communicating groups of hosts or subnetworks — for instance, the network operator may want to prevent every host in one group of IP addresses from accessing any host in another group of IP addresses. Given the limitations of a simple address/mask syntax specification, a separate rule must be written for each pair in the two (or more) groups. This observation is used in an optimization of the basic algorithm, described later in Section 5.1.

7. We found that 15% of the rules were redundant. A rule R is said to be redundant if one of the following conditions hold (here, we think of a rule R as the set of all packet headers which could match R): (a) There exists a rule T appearing earlier than R in the classifier such that R is a subset of T. Thus, no packet will ever match R, i.e., R is redundant. We call this *backward redundancy* — 7.8% of the rules were found to be backward redundant. (b) There exists a rule T appearing after R in the classifier such that (i) R is a subset of T, (ii) R and T have the same actions, and (iii) For each rule V appearing in between R and T in the classifier, either V is disjoint from R, or V has the same action as R. We call this *forward redundancy* — 7.2% of the rules were forward redundant. In this case, R can be eliminated to obtain a new smaller classifier. A packet matching R in the original classifier will match T in the new classifier, but will yield the same action.

3.3 Observations about the structure of the classifiers

To illustrate the structure we found in our dataset, we start with an example 2-dimensional classifier containing three rules. Figure 4.14(a) shows three such rectangles, where each rectangle represents a rule with a range of values in each dimension. The classifier contains three explicitly defined rules, and the default rule (represented by the background



(a) 4 regions

(b) 5 regions

(c) 7 regions

Figure 4.14 Some possible arrangements of three rectangles (2-dimensional rules). Each differently shaded rectangle comprises one region. The total number of regions indicated includes the white background region.

rectangle). The arrangement of the three rules in Figure 4.14(a) is such that four distinct regions, differently shaded, are created (including the white background region). A different arrangement could create five regions, as in Figure 4.14(b), or seven regions, as in Figure 4.14(c). A classification algorithm must keep a record of each region and be able to determine the region to which each newly arriving packet belongs. Intuitively, the larger the number of regions that the classifier contains, the more storage is required, and the longer it takes to classify a packet.

Even though the number of rules is the same in each of the three cases in Figure 4.14, the task of the classification algorithm becomes progressively harder as it needs to distinguish more regions. In general, it can be shown that the number of regions created by N rules in d dimensions can be $O(N^d)$. Such a worst case example for two dimensions is shown in Figure 4.15.

We analyzed the structure in our dataset and found that the number of overlapping regions is considerably smaller than the worst case. Specifically, for the biggest classifier with 1733 rules, the number of distinct overlapping regions in four dimensions was found to be 4316, compared to approximately 10^{11} regions for the worst possible combination of rules. Similarly, the number of overlapping regions was found to be relatively small in each of the classifiers in the dataset. This is because rules originate from specific policies

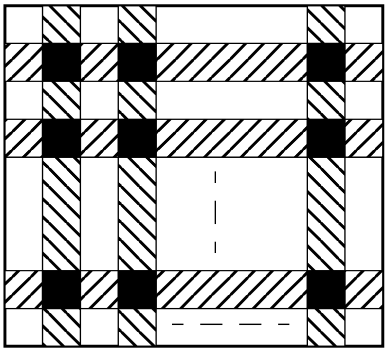


Figure 4.15 A worst case arrangement of N rectangles. $N/2$ rectangles span the first dimension, and the remaining $N/2$ rectangles span the second dimension. Each of the black squares is a distinct region. The total number of distinct regions is therefore $N^2/4 + N + 1 = O(N^2)$.

of network operators and agreements between different networks. For example, the operators of two different networks may specify several policies relating to the interaction of the hosts in one network with the hosts in the other. This implies that rules tend to be clustered in small groups instead of being randomly distributed. As we will see, the proposed algorithm exploits this structure to simplify its task.

3.4 The RFC algorithm

Classifying a packet can be viewed as mapping S bits in the packet header to T bits of *classID* (an identifier denoting the rule, or action), where $T = \log N$, $T \ll S$, in a manner dictated by the N classifier rules. A simple and fast, but unrealistic, way of doing this mapping might be to precompute the value of classID for each of the 2^S different packet header values. This would yield the answer in one step (i.e., one memory access) but would require too much memory. The main aim of RFC is to perform the same mapping but over several stages. As shown in Figure 4.16, RFC performs this mapping recursively — in each stage the algorithm performs a *reduction*, mapping one set of values to a smaller set.

The RFC algorithm has P *phases*, where each phase consists of a set of parallel memory lookups. Each lookup is a reduction in the sense that the value returned by the memory

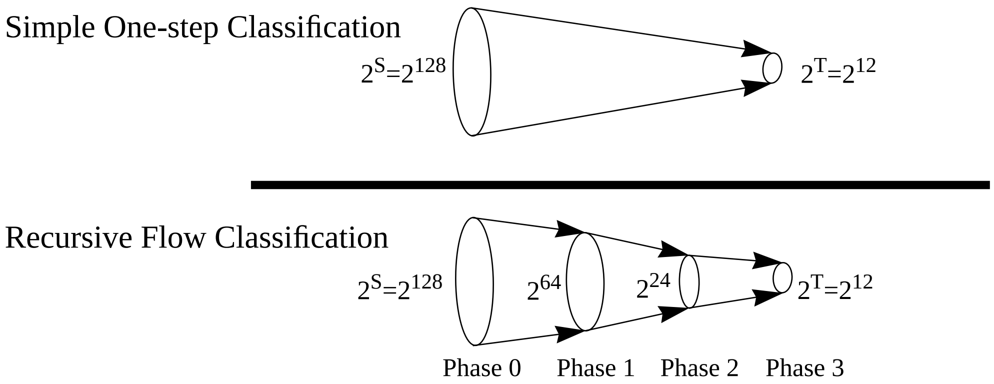


Figure 4.16 Showing the basic idea of Recursive Flow Classification. The reduction is carried out in multiple phases, with a reduction in phase *I* being carried out recursively on the image of the phase *I-1*. The example shows the mapping of 2^S bits to 2^T bits in 4 phases.

lookup is shorter (is expressed in fewer bits) than the index of the memory access. The algorithm, as illustrated in Figure 4.17, operates as follows:

1. In the first phase (phase 0), d fields of the packet header are split up into multiple chunks that are used to index into multiple memories in parallel. For example, the number of chunks equals 8 in Figure 4.17. Figure 4.18 shows an example of how the fields of a packet may be split into chunks. Each of the parallel lookups yields an output value that we will call *eqID*. (The reason for calling this identifier *eqID* will become clear shortly). The contents of each memory are chosen so that the result of the lookup is narrower than the index, i.e., requires fewer bits.
2. In subsequent phases, the index into each memory is formed by combining the results of the lookups from earlier phases. For example, the results from the lookups may be concatenated to form a wider index — we will consider another way to combine them later.
3. After successive combination and reduction, we are left with one result from the memory lookup in the final phase. Because of the way the memory contents have been precomputed, this value corresponds to the classID of the packet.

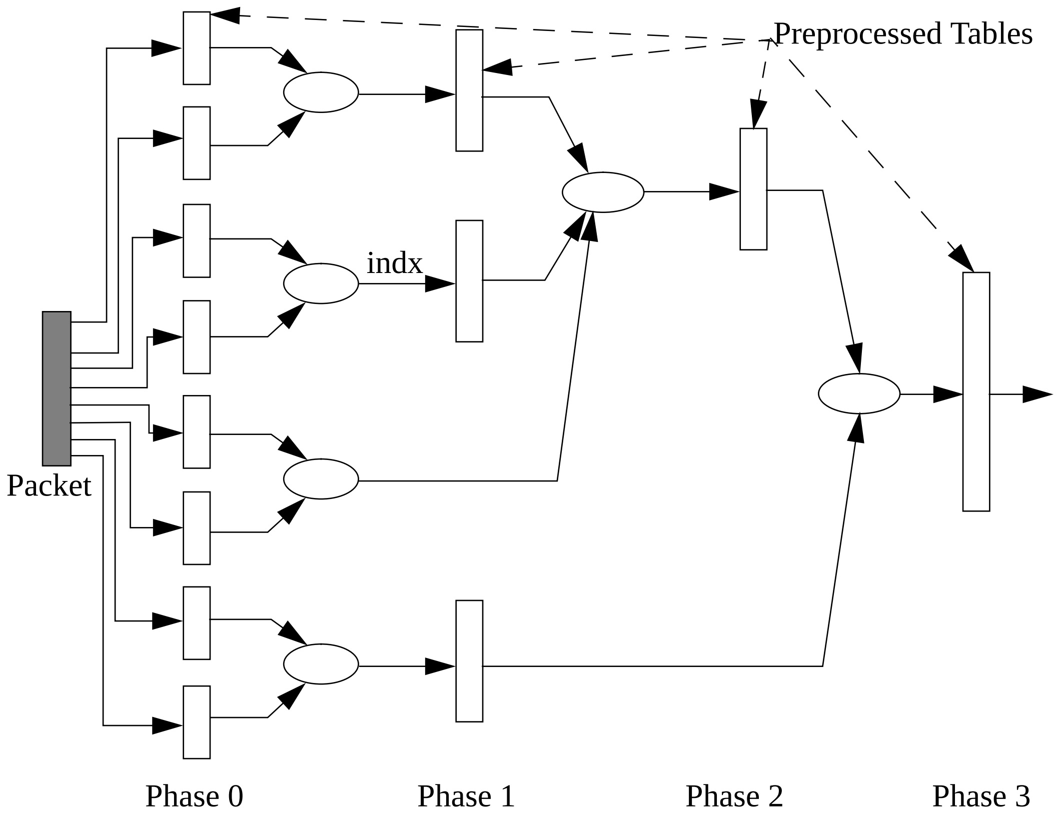


Figure 4.17 Packet flow in RFC.

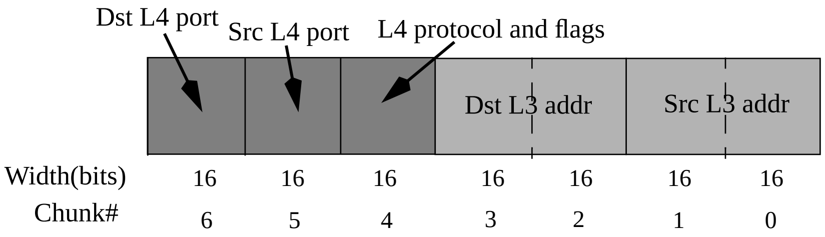


Figure 4.18 Example chopping of the packet header into chunks for the first RFC phase. L3 and L4 refer to the network-layer and transport-layer fields respectively.

For the above scheme to work, the contents of each memory are filled after suitably preprocessing the classifier. To illustrate how the memories are populated, we consider a simple example based on the classifier in Table 4.4.

TABLE 4.4. An example 4-dimensional classifier.

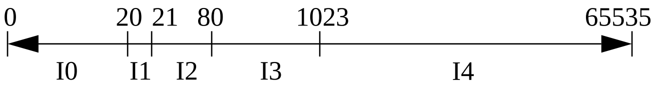
Dst L3 (value/mask)	Src L3 (value/mask)	Dst L4	L4 protocol
152.163.190.69/ 255.255.255.0	152.163.80.1/ 255.255.255.255	*	*
152.168.3.0/ 255.255.255.0	152.163.200.157/ 255.255.255.255	eq http	udp
152.168.3.0/ 255.255.255.0	152.163.200.157/ 255.255.255.255	range 20-21	udp
152.168.3.0/ 255.255.255.0	152.163.200.157/ 255.255.255.255	eq http	tcp
152.168.3.198.4/ 255.255.255.255	152.163.160.0/ 255.255.252.0	gt 1023	tcp
152.163.198.4/ 255.255.255.255	152.163.36.0/ 255.255.255.0	gt 1023	tcp

We will see how the 24 bits used to express the two chunks: chunk #4 (L4, i.e., transport-layer protocol) and chunk #6 (Dst L4, i.e, transport-layer destination) are reduced to just three bits by Phases 0 and 1 of the RFC algorithm. We start with chunk #6, which contains the 16-bit transport-layer destination address. The column corresponding to the transport-layer field in Table 4.4 partitions the set of all possible chunk values into four sets: (a) {20, 21} (b) {http (=80)} (c) {>1023} (d) {all remaining numbers in the range 0-65535}. The four sets can be encoded using two bits 00 through 11. We call these two bit values the *equivalence class IDs* (*eqIDs*) of the respective sets. The memory corresponding to chunk #6, in Phase 0, is indexed using the 2^{16} different values of 16-bit wide chunk #6. In each memory location m, we place the *eqID* for the set containing the value m. For example, the value in the memory location 20 is 00, denoting the set {20,21}. In this way, a 16-bit to 2-bit reduction is obtained for chunk #6 in Phase 0. Similarly, the column corresponding to 8-bit transport-layer protocol in Table 4.4 consists of three sets: (a) {tcp} (b) {udp} (c) {all remaining protocol values in the range 0-255} — which can be encoded using two-bit *eqID*s. Hence, chunk #4 undergoes an eight-bit to two-bit reduction in Phase 0.

In the second phase (Phase 1), we consider the combination of the transport-layer Destination and protocol chunks. Table 4.4 shows that the five sets corresponding to the combination of these chunks are: (a) {({80}, {udp})} (b) {({20-21}, {udp})} (c) {({80}, {tcp})} (d) {({gt 1023}, {tcp})} (e) {all remaining crossproducts of the two columns}. The five sets can be represented using 3-bit *eqID*s. The index into the memory in Phase 1 is constructed by concatenating the two 2-bit *eqIDs* from Phase 0. Hence, Phase 1 reduces the number of bits from four to three. If we now consider the combination of both Phase 0 and Phase 1, we find that 24 bits have been reduced to just 3 bits. Hence, the RFC algorithm uses successive combination and reduction to map the long original packet header to a short classID.

We will now see how a classifier is preprocessed to generate the values to be stored in the memory tables at each phase. In what follows, we will use the term *Chunk Equivalence Set* (CES) to denote a set mentioned in the example above, e.g., each of the three sets: (a) {tcp} (b) {udp} (c) {all remaining protocol values in the range 0-255} is said to be a Chunk Equivalence Set because if there are two packets with different protocol values lying in the same set (and having otherwise identical headers), the rules of the classifier do not distinguish between them. Each CES can be constructed in the following manner.

First phase (Phase 0): The process of constructing a CES in a single dimension is similar to the procedure mentioned earlier for constructing non-overlapping basic intervals from the projections of the rules onto this dimension. The difference lies in that two non-contiguous ranges may now form a part of the same CES. Consider a fixed chunk of size b bits, and those component(s) of the rules in the classifier corresponding to this



E0 = {20,21}　E2 = {1024-65535}
E1 = {80}　E3 = {0-19,22-79,81-1023}

Figure 4.19 An example of computing the four equivalence classes E0...E3 for chunk #6 (corresponding to the 16-bit transport-layer destination port number) in the classifier of Table 4.4.

chunk. Project the rules in the classifier on the number line $[0,2^{\upsilon}-1]$. Each component projects to a set of (not necessarily contiguous) intervals on the number line. The end points of all the intervals projected by these components form a set of non-overlapping intervals. Two points in the same interval always belong to the same equivalence set. Also, two intervals are in the same equivalence set if exactly the same rules project onto them. As an example, consider chunk #6 (destination L4 port) of the classifier in Table 4.4. The intervals, $I0\dots I4$, and the constructed equivalence sets, $E0\dots E3$ are shown in Figure 4.19. The RFC table kept in the memory for this chunk is filled with the corresponding *eqID*s. Thus, in this example, *table[20]* $=00$, *table[23]* $=11$, etc. The pseudocode for computing the *eqID*s in Phase 0 is shown in Figure 4.20.

To facilitate the calculation of *eqIDs* for subsequent RFC phases, we assign a *class bitmap* (CBM) for each CES. The CBM has one bit for each rule in the classifier, and indicates those rules that contain the corresponding CES. For example, E0 in Figure 4.19 will have the CBM 101000, indicating that the first and the third rules of the classifier in Table 4.4 contain E0 in chunk #6. Note that the class bitmap is *not* physically stored in the RFC table: it is just used to facilitate the calculation of the stored *eqIDs* by the preprocessing algorithm.

Subsequent phases: A chunk in a subsequent phase is formed by a combination of two (or more) chunks obtained from memory lookups in previous phases. If, for example, the resulting chunk is of width b bits, we again create equivalence sets such that two b-bit

```
/* Phase 0, Chunk j of width b bits*/
for each rule rl in the classifier
begin
    project the $i^{th}$ component of rl onto the number line $[0,2^b - 1]$, marking the start and end points of
    each of its constituent intervals.
endfor
/* Now scan through the number line looking for distinct equivalence classes */
bmp := 0; /* all bits of bmp are initialised to ‘0’ */
for n in 0..$2^b$-1
    begin
    if (any rule starts or ends at n)
    begin
        update bmp;
        if (bmp not seen earlier)
        begin
            eq := new_equivalence_class();
            eq->cbm := bmp;
        endif
    endif
    else eq := the equivalence class whose cbm is bmp;
table_0_j[n] = eq->ID; /* fill ID in the rfc table*/
endfor
```

Figure 4.20 Pseudocode for RFC preprocessing for chunk j of Phase 0.

packet header values that are not distinguished by the rules of the classifier belong to the same CES. Hence, (20,udp) and (21,udp) will be in the same CES in the classifier of Table 4.4 in Phase 1. The new equivalence sets for a phase are determined by computing all possible intersections of equivalence sets from the previous phases being combined. Each distinct intersection is an equivalence set for the newly created chunk. The pseudocode for this preprocessing is shown in Figure 4.20.

3.5 A simple complete example of RFC

Realizing that the preprocessing steps are involved, we present a complete example of RFC operation on a classifier, showing how preprocessing is performed to determine the

```
    /* Assume that chunk i is formed by combining m distinct chunks c1, c2, ..., cm of phases p1,p2, ...,
pm where p1, p2, ..., pm < j */
    indx := 0; /* indx runs through all the entries of the RFC table, table_j_i */
    listEqs := nil;
    for each CES, c1eq, of chunk c1
    for each CES, c2eq, of chunk c2
    ........
    for each CES, cmeq of chunk cm
    begin
     intersectedBmp := c1eq->cbm & c2eq->cbm & ... & cmeq->cbm;/* bitwise ANDing */
     neweq := searchList(listEqs, intersectedBmp);
     if (not found in listEqs)
     begin
      /* create a new equivalence class */
      neweq := new_Equivalence_Class();
      neweq->cbm := bmp;
      add neweq to listEqs;
     endif
     /* Fill up the relevant RFC table contents.*/
     table_j_i[indx] := neweq->ID;
     indx++;
    endfor
```

Figure 4.21 Pseudocode for RFC preprocessing for chunk i of Phase j.

contents of the memories, and how a packet is looked up. The example is based on a 4-field classifier of Table 4.5 and is shown in Figure 4.22.

TABLE 4.5. The 4-dimensional classifier used in Figure 4.22.

Rule#	Chunk#0 (Src L3 bits 31..16)	Chunk#1 (Src L3 bits 15..0)	Chunk#2 (Dst L3 bits 31..16)	Chunk#3 (Dst L3 bits 15..0)	Chunk#4 (L4 protocol) [8 bits]	Chunk#5 (Dstn L4) [16 bits]
R0	0.83/0.0	0.77/0.0	0.0/0.0	4.6/0.0	udp (17)	*
R1	0.83/0.0	1.0/0.255	0.0/0.0	4.6/0.0	udp	range 20 30
R2	0.83/0.0	0.77/0.0	0.0/255.255	0.0/255.255	*	21
R3	0.0/255.255	0.0/255.255	0.0/255.255	0.0/255.255	*	21
R4	0.0/255.255	0.0/255.255	0.0/255.255	0.0/255.255	*	*

4 Performance of RFC

In this section, we look at the performance obtained by the RFC algorithm on the classifiers in our dataset. First, we consider the storage requirements of RFC. Then we consider its performance to determine the rate at which packets can be classified.

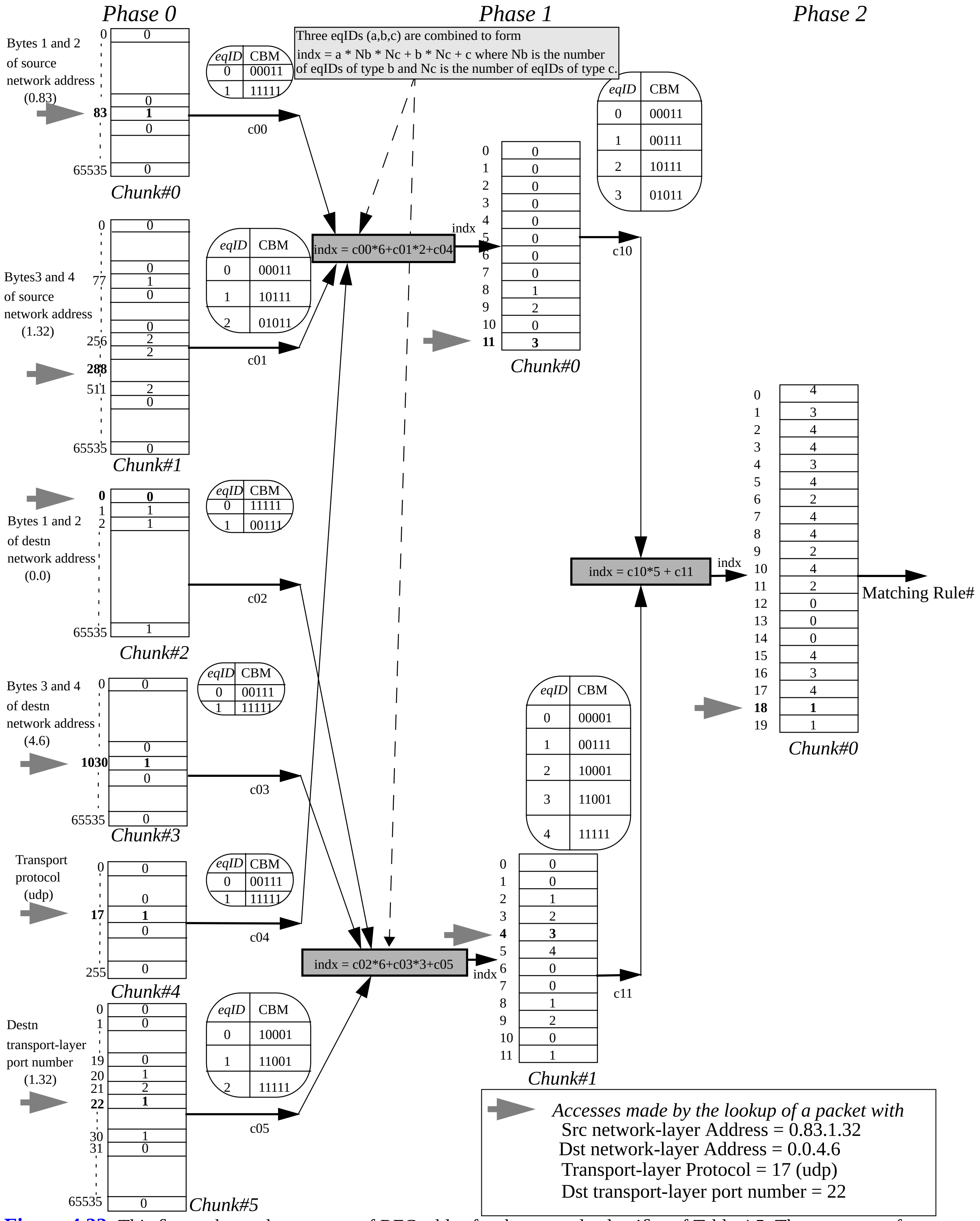

Phase 0
Phase 1
Phase 2
Bytes 1 and 2 of source network address (0.83)
Chunk#0
eqID CBM
0 00011
1 11111
c00
Three eqIDs (a,b,c) are combined to form indx = a * Nb * Nc + b * Nc + c where Nb is the number of eqIDs of type b and Nc is the number of eqIDs of type c.
Bytes3 and 4 of source network address (1.32)
Chunk#1
eqID CBM
0 00011
1 10111
2 01011
c01
indx = c00*6+c01*2+c04
indx
c10
Bytes 1 and 2 of destn network address (0.0)
Chunk#2
eqID CBM
0 11111
1 00111
c02
Bytes 3 and 4 of destn network address (4.6)
Chunk#3
eqID CBM
0 00111
1 11111
c03
Transport protocol (udp)
Chunk#4
eqID CBM
0 00111
1 11111
c04
Destn transport-layer port number (1.32)
Chunk#5
eqID CBM
0 10001
1 11001
2 11111
c05
indx = c02*6+c03*3+c05
c11
eqID CBM
0 00011
1 00111
2 10111
3 01011
Chunk#0
eqID CBM
0 00001
1 00111
2 10001
3 11001
4 11111
Chunk#1
indx = c10*5 + c11
Matching Rule#
Chunk#0
Accesses made by the lookup of a packet with
Src network-layer Address = 0.83.1.32
Dst network-layer Address = 0.0.4.6
Transport-layer Protocol = 17 (udp)
Dst transport-layer port number = 22

Figure 4.22 This figure shows the contents of RFC tables for the example classifier of Table 4.5. The sequence of accesses made by the example packet have also been shown using big gray arrows. The memory locations accessed in this sequence have been marked in bold.

4.1 RFC preprocessing

As our dataset has a maximum of four fields, the chunks for Phase 0 are created as shown in Table 4.6.

TABLE 4.6. Packet header fields corresponding to chunks for RFC Phase 0.

Chunk#	Field (subfield)
0	Source L3 address (most significant 16-bits)
1	Source L3 address (least significant 16-bits)
2	Destination L3 address (most significant 16-bits)
3	Destination L3 address (most significant 16-bits)
4	L4 protocol and flags
5	L4 destination port number

The performance of RFC (storage requirements and classification time) can be tuned with two parameters: (i) The number of phases, P, and (ii) The reduction tree used for a given P. For instance, two of the several possible reduction trees for $P = 3$ and $P = 4$ are shown in Figure 4.23 and Figure 4.24 respectively. (For $P = 2$, there is only one reduction tree possible.) When there is more than one reduction tree possible for a given value of P, the algorithm chooses a tree based on two heuristics: (i) Given a classifier, the maximum amount of pruning of the search space is likely to be obtained by combining those chunks together which have the most “correlation.” As an example, the combination of chunk 0 (most significant 16 bits of the source network address) and chunk 1 (least significant 16 bits of the source network address) in the toy example of Figure 4.22 would result in only 3 *eqID*s, while the combination of chunk 0 and chunk 4 (destination transport port number) would result in 5 *eqID*s. (ii) The algorithm combines as many chunks as it can without causing unreasonable memory consumption. Following these heuristics, we find that the “best” reduction tree for $P = 3$ is *tree_B* in Figure 4.23, and the “best” reduction tree for $P = 4$ is *tree_A* in Figure 4.24.[1]

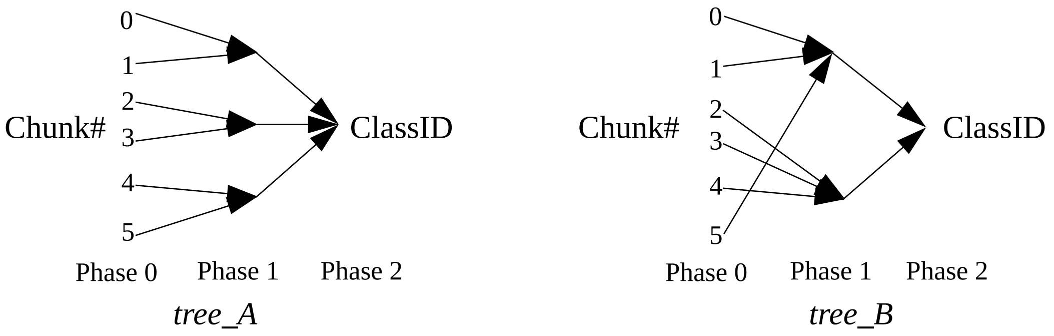


Figure 4.23 Two example reduction trees for three phases in RFC.

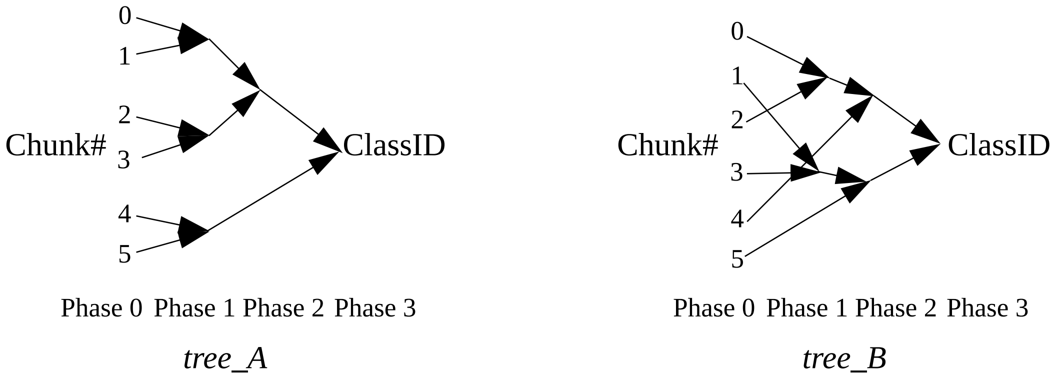


Figure 4.24 Two example reduction trees for four phases in RFC.

We now look at the performance of RFC on our dataset. Our first goal is to keep the total storage consumption small. The storage requirements for each of our classifiers is plotted in Figure 4.25, Figure 4.26 and Figure 4.27 for 2, 3 and 4 phases respectively.The graphs show how memory usage increases with the number of rules in each classifier. For practical purposes, it is assumed that memory is only available in widths of 8, 12 or 16 bits. Hence, an *eqID* requiring 13 bits is assumed to occupy 16 bits in the RFC table.

As we might expect, the graphs show that storage requirements decrease with an increase in the number of phases from three to four. However, this comes at the expense of

[1] These reduction trees gave better performance results over other trees for a vast majority of the classifiers in our experiments.

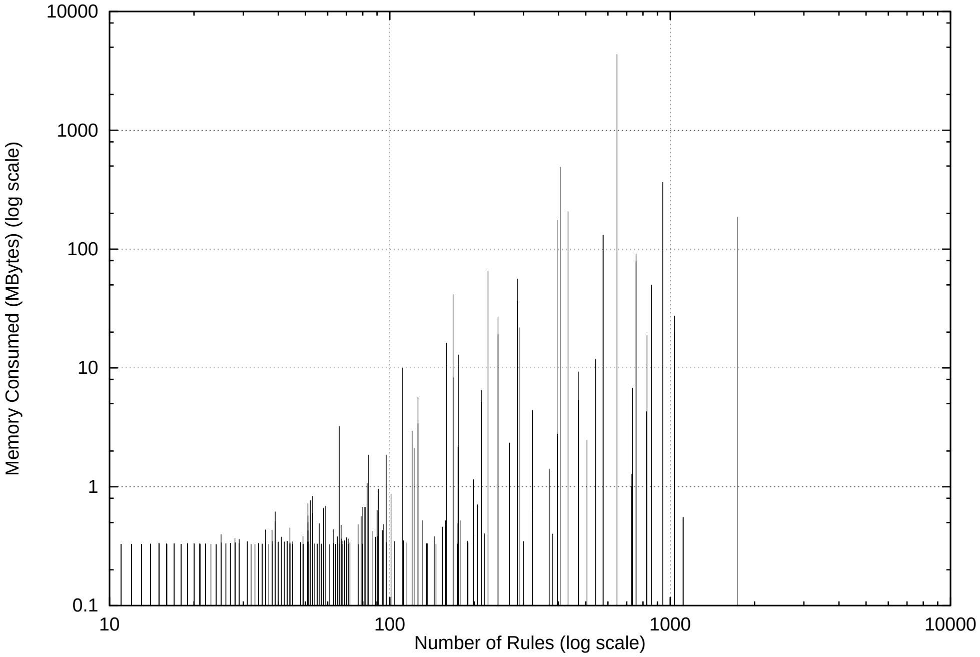


Figure 4.25 The RFC storage requirement in Megabytes for two phases using the dataset. This special case of RFC with two phases is identical to the Crossproducting method of [95].

two additional memory accesses, illustrating the trade-off between memory consumption and lookup time in RFC.

Like most algorithms in the literature, RFC does not support quick incremental updates, and may require rebuilding the data structure in the worst case. It turns out, however, that rebuilding is only necessary in the case of the addition of a new rule. Deletion of existing rules can be simply handled by changing the chunk equivalence sets of *eqID*s in the final phase. The performance of an implementation of such an incremental delete algorithm on random deletes is shown in Figure 4.28.

Our second goal is to keep the preprocessing time small — this is useful when updates necessitate rebuilding the data structure. Figure 4.29 plots the preprocessing time required for both three and four phases of RFC.[1] These graphs indicate that, if the data structure is

[1] The case P=2 is not plotted: it was found to take hours of preprocessing time because of the unwieldy size of the RFC tables.

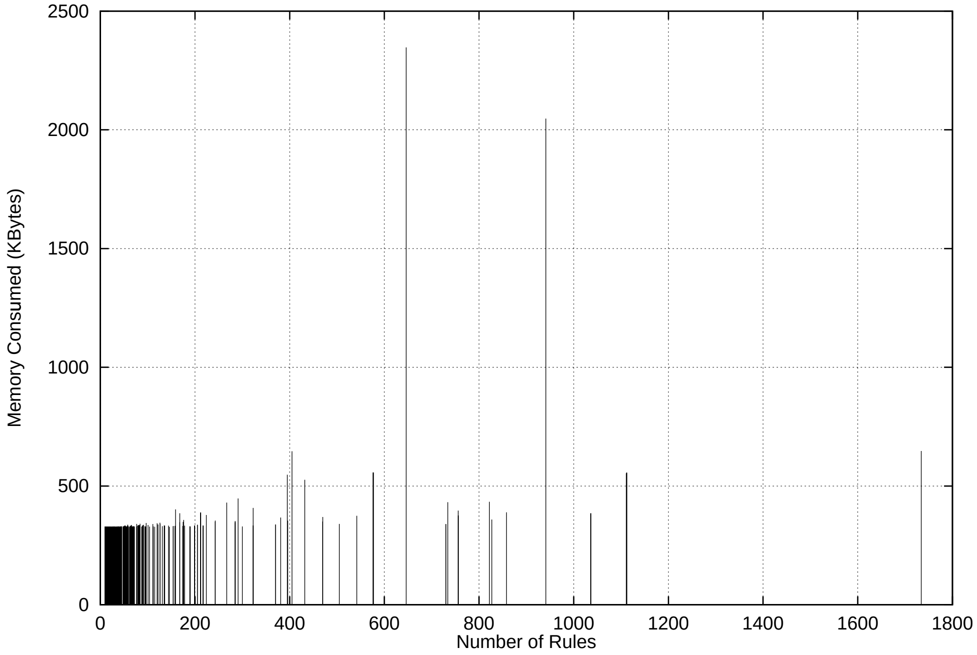


Figure 4.26 The RFC storage requirement in Kilobytes for three phases using the dataset. The reduction tree used is *tree_B* in Figure 4.23.

rebuilt on the addition of every rule, RFC may be suitable if (and only if) the rules change relatively slowly — for example, not more than once every few seconds. Thus, RFC may be suitable in environments where rules are changed infrequently; for example, if they are added manually, or on a router reboot.

Finally, note that there are some similarities between the RFC algorithm and the bitmap-intersection scheme of [48]; each distinct bitmap in [48] corresponds to a CES in the RFC algorithm. Also, note that when there are just two phases, RFC corresponds to the crossproducting method described in [95]. RFC is different from both these schemes in that it generalizes the concept of crossproducting to make storage requirements feasible for larger classifiers, along with a lookup time that scales better than that of the bitmap-intersection approach.

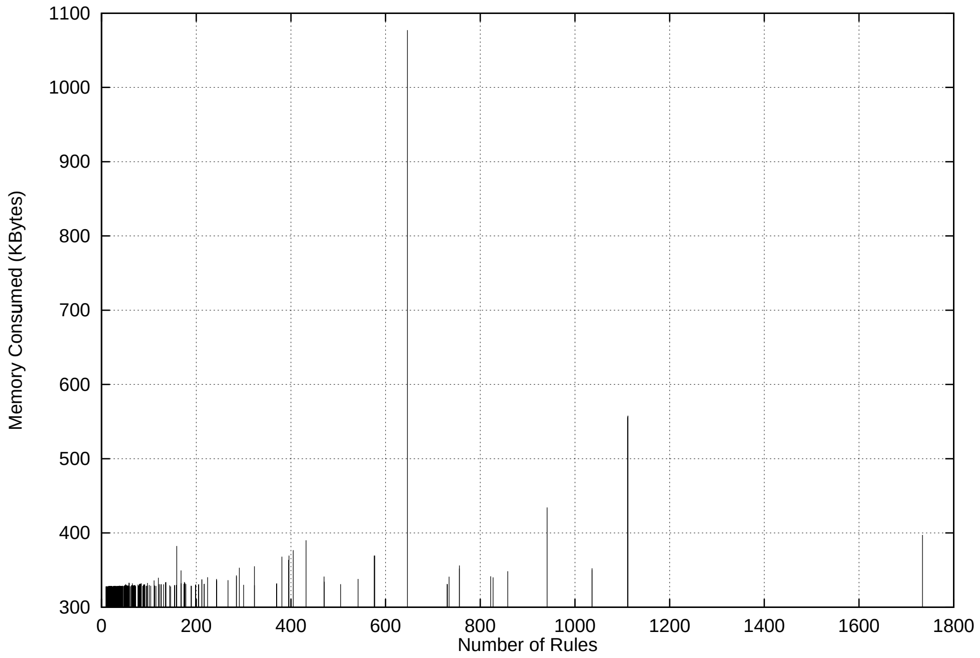


Figure 4.27 The RFC storage requirement in Kilobytes for four phases using the dataset. The reduction tree used is *tree_A* in Figure 4.24.

4.2 RFC lookup performance

The RFC lookup operation can be performed in hardware or in software.[1] We will discuss the lookup performance in each case separately.

4.2.1 Lookups in hardware

An example hardware implementation for the tree *tree_B* in Figure 4.23 (three phases) is illustrated in Figure 4.30 for four fields (six chunks in Phase 0). This design is suitable for all the classifiers in our dataset, and uses two 4 Mbit SRAMs and two 4-bank 64 Mbit SDRAMs clocked at 125 MHz.[2] The design is pipelined such that a new lookup may begin every four clock cycles.

[1] Note that preprocessing is always performed in software.

[2] These devices are in production in industry at the time of writing. In fact, even bigger and faster devices are available at the time of writing — see for example, reference [137].

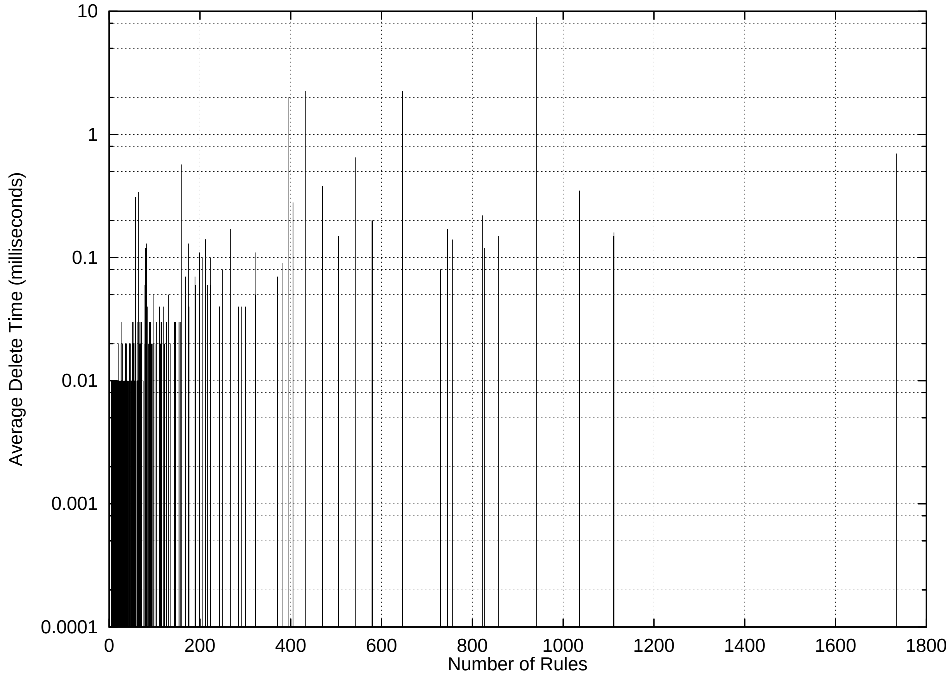


Figure 4.28 This graph shows the average amount of time taken by the incremental delete algorithm in milliseconds on the classifiers available to us. Rules deleted were chosen randomly from the classifier. The average is taken over 10,000 delete operations, and although not shown, variance was found to be less than 1% for all experiments. This data is taken on a 333 MHz Pentium-II PC running the Linux operating system.

The pipelined RFC lookup proceeds as follows:

1. **Pipeline Stage 0: Phase 0** *(Clock cycles 1-4)*: In the first three clock cycles, three accesses are made to the two SRAM devices in parallel to yield the six *eqID*s of Phase 0. In the fourth clock cycle, the *eqID*s from Phase 0 are combined to compute the two indices for the next phase.
2. **Pipeline Stage 1: Phase 1***(Clock cycles 5-8)*: The SDRAM devices can be accessed every two clock cycles, but we assume that a given bank can be accessed again only after eight clock cycles. By keeping the two memories for Phase 1 in different banks of the SDRAM, we can perform the Phase 1 lookups in four clock cycles. The data is replicated in the other two banks (i.e. two banks of memory hold a fully redundant copy of the lookup tables for Phase 1). This allows Phase 1 lookups to be performed on the next packet as soon as the current packet has completed. In this way, any given bank is accessed once every eight clock cycles.

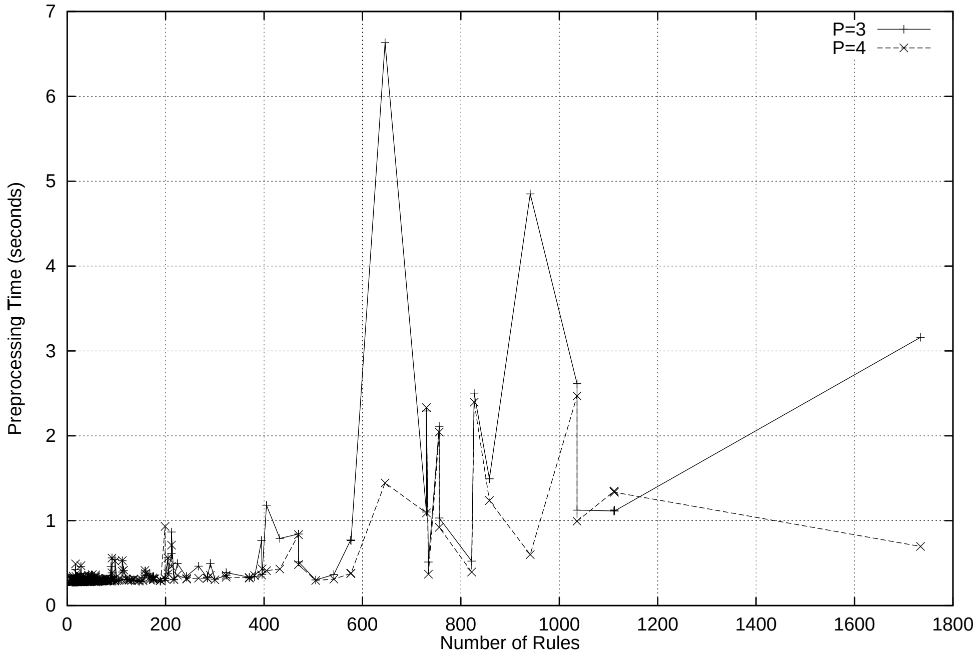


Figure 4.29 The preprocessing times for three and four phases in seconds, using the set of classifiers available to us. This data is taken by running the RFC preprocessing code on a 333 MHz Pentium-II PC running the Linux operating system.

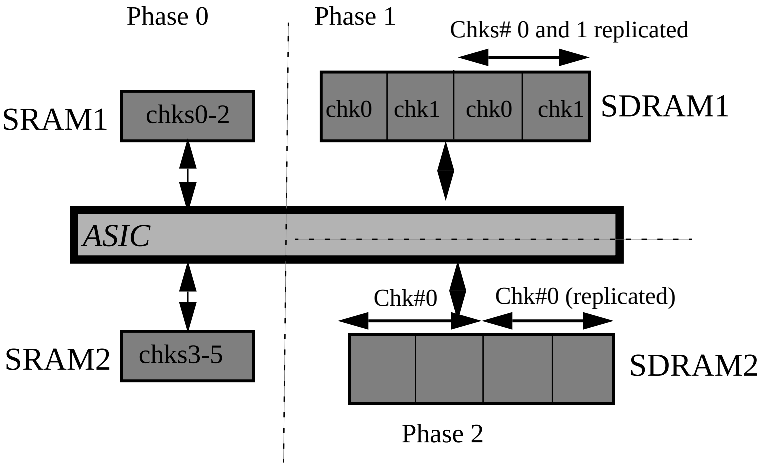


Figure 4.30 An example hardware design for RFC with three phases. The registers for holding data in the pipeline and the on-chip control logic are not shown. This design achieves OC192c rates in the worst case for 40 byte packets. The phases are pipelined with 4 clock cycles (at 125 MHz clock rate) per pipeline stage.

3. **Pipeline Stage 2: Phase 2** *(Clock cycles 9-12)*: Only one lookup is to be made. The operation is otherwise identical to Phase 1.

This design classifies approximately 30 million packets per second (to be exact, 31.25 million packets per second with a 125 MHz clock) with a total memory cost of approximately $40.[1] This is fast enough to process minimum length TCP/IP packets at OC192 rates.

Discussion of how RFC exploits the structure in real-life classifiers

We saw in Section 3.3 that rules in real-life classifiers form a small number of overlapping regions and tend to cluster in small groups. The idea behind the *reduction* steps used in the RFC algorithm is to quickly narrow down the large search space to smaller subspaces containing these clusters. In order to do this without consuming too much storage, the reduction is carried out on small-sized chunks. However, the whole packet header needs to be looked at in order to prune the search space completely to arrive at the best matching rule — this is the purpose of the *combination* steps used in the RFC algorithm that incrementally combine a few chunks at a time till the whole packet header has been considered. Because the rules form a small number of overlapping regions, combining results of the reduction steps creates chunks that are still small enough to keep the total storage requirements reasonable.

Discussion of hardware implementation of RFC

We have seen that lower bounds to the multi-field packet classification problem imply that any solution will be either too slow, or will consume a large amount of storage in the worst case. Given that it is difficult to design hardware around an engine with unpredictable speed, RFC takes the approach of ensuring bounded worst-case classification time.

[1] At the time of writing, SDRAMs are available at approximately $1.0 per megabyte, and SRAMs at $12 for a 4 Mbit device running at 125 MHz [119][129].

```
/* pktFields[i] stores the value of field i in the packet header */
for (each chunk numbered chkNum of phase 0)
  eqNums[0][chkNum] = contents of appropriate rfc table at memory location pktFields[chkNum];
for (phaseNum = 1..numPhases-1)
for (each chunk numbered chkNum in Phase phaseNum)
begin
 /* chd stores the number and description about this chunk's parents chkParents[0..numChkParents-1]*/
 chd = parent descriptor of (phaseNum, chkNum);
 indx = eqNums[phaseNum of chkParents[0]][chkNum of chkParents[0]];
 for (i=1..chd->numChkParents-1)
   begin
       indx = indx * (total #eqIDs of chd->chkParents[i]) + eqNums[phaseNum of chd->chkParents[i]][chkNum of chd->chkParents[i]];
      /*** Alternatively: indx = (indx << (#bits of equivID of chd->chkParents[i])) ^ (eqNums[phaseNum of chkParents[i]][chkNum of chkParents[i]] ***/
   endfor
 eqNums[phaseNum][chkNum] = contents of appropriate rfc table at address indx
endfor
endfor
return (eqNums[numPhases-1][0]); /* this contains the desired classID */
```

Figure 4.31 Pseudocode for the RFC lookup operation.

This has the side-effect of making it difficult to accurately predict the storage requirements of RFC as a function of the size of the classifier — the performance of RFC is determined by the structure present in the classifier. Even though pathological sets of rules do not seem to appear in practice, RFC storage requirements could scale geometrically with the number of rules in the worst case. This lack of characterization of the precise storage requirements of RFC as a function of only the number of rules in a classifier is a disadvantage to designers implementing RFC in hardware.

4.2.2 Lookups in software

Figure 4.31 shows the pseudocode to perform RFC lookups. When written in 'C,' approximately 30 lines of code are required to implement RFC. When compiled on a 333 MHz Pentium-II PC running Windows NT, we found that the worst case path for the code took $(140clk + 9t_m)$ time for three phases, and $(146clk + 11t_m)$ for four phases, where

t_m is the memory access time, and *clk* equals 3 ns.[1] With $t_m = 60ns$, this corresponds to $0.96\mu s$ and $1.1\mu s$ for three and four phases respectively. This implies that RFC can classify close to one million packets per second in the worst case for this dataset. The average lookup time was found to be approximately 50% faster than the worst case — Table 4.7 shows the average time taken per packet classification for 100,000 randomly generated packets for some classifiers in the dataset.

TABLE 4.7. Average time to classify a packet using a software implementation of RFC.

Number of rules in classifier	Average time per classification (ns)
39	587
113	582
646	668
827	611
1112	733
1733	621

The pseudocode in Figure 4.31 calculates the indices into each memory using multiplication/addition operations on *eqIDs* from previous phases. Alternatively, the indices can be computed by simple concatenation. This has the effect of increasing the memory consumed because the tables do not remain as tightly packed.[2] Given the simpler processing, we might expect the classification time to decrease at the expense of increased memory usage. Indeed the memory consumed grows approximately by a factor of two for the classifiers we have considered. Surprisingly, we saw no significant reduction in classification times. We believe that this is because the processing time is dominated by memory access time as opposed to the CPU cycle time.

[1] The performance of the lookup code was analyzed using VTune [138], an Intel performance analyzer for processors of the Pentium family.

[2] Not packing rfc tables in memories may in fact be desirable to accomodate newly added rules in the classifier.

4.3 Larger classifiers

To estimate how RFC might perform with future, larger classifiers, we synthesized large artificial classifiers. We used two different ways to create large classifiers (given the importance of the structure, it did not seem meaningful to generate rules randomly):

1. A large classifier can be created by concatenating classifiers for different services, but belonging to the same network, into a single classifier. This is actually desirable in scenarios where only one set of RFC tables is desired for the whole network. In such cases, the classID obtained would have to be combined with some other information (such as the classifier ID) to obtain the correct intended action. By only concatenating classifiers from the same network, we were able to create classifiers such that the biggest classifier had 3896 rules. For each classifier created, we measured the storage requirements of RFC with both three and four phases. This is shown in Figure 4.32.
2. To create even larger classifiers, we concatenated all the classifiers of a few (up to ten) different networks. The performance of RFC with four phases is plotted as the ‘Basic RFC’ curve in Figure 4.35. We found that RFC frequently runs into storage problems for classifiers with more than 6000 rules. Employing more phases does not help as we must combine at least two chunks in every phase, and finish with one chunk in the final phase.[1] An alternative way to process large classifiers would be to split them into two (or more) parts and construct separate RFC tables for each part. This would of course come at the expense of doubling the number of memory accesses.[2]

5 Variations

Several variations and improvements of RFC are possible. First, it is easy to see how RFC can be extended to process a larger number of fields in each packet header.

Second, we can possibly speed up RFC by taking advantage of fast lookup algorithms that find longest matching prefixes in one field. Note that in our examples, we use three

[1] With six chunks in Phase 0, we could have increased the number of phases to a maximum of six. However we found no appreciable improvement by doing so.

[2] For Phase 0, we need not lookup memory twice for the same chunk if we use wide memories. This would help us access the contents of both the RFC tables in one memory access.

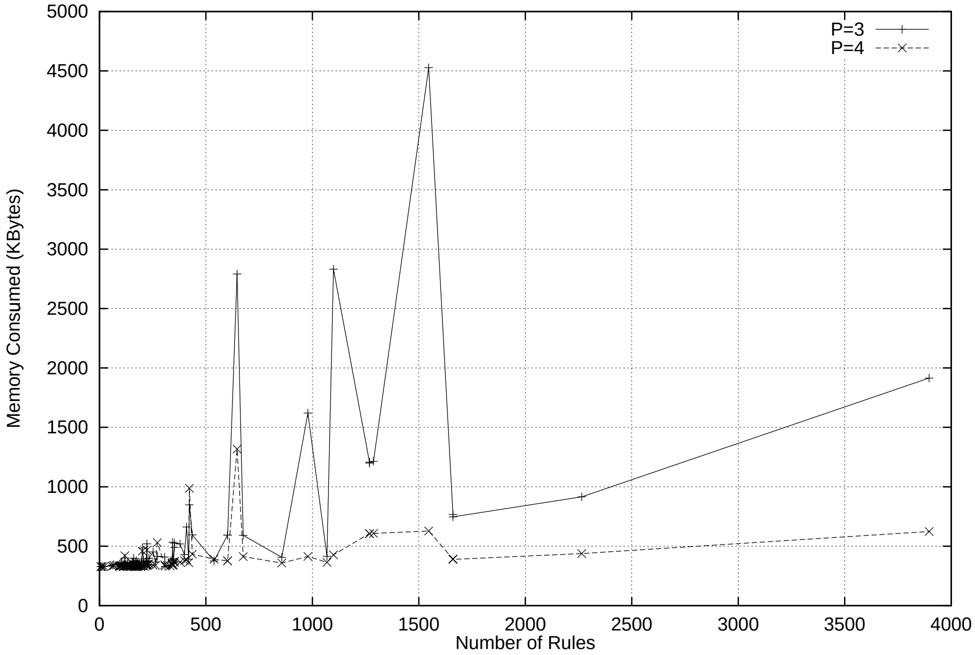


Figure 4.32 The memory consumed by RFC for three and four phases on classifiers created by merging all the classifiers of one network.

memory accesses each for the source and destination network-layer address lookups during the first two phases of RFC. This is necessary because of the large number of non-contiguous address/mask specifications. If only prefixes are allowed in the specification, one can use a more sophisticated and faster algorithm for looking up in one dimension, for instance, one of the algorithms described in Chapter 2.

Third, we can employ the technique described below to decrease the storage requirements for large classifiers.

5.1 Adjacency groups

Since the size of RFC tables depends on the number of chunk equivalence classes, we try to reduce this number by merging two or more rules of the original classifier as explained below. We find that each additional phase of RFC further increases the amount of compaction possible on the original classifier.

First we define some notation. We call two distinct rules R and S, with R appearing first in the classifier, to be *adjacent in dimension* i if all of the following three conditions are satisfied: (1) Both rules have the same action, (2) All but the i^{th} field have the exact same specification in the two rules, and (3) All rules appearing in between R and S in the classifier have either the same action or are disjoint from R (i.e., do not overlap with R). Two rules are simply said to be *adjacent* if they are adjacent in some dimension. Adjacency can also be viewed in the following way: Treat each rule with d fields as a boolean expression of d (multi-valued) variables. Each rule is a conjunction (logical-AND) of these variables. Two rules are now defined to be adjacent if they are adjacent vertices in the d-dimensional hypercube created by the symbolic representation of the d fields.

Example 4.3: For the example classifier of Table 4.8, R2 and R3 are adjacent in the dimension corresponding to the transport-layer Destination field. Similarly R5 is adjacent to R6 (in the dimension network-layer Source), but not to R4 (different actions), or to R7.

TABLE 4.8. An example classifier in four dimensions. The column headings indicate the names of the corresponding fields in the packet header. "*gt N*" in a field specification specifies a value strictly greater than *N*.

Rule	Network-layer destination (address/ mask)	Network-layer source (address/ mask)	Transport-layer destination	Transport -layer protocol	Action
R1	152.163.190.69/ 255.255.255.255	152.163.80.11/ 255.255.255.255	*	*	Deny
R2	152.168.3.0/ 255.255.255.0	152.163.200.157/ 255.255.255.255	eq http	udp	Permit
R3	152.168.3.0/ 255.255.255.0	152.163.200.157/ 255.255.255.255	range 20-21	udp	Permit
R4	152.168.3.0/ 255.255.255.0	152.163.200.157/ 255.255.255.255	eq http	tcp	Deny
R5	152.163.198.4/ 255.255.255.255	152.163.161.0/ 255.255.252.0	gt 1023	tcp	Permit
R6	152.163.198.4/ 255.255.255.255	152.163.0.0/ 255.255.252.0	gt 1023	tcp	Permit
R7	0.0.0.0/0.0.0.0	0.0.0.0/0.0.0.0	*	*	Permit

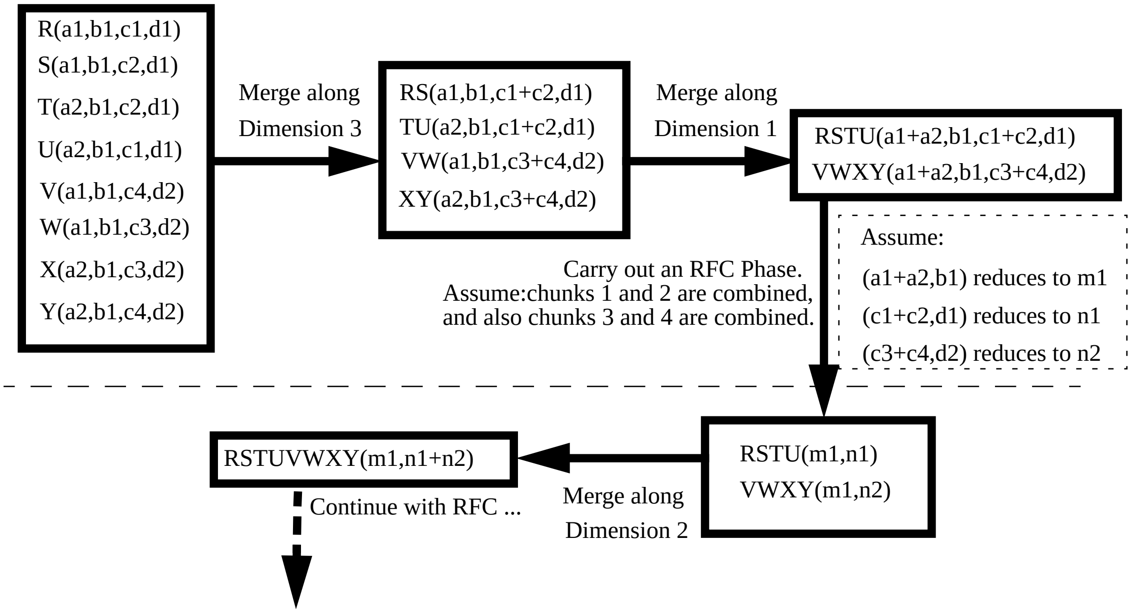


Figure 4.33 This example shows how adjacency groups are formed on a classifier. Each rule is denoted symbolically by *RuleName(value-of-field1, value-of-field2,...)*. All rules shown are assumed to have the same action. '+' denotes a logical OR.

Two rules R and S that are adjacent in dimension i are merged to form a new rule T with the same action as R (or S). T has the same specifications as that of R (or S) for all fields except that of the i^{th}, which is simply the *logical-OR* of the i^{th} field specifications in R and S. The third condition above ensures that the relative priority of the rules in between R and S will not be affected by this merging.

An *adjacency group* is defined recursively as: (1) Every rule in the original classifier is an adjacency group, and (2) Every merged rule that is created by merging two or more adjacency groups is an adjacency group.

The classifier is compacted as follows. Initially, every rule is in its own adjacency group. Next, adjacent rules are combined to create a new smaller classifier. This is implemented by iterating over all fields in turn, checking for adjacency in each dimension. After

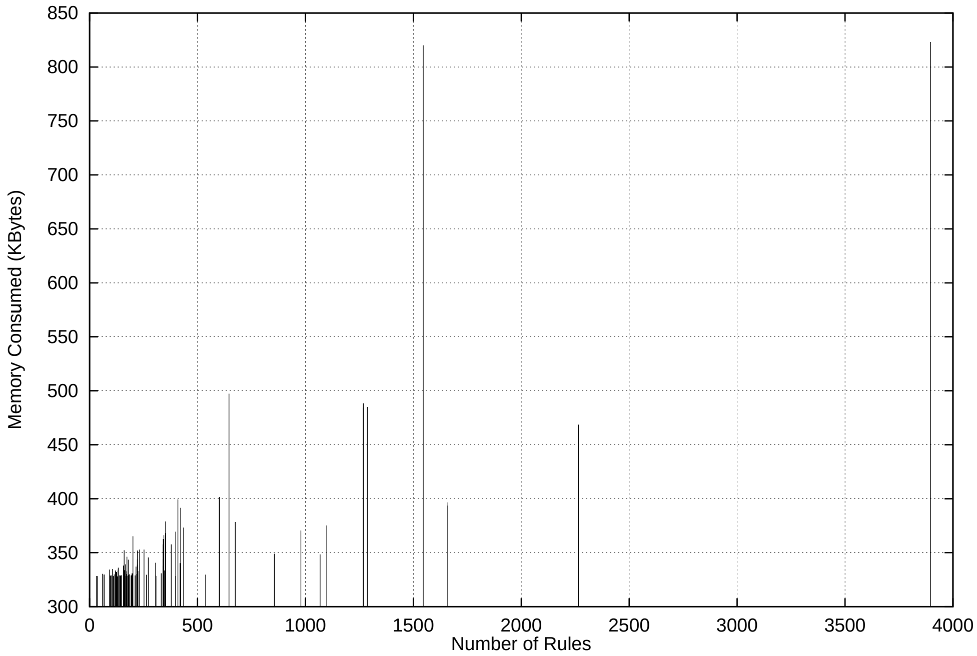


Figure 4.34 The memory consumed by RFC for three phases with the adjacency group optimization enabled on classifiers created by merging all the classifiers of one network. The memory consumed by the basic RFC scheme for the same set of classifiers is plotted in Figure 4.35.

these iterations are completed, the resulting classifier will not have any more adjacent rules. As each RFC phase collapses some dimensions, groups which were not adjacent in earlier phases may become so in later stages. In this way, the number of adjacency groups, and hence the size of the compacted classifier, keeps on decreasing with every phase. An example of this operation is shown in Figure 4.33.

Note that there is no change in the actual lookup operation: the equivalence class identifiers now represent bitmaps which keep track of adjacency groups rather than the original rules. The benefits of the adjacency group optimization are demonstrated in Figure 4.34 (using 3 RFC phases on 101 large classifiers created by concatenating all the classifiers belonging to one network) and in Figure 4.35 (using 4 RFC phases on even larger classifiers created by concatenating all the classifiers of a few different networks together) respectively. With this optimization, the storage requirements of RFC for a 15,000 rule

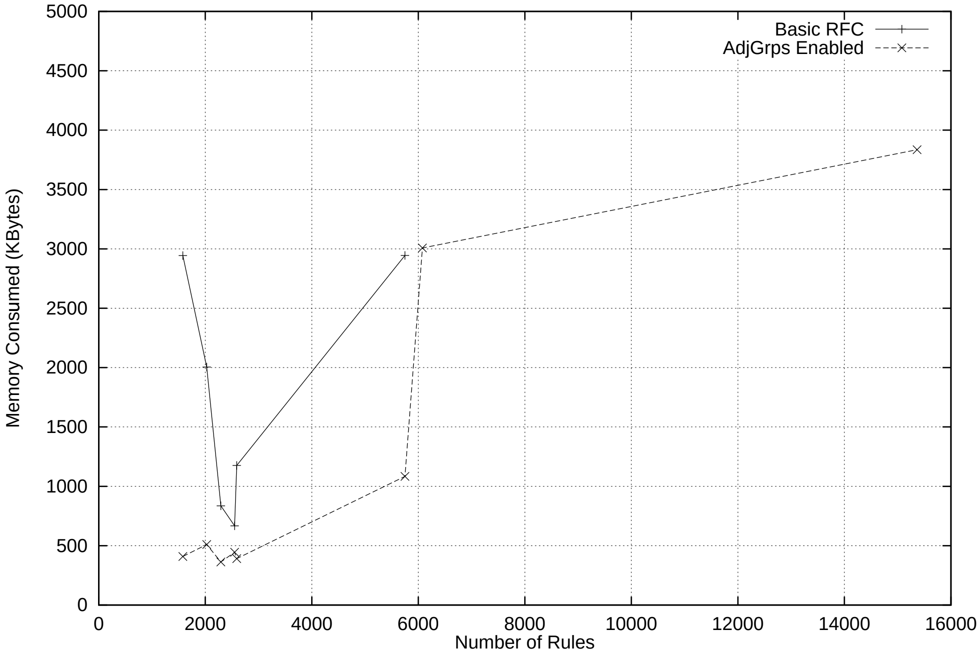


Figure 4.35 The memory consumed with four phases with the adjacency group optimization enabled on the large classifiers created by concatenating all the classifiers of a few different networks. Also shown is the memory consumed when the optimization is not enabled (i.e. the basic RFC scheme). Notice the absence of some points in the "Basic RFC" curve. For those classifiers, the basic RFC scheme takes too much memory/preprocessing time.

classifier decreases to only 3.85 MB. The intuitive reason for the reduction in storage is that several rules in the same classifier commonly share a number of specifications for many fields (an observation mentioned in Section 3.2).

However, the storage space savings come at a cost. Although the classifier will correctly identify the action for each arriving packet, it cannot tell which rule in the original classifier it matched — as the rules have been merged to form adjacency groups, the distinction between each rule has been lost. This may be undesirable in applications that wish to maintain matching statistics for each rule.

TABLE 4.9. A qualitative comparison of some multi-dimensional classification algorithms.

Scheme	Pros	Cons
Sequential evaluation	Good storage and update requirements. Suitable for multiple fields.	High classification time.
Grid-of-tries and FIS-tree	Good storage requirements and fast lookup rates for two fields. Suitable for big 2-dimensional classifiers.	Results in two dimensions do not extend as well to more than two fields. Not suitable for non-contiguous masks.
Crossproducting	Fast accesses. Suitable for multiple fields. Can be adapted to rules with non-contiguous masks.	Large memory requirements. Suitable without caching for small classifiers up to 50 rules.
Bitmap-intersection	Suitable for multiple fields. Can be adapted to rules with non-contiguous masks.	Large memory size and memory bandwidth required. Comparatively slow lookup rate. Hardware only.
Tuple space search	Suitable for multiple fields. Fast average classification and update time.	Non-deterministic and high classification time.
Recursive flow classification	Suitable for multiple fields. Supports rules with non-contiguous masks. Reasonable storage requirements for real-life classifiers. Fast classification.	High preprocessing time and memory requirements for large classifiers (i.e. having more than 6000 rules without adjacency group optimization).

6 Comparison with related work

Table 4.9 shows a qualitative comparison of RFC with previously proposed schemes for doing packet classification.

7 Conclusions and summary of contributions

It is relatively simple to perform packet classification at high speeds using excessively large amounts of storage, or at low speeds with small amounts of storage. When matching multiple fields simultaneously, theoretical bounds show that it is difficult to achieve both high classification rate and modest storage in the worst case. This chapter shows that real classifiers exhibit a considerable amount of structure and redundancy, and introduces for the first time the idea of using simple heuristic algorithms to solve the multi-dimensional packet classification problem.

The contribution of this chapter is the first proposed algorithm, RFC, that deliberately attempts to exploit this structure. RFC appears to perform well with the selection of real-life classifiers available to us. A hardware implementation of RFC can classify minimum-sized IP packets at OC192c rates with commercial memories commonly available today, while a software implementation can classify at OC48c rates. This chapter also shows that while the basic RFC scheme may consume a large amount of storage for large four-field classifiers (with more than 6000 rules), the structure and redundancy in the classifiers can be further exploited with an optimization of the basic RFC scheme. This optimization makes RFC practical for classifiers containing up to approximately 15,000 rules.

CHAPTER 5

Hierarchical Intelligent Cuttings: A Dynamic Multi-dimensional Packet Classification Algorithm

1 Introduction

We saw in the previous chapter that real-life classifiers exhibit structure and redundancy that can be exploited by simple algorithms. One such algorithm RFC, was described in the previous chapter. RFC enables fast classification in multiple dimensions. However, its data structure (reduction tree) has a fixed *shape* independent of the characteristics of the classifier.

This chapter is motivated by the observation that an algorithm capable of adapting its data structure based on the characteristics of the classifier may be better suited for exploiting the structure and redundancy in the classifier. One such classification algorithm, called *HiCuts* (Hierarchical Intelligent Cuttings), is proposed in this chapter.

HiCuts discovers the structure while preprocessing the classifier and adapts its data structure accordingly. The data structure used by HiCuts is a decision tree on the set of rules in the classifier. Classification of a packet is performed by a traversal of this tree followed by a linear search on a bounded number of rules. As computing the optimal decision tree for a given search space is known to be an NP-complete problem [40], HiCuts uses simple heuristics to partition the search space in each dimension.

Configuration parameters of the HiCuts algorithm can be tuned to trade-off query time against storage requirements. On 40 real-life four-dimensional classifiers obtained from ISP and enterprise networks with 100 to 1733 rules,[1] HiCuts requires less than 1 Mbyte of storage. The worst case query time is 12, and the average case query time is 8 memory accesses, plus a linear search on 8 rules. The preprocessing time can be sometimes large — up to a minute[2] — but the time to incrementally update a rule in the data structure is less than 100 milliseconds on average.

1.1 Organization of the chapter

Section 2 describes the data structure built by the HiCuts algorithm, including the heuristics used while preprocessing the classifier. Section 3 discusses the performance of HiCuts on the classifier dataset available to us. Finally, Section 4 concludes with a summary and contributions of this chapter.

[1] The dataset used in this chapter is identical to that in Chapter 4 except that small classifiers having fewer than 100 rules are not considered in this chapter.

[2] Measured using the *time()* lynx system call in user level ‘C’ code on a 333MHz Pentium-II PC, with 96 Mbytes of memory and 512 Kbytes of L2 cache.

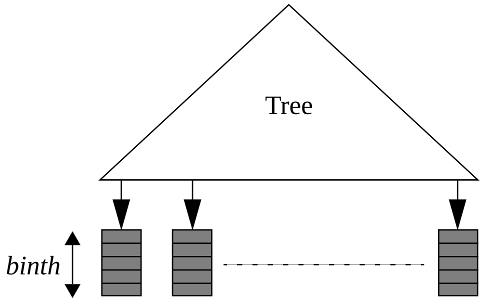


Figure 5.1 This figure shows the tree data structure used by HiCuts. The leaf nodes store a maximum of *binth* classification rules.

2 The Hierarchical Intelligent Cuttings (HiCuts) algorithm

2.1 Data structure

HiCuts builds a decision tree data structure (shown in Figure 5.1) such that the internal nodes contain suitable information to guide the classification algorithm, and the external nodes (i.e., the leaves of the tree) store a small number of rules. On receiving an incoming packet, the classification algorithm traverses the decision tree to arrive at a leaf node. The algorithm then searches the rules stored at this leaf node sequentially to determine the best matching rule. The tree is constructed such that the total number of rules stored in a leaf node is bounded by a small number, which we call *binth* (for 'bin-threshold'). The shape characteristics of the decision tree — such as its depth, the degree of each node, and the local search decision to be made by the query algorithm at each node — are chosen while preprocessing the classifier, and depend on the characteristics of the classifier.

Next, we describe the HiCuts algorithm with the help of the following example.

Example 5.1: Table 5.1 shows a classifier in two 8-bit wide dimensions. The same classifier is illustrated geometrically in Figure 5.2. A decision tree is constructed by recursively partitioning[1] the two-dimensional geometric space. This is shown in Figure 5.3 with a binth of 2. The root node of the tree represents the complete two-dimensional space of size $2^8 \times 2^8$. The algorithm partitions this space into four, equal

[1] We will use the terms 'partition' and 'cut' synonymously throughout this chapter.

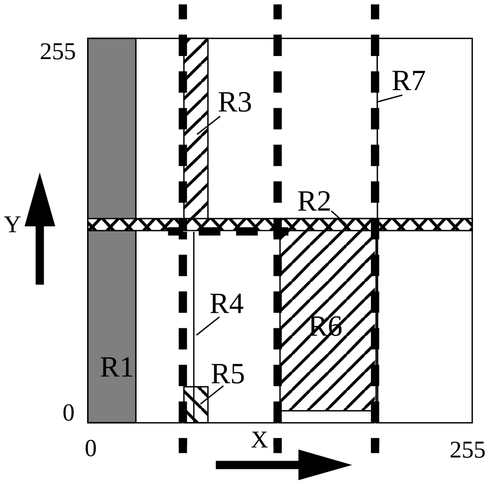


Figure 5.2 An example classifier in two dimensions with seven 8-bit wide rules.

sized geometric subspaces by cutting across the X dimension. The subspaces are represented by each of the four child nodes of the root node. All child nodes, except the node labeled A, have less than or equal to *binth* rules. Hence, the algorithm continues with the tree construction only with node A. The geometric subspace of size $2^6 \times 2^8$ at node A is now partitioned into two equal-sized subspaces by a cut across dimension Y. This results in two child nodes, each of which have two rules stored in them. That completes the construction of the decision tree, since all leaf nodes of this tree have less than or equal to *binth* rules

TABLE 5.1. An example 2-dimensional classifier.

Rule	Xrange	Yrange
R1	0-31	0-255
R2	0-255	128-131
R3	64-71	128-255
R4	67-67	0-127
R5	64-71	0-15
R6	128-191	4-131
R7	192-192	0-255

We can now generalize the description of the HiCuts algorithm in d dimensions as follows. Each internal node, v, of the tree represents a portion of the geometric search space

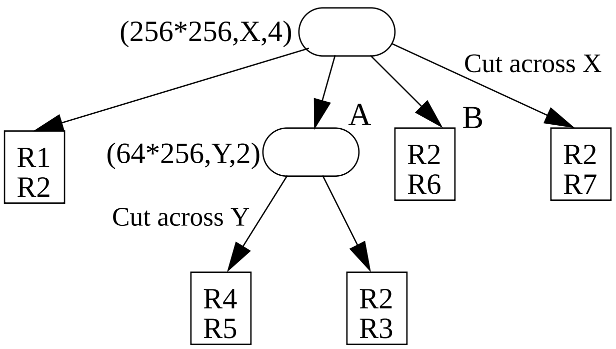


Figure 5.3 A possible HiCuts tree with binth = 2 for the example classifier in Figure 5.2. Each ellipse denotes an internal node v with a tuple $\langle B(v), dim(C(v)), np(C(v))\rangle$. Each square is a leaf node which contains the actual classifier rules.

— for example, the root node represents the complete geometric space in d dimensions. The geometric space at node v is partitioned into smaller geometric subspaces by cutting across one of the d dimensions. These subspaces are represented by each of the child nodes of v. The subspaces are recursively partitioned until a subspace has no more than *binth* number of rules — in which case, the rules are allocated to a leaf node.

More formally, we associate the following entities with each internal node v of the HiCuts data structure for a d-dimensional classifier:

- A hyperrectangle $B(v)$, which is a d-tuple of ranges (i.e., intervals): $([l_1,r_1]$, $[l_2,r_2]$, ..., $[l_d,r_d])$. This rectangle defines the geometric subspace stored at v.
- A cut $C(v)$, defined by two entities. (1) $k = dim(C(v))$, the dimension across which $B(v)$ is partitioned. (2) $np(C(v))$, the number of partitions of $B(v)$, i.e., the number of cuts in the interval $[l_d,r_d]$. Hence, the cut, $C(v)$, divides $B(v)$ into smaller rectangles which are then associated with the child nodes of v.
- A set of rules, $CRS(v)$. If v is a child of w, then $CRS(v)$ is defined to be the subset of $CRS(w)$ that 'collides' with $B(v)$, i.e., every rule in $CRS(w)$ that spans, cuts or is contained in $B(v)$ is also a member of $CRS(v)$. $CRS(root)$ is the set of all

rules in the classifier. We call $CRS(v)$ the colliding rule set of v, and denote the number of rules in $CRS(v)$ by $NumRules(v)$.

As an example of two W-bit wide dimensions, the root node represents a rectangle of size $2^W \times 2^W$. The cuttings are made by axis-parallel hyperplanes, which are simply lines in the case of two dimensions. The cut C of a rectangle B is described by the number of equal-sized intervals created by partitioning one of the two dimensions. For example, if the algorithm decides to cut the root node across the first dimension into D intervals, the root node will have D children, each with a rectangle of size $\left(2^W / D\right) \times 2^W$ associated with it.

2.2 Heuristics for decision tree computation

There are many ways to construct a decision tree for a given classifier. During preprocessing, the HiCuts algorithm uses the following heuristics based on the structure present in the classifier:

1. A heuristic that chooses a suitable number of interval cuts, $np(C)$, to make at an internal node. A large value of $np(C)$ will decrease the depth of the tree, hence accelerating query time at the expense of increased storage. To balance this tradeoff, the preprocessing algorithm follows a heuristic that is guided and tuned by a pre-determined space measure function $spmf()$. For example, the definition $spmf(n) = spfac \times n$, where *spfac* is a constant, is used in the experimental results in Section 3. We also define a *space measure* for a cut $C(v)$ as: $sm(C(v)) = \sum_{i=1}^{np(C(v))} NumRules(child_i) + np(C(v))$, where $child_j$ denotes the j^{th} child node of node v. HiCuts makes as many cuttings as $spmf()$ allows at a certain node, depending on the number of rules at that node. For instance, $np(C(v))$ could be chosen to be the largest value (using a simple binary search) such that $sm(C(v)) < spmf(NumRules(v))$. The pseudocode for such a search algorithm is shown in Figure 5.4.
2. A heuristic that chooses the dimension to cut across, at each internal node. For example, it can be seen from Figure 5.2 that cutting across the Y-axis would be

```
/* Algorithm to do binary search on the number of cuts to be made at node v. When the number of cuts are
such that the corresponding storage space estimate becomes more than what is allowed by the spacemea-
sure function spmf(), we end the search. Note that it is possible to do smarter variations of this search algo-
rithm.*/
n = numRules(v);
nump = max(4, sqrt(n)); /* arbitrary starting value of number of partitions to make at this node */
for (done=0;done == 0;)
{
sm(C) = 0;
  for each rule r in R(v)
     { sm(C) += number of partitions colliding with rule r; }
  sm(C) += nump;
  if (sm(C) < spmf(n))
  {
     nump = nump * 2; /* increase the number of partitions in a binary search fashion */
  }
  else { done = 1;}
}
/* The algorithm has now found a value of nump (the number of children of this node) that fits the storage
requirements */
```

Figure 5.4 Pseudocode for algorithm to choose the number of cuts to be made at node v.

less beneficial than cutting across the X-axis at the root node. There are various methods to choose a good dimension: *(a)* Minimizing $max_j(NumRules(child_j))$ in an attempt to decrease the worst-case depth of the tree. *(b)* Treating $\left\{NumRules(child_j)/\left(\sum_{i=1}^{np(C(v))} NumRules\,(child_i)\right)\right\}$ as a probability distribution with $np(C)$ elements, and maximizing the entropy of the distribution. Intuitively, this attempts to pick a dimension that leads to the most uniform distribution of rules across nodes, so as to create a balanced decision tree. *(c)* Minimizing $sm(C)$ over all dimensions. *(d)* Choosing the dimension that has the largest number of distinct components of rules. For instance, in the classifier of Table 5.1, rules $R3$ and $R5$ share the same rule component in the X dimension. Hence, there are 6 components in the X dimension and 7 components in the Y dimension. The use of this heuristic would dictate a cut across dimension Y for this classifier.

3. A heuristic that maximizes the reuse of child nodes. We have observed in our experiments that in real classifiers, many child nodes have identical colliding rule sets. Hence, a single child node can be used for each distinct colliding rule set,

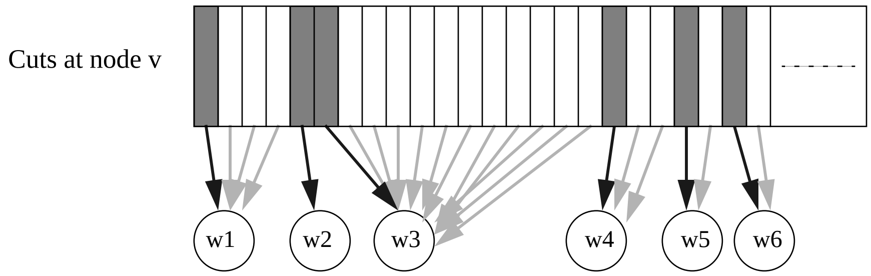


Figure 5.5 An example of the heuristic maximizing the reuse of child nodes. The gray regions correspond to children with distinct colliding rule sets.

while other child nodes with identical rule sets can simply point to this node. Figure 5.5 illustrates this heuristic.

4. A heuristic that eliminates redundancies in the colliding rule set of a node. As a result of the partitioning of a node, rules may become redundant in some of the child nodes. For example, in the classifier of Table 5.1, if $R6$ were higher priority than $R2$, then $R2$ would be made redundant by $R6$ in the third child of the root node labeled B (see Figure 5.3). Detecting and eliminating these redundant rules can decrease the data structure storage requirements at the expense of increased preprocessing time. In the experiments described in the next section, we invoked this heuristic when the number of rules at a node fell below a threshold.

3 Performance of HiCuts

We built a simple software environment to measure the performance of the HiCuts algorithm. Our dataset consists of 40 classifiers containing between 100 and 1733 rules from real ISP and enterprise networks. For each classifier, a data structure is built using the heuristics described in the previous section. The preprocessing algorithm is tuned by two parameters: (1) *binth*, and (2) *spfac* — used in the space measure function *spmf()*, defined as $spmf(n) = spfac \times n$.

Figure 5.6 shows the total data structure storage requirements for $binth = 8$ and $spfac = 4$. As shown, the maximum storage requirement for any classifier is approximately 1 Mbyte, while the second highest value is less than 500 Kbytes. These small stor-

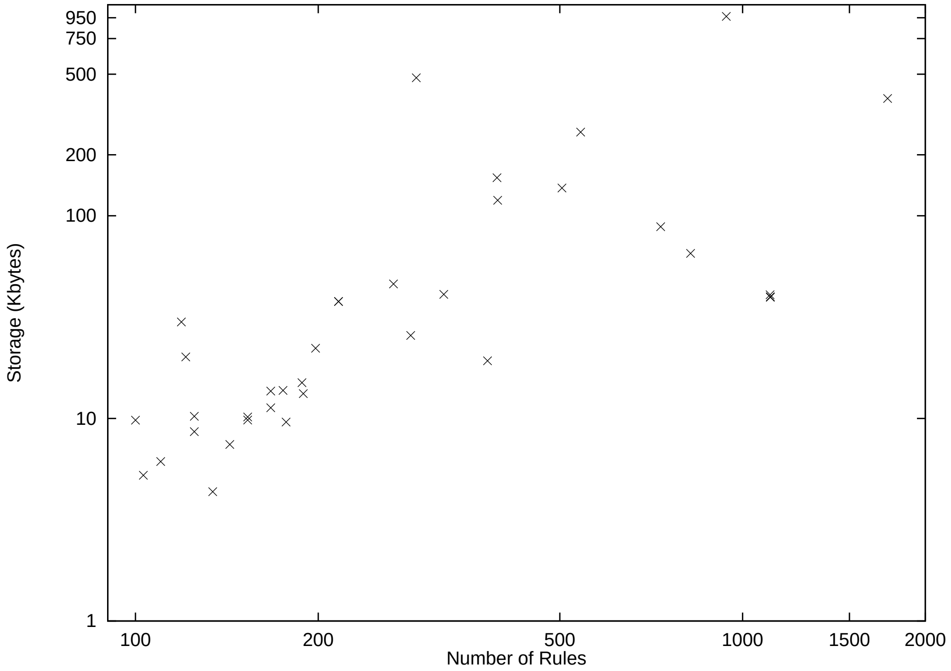


Figure 5.6 Storage requirements for four dimensional classifiers for binth=8 and spfac=4.

age requirements imply that in a software implementation of the HiCuts algorithm, the whole data structure would readily fit in the L2 cache of most general purpose processors.

For the same parameters ($binth = 8$ and $spfac = 4$), Figure 5.7 shows the maximum and average tree depth for the classifiers in the dataset. The average tree depth is calculated under the assumption that each leaf is accessed in proportion to the number of rules stored in it. As shown in Figure 5.7, the worst case tree depth for any classifier is 12, while the average is approximately 8. This implies that — in the worst case — a total of 12 memory accesses are required, followed by a linear search on 8 rules to complete the classification. Hence, a total of 20 memory accesses are required in the worst case for these parameters.

The preprocessing time required to build the decision tree is plotted in Figure 5.8. This figure shows that the highest preprocessing time is 50.5 seconds, while the next highest

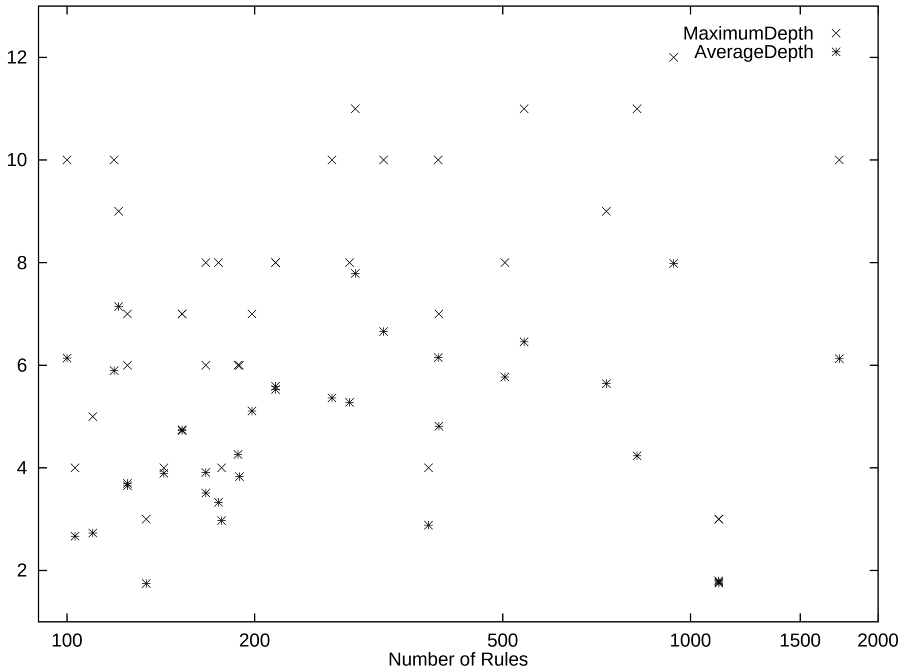


Figure 5.7 Average and worst case tree depth for binth=8 and spfac=4.

value is approximately 20 seconds. All but four classifiers have a preprocessing time of less than 8 seconds.

The reason for the fairly large preprocessing time is mainly the number and complexity of the heuristics used in the HiCuts algorithm. We expect this preprocessing time to be acceptable in most applications, as long as the time taken to incrementally update the tree remains small. In practice, the update time depends on the rule to be inserted or deleted. Simulations indicate an average incremental update time between 1 and 70 milliseconds (averaged over all the rules of a classifier), and a worst case update time of nearly 17 seconds (see Figure 5.9).

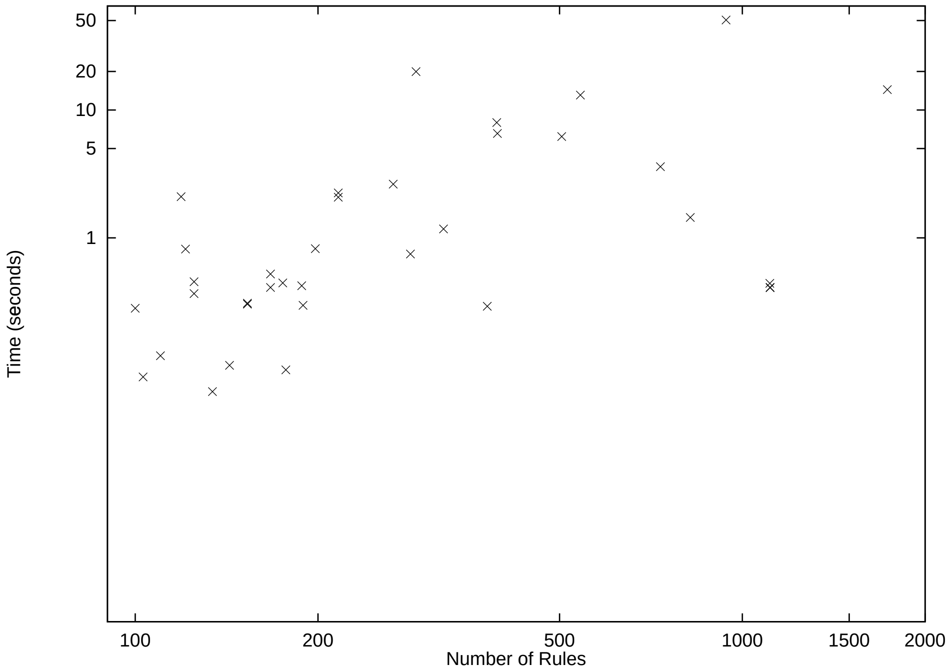


Figure 5.8 Time to preprocess the classifier to build the decision tree. The measurements were taken using the *time()* linux system call in user level 'C' code on a 333 MHz Pentium-II PC with 96 Mbytes of memory and 512 Kbytes of L2 cache.

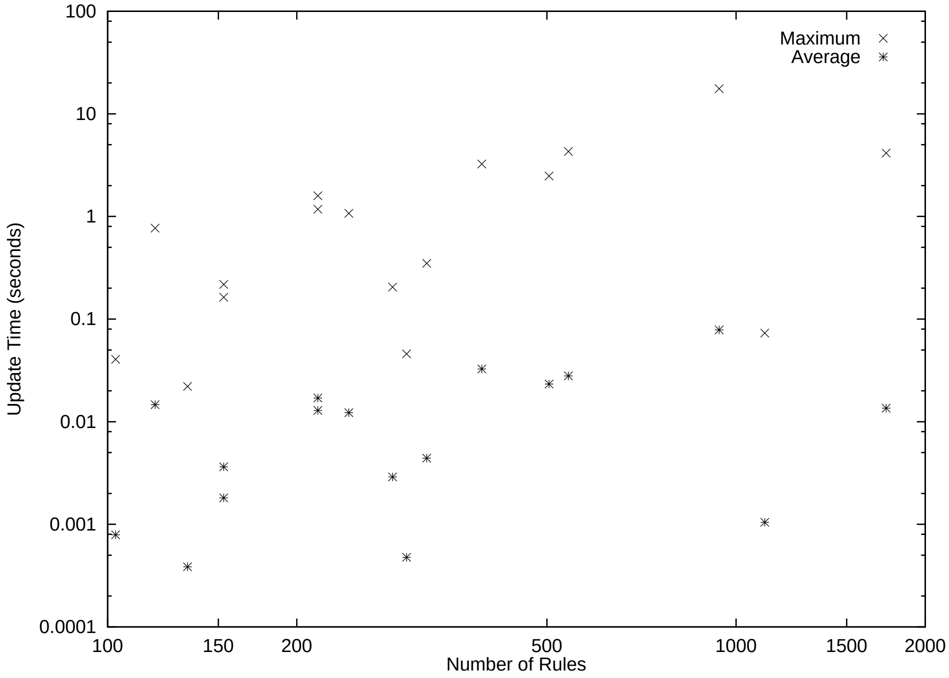


Figure 5.9 The average and maximum update times (averaged over 10,000 inserts and deletes of randomly chosen rules for a classifier). The measurements were taken using the *time()* linux system call in user level 'C' code on a 333 MHz Pentium-II PC with 96 Mbytes of memory and 512 Kbytes of L2 cache.

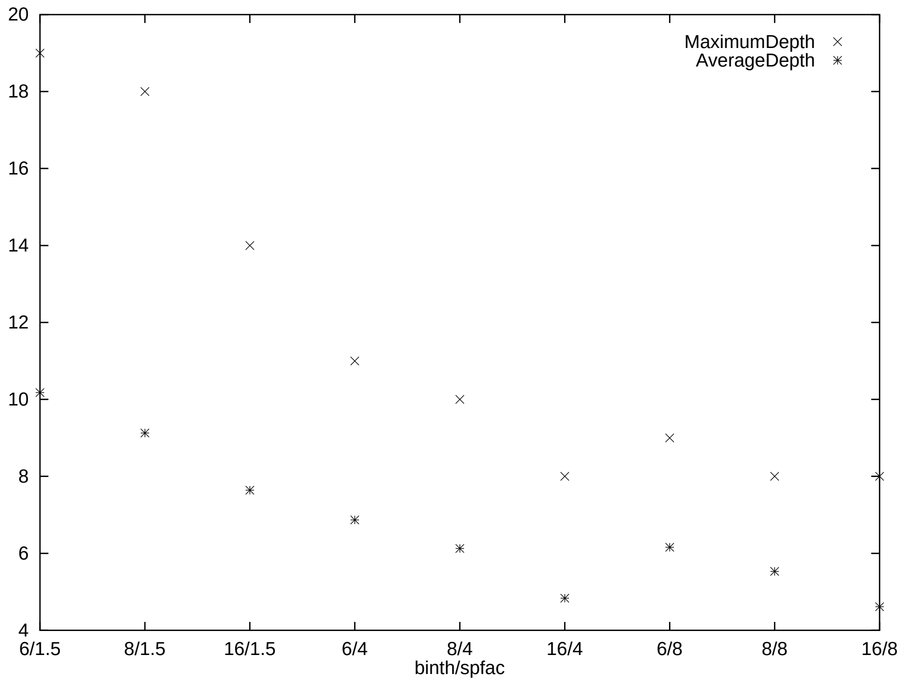


Figure 5.10 Variation of tree depth with parameters binth and spfac for a classifier with 1733 rules.

3.1 Variation with parameters *binth* and *spfac*

Next, we show the effect of varying the configuration parameters *binth* and *spfac* on the data structure for the largest 4-dimensional classifier available to us containing 1733 rules. We carried out a series of experiments where parameter *binth* took the values 6, 8 and 16; and parameter *spfac* took the values 1.5, 4 and 8. We make the following, somewhat expected, observations from our experiments:

1. The HiCuts tree depth is inversely proportional to both *binth* and *spfac. This is* shown in Figure 5.10.
2. As shown in Figure 5.11, the data structure storage requirements are directly proportional to *spfac* but inversely proportional to *binth*.

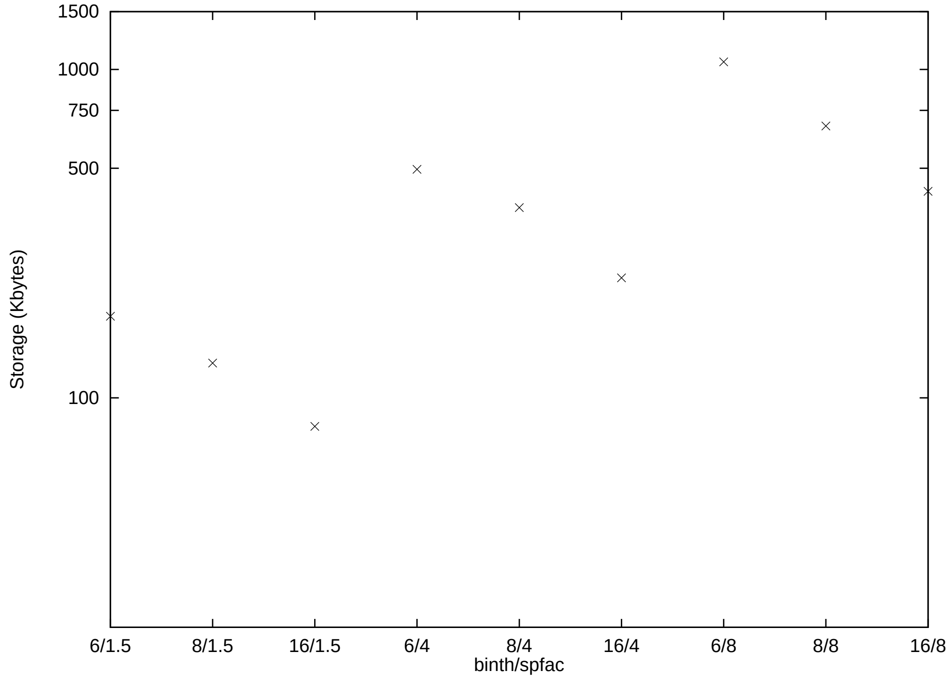


Figure 5.11 Variation of storage requirements with parameters *binth* and *spfac* for a classifier with 1733 rules.

3. The preprocessing time is proportional to the storage requirements, as shown in Figure 5.12.

3.2 Discussion of implementation of HiCuts

Compared with the RFC algorithm described in Chapter 4, the HiCuts algorithm is slower but consumes a smaller amount of storage. As with RFC, it seems difficult to characterize the storage requirements of HiCuts as a function of the number of rules in the classifier. However, given certain design constraints in terms of the maximum available storage space or the maximum available classification time, HiCuts seems to provide greater flexibility in satisfying these constraints by allowing variation of the two parameters, *binth* and *spfac*.

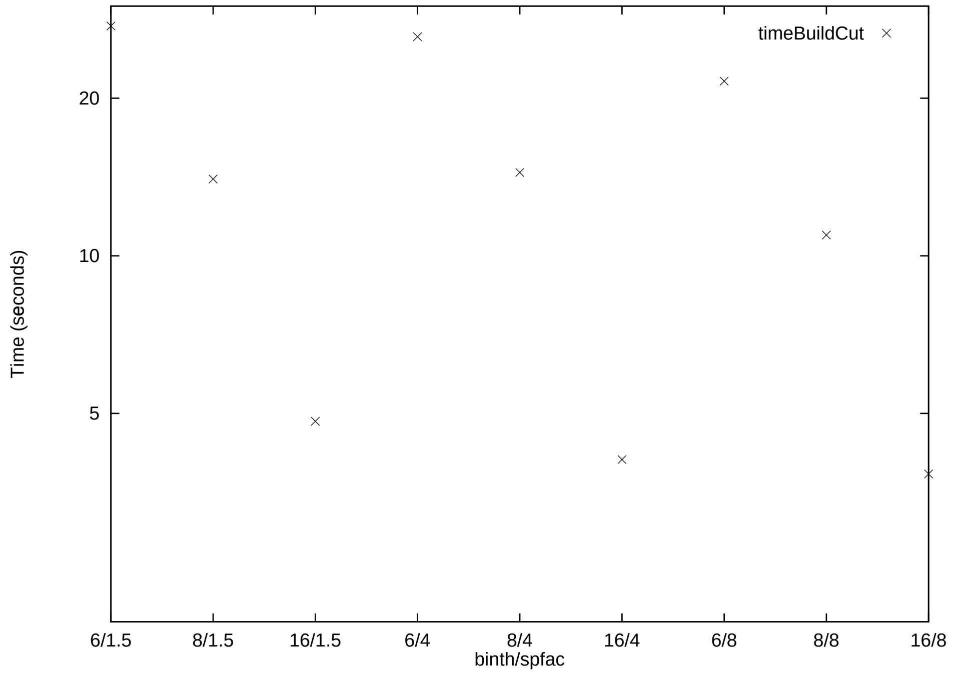


Figure 5.12 Variation of preprocessing times with *binth* and *spfac* for a classifier with 1733 rules. The measurements were taken using the time() linux system call in user level 'C' code on a 333 MHz Pentium-II PC with 96 Mbytes of memory and 512 Kbytes of L2 cache.

4 Conclusions and summary of contributions

The design of multi-field classification algorithms is hampered by worst-case bounds on query time and storage requirements that are so onerous as to make generic algorithms unusable. So instead we must search for characteristics of real-life classifiers that can be exploited in pursuit of fast algorithms that are also space-efficient. Similar to Chapter 4, this chapter resorts to heuristics that, while hopefully well-founded in a solid understanding of today's classifiers, exploit the structure of classifiers to reduce query time and storage requirements.

While the data structure of Chapter 4 remains the same for all classifiers, HiCuts goes a step further in that it attempts to compute a data structure that varies depending on the structure of the classifier — the structure is itself discovered and utilized while preprocessing the classifier. The HiCuts algorithm combines two data structures for better performance — a tree and a linear search data structure — each of which would not be as useful separately. The resulting HiCuts data structure is the only data structure we know of that simultaneously supports quick updates along with small deterministic classification time and reasonable data structure storage requirements.

CHAPTER 6

Future Directions

As we saw in Section 1 of Chapter 1, the packet processing capacity of IP routers needs to keep up with the exponential increase in data rates of physical links. This chapter sets directions for future work by proposing the characteristics of what we believe would be ‘ideal’ solutions to the routing lookup and packet classification problems.

1 Ideal routing lookup solution

We believe that an ideal routing lookup engine (we restrict our scope to IPv4 unicast forwarding) has *all* of the following characteristics:

- **Speed:** An ideal solution achieves one routing lookup in the time it takes to complete one access in a (random-access) memory in the worst-case. This characteristic implies that an ideal solution lends itself to pipelining in hardware.
- **Storage:** The data structure has little or no overhead in storing prefixes. In other words, the storage requirements are nearly $32N$ bits, or better, for N prefixes in the worst-case. A less stringent, though acceptable, characteristic could be that the storage requirements scale no worse than linearly with the size of the forwarding

table. If the backbone forwarding tables continue to grow as rapidly as we saw in Section 1.2.1 of Chapter 1, exponentially decreasing transistor feature sizes will enable implementations of an ideal routing lookup solution to continue to gracefully meet the demands posed by future routing table growth.

- **Update rate:** Based on current BGP update rates, an ideal solution supports at least 20,000 updates per second[1] in the worst case. Furthermore, updates are atomic in that they do not cause interleaved search operations to give incorrect results.
- **Feasibility of implementation:** Implementations of an ideal lookup solution should be feasible with current technology, e.g., should not consume an unreasonable number of chips, or dissipate unreasonable amount of power, or be too expensive.

Note that amongst the solutions known at the time of writing, ternary CAMs have desirable storage (and possibly update rate) characteristics, but do not have the speed of one RAM access, and do not admit feasible implementations supporting large routing tables. Even though the algorithm proposed in Chapter 2 seems to satisfy all but the update rate requirements, the large storage requirements of this algorithm dictate that the fastest memory technology, i.e., SRAM, cannot be used. This imposes an inherent limitation on the routing lookup rates achievable using this algorithm.

If an ideal solution did exist today, it would consume $32 \times 256K = 8$ Mb of memory for 256K prefixes.[2] Now, 8 Mb of fast SRAM (with 3 ns cycle time) can be easily put on a reasonable sized chip in current 0.18 micron technology. Hence, an ideal solution would be able to lookup 333 million packets per second, enough to process 40 byte minimum-sized TCP/IP packets at line rates of 100 Gbps.[3] In contrast, only 66 million packets per

[1] This is two orders of magnitude greater than the peak of a few hundred reported by Labovitz [47].

[2] 256,000 is more than double the number of prefixes (98,000) at the time of writing (see Section 1.2 of Chapter 1).

[3] Again, this ignores the packet-over-SONET overhead bytes.

second can be looked up by solutions available today — the algorithm proposed in Chapter 2 (using embedded DRAM), and ternary CAMs (using 2-4 chips).

2 Ideal packet classification solution

An ideal packet classification solution not only has all the characteristics we saw above of an ideal lookup solution — that of high speed (classification at line-rate), low storage (to support thousands of classification rules), fast incremental updates, and feasible implementation (cost effective) — but also the characteristics of flexibility in the number and specification syntax of packet header fields supported. In contrast with routing lookups, it is harder to quantify the desirable values of these parameters for packet classification because of the lack of a sufficient amount of statistical data about real-life classifiers. However, it is not unrealistic to imagine a carrier's edge router supporting 1000 ISP subscribers, each with at least 256 five-field classification rules, for a total of 256,000 128-bit rules required to be supported by the classification engine of a router.

In light of the worst-case lower bounds on multi-dimensional classification algorithms mentioned in Chapter 4, an ideal classification solution is unlikely to be able to support all possible worst case combinations of classification rules, and yet satisfy all the other characteristics mentioned above. We believe that intelligent heuristic solutions should be acceptable.

We now see how close the solutions known at the time of the writing of this thesis approach that of an ideal solution. A total of sixteen 2 Mb ternary CAM chips are required to support 256,000 128-bit classification rules. The resulting power dissipation and cost of the system would be clearly excessive. Similarly, the Recursive Flow Classification algorithm of Chapter 4 would require approximately 9 DRAM chips.[1] Though not in terms of

[1] Based on experiments shown in Section 4.3 of Chapter 4, we assume that 4K rules occupy a maximum of approximately 4.5 Mbytes of memory, and that each DRAM chip has a density of 256 Mbits.

power dissipation, this solution is still expensive in terms of board real-estate. The HiCuts algorithm of Chapter 5 would require 4 DRAM chips but would be two to four times slower than a ternary CAM or recursive flow classification solution.

3 Final words

The above mentioned ideal lookup and classification solutions appear challenging to obtain, and will probably require not only improved circuit technologies, but also new data structures and algorithms. Hopefully, this dissertation will serve as a useful foundation for future research in this exciting field in general, and in attempts to obtain these (or similar) ideal solutions in particular.

Appendix A

Proof of correctness for the Optimized one-instruction-update algorithm of Chapter 2

Define $D(m)$, where m is a memory entry, to be the depth of the longest (i.e., deepest) prefix p that covers m. Also define $L(p)$ to be the length of a prefix p.

Claim C1**:** For all prefixes p, $PS(p) \le MS(p)$.
Proof: By definition.

Claim C2**:** For all prefixes p, $ME(p) \le PE(p)$.
Proof: By definition.

Claim C3**:** For all prefixes p and q, either of the following hold:
(1) $PS(p) \le PS(q) \le PE(q) \le PE(p)$ i.e. q is completely contained in p.
(2) $PS(q) \le PS(p) \le PE(p) \le PE(q)$ i.e. p is completely contained in q.
(3) $PS(p) \le PE(p) \le PS(q) \le PE(q)$ i.e. p and q are completely non-overlapping.

Proof: Follows from the definitions of *PS* and *PE* and the basic *parenthesis* property of prefixes.

Claim C4**:** For all memory entries m such that $PS(p) \le m < MS(p)$, $D(m) > L(p)$.
Proof: As $MS(p)$ is the first memory entry covered by prefix p, all memory entries between its prefix start and memory start must be covered by deeper (i.e., longer) prefixes.

Claim C5: If a prefix p is deeper than q and $PS(q) \le PS(p) < MS(q)$, then $MS(q) > PE(p)$; i.e. p has to end (prefix end) before $MS(q)$.
Proof: Follows from the fact that q cannot actually start in memory before any deeper prefix has completely ended.

Now, let the update instruction passed on to the hardware by the processor be *Update(m,Y,Z)*. Before any updates, as m is the first memory entry chosen to be updated by the processor,

$D(m) \leq Y$. If while executing the algorithm, hardware encounters a start-marker marking a new prefix, say q, on a memory entry $m2$, $m2$ equals $MS(q)$ but it may or may not equal $PS(q)$. We will show that $L(q) > Y$ in both cases.

Case (S1): $m2 = PS(q) = MS(q)$
Proof: Since the hardware is not done scanning, it has not yet encountered $PE(p)$. As it has encountered $m2 (= PS(q))$, (C3) tells us that q is wholly contained in p, and so $L(q) > L(p) = Y$.

Case (S2): $PS(q) < m2 = MS(q)$
There are two subcases possible within this case depending upon where $PS(q)$ lies with respect to m:
Case(S2.1) $m < PS(q) < m2 = MS(q)$
Clearly in this case, prefix q is wholly contained in prefix p, and so $L(q) > Y$.
Case (S2.2) $PS(q) \leq m < m2 = MS(q)$. Clearly in this case, p is deeper than q (as m needs to be updated, and m lies in between $PS(q)$ and $MS(q)$). By (C3) and (C5), $PE(p) < MS(q)$, and therefore the hardware should have stopped scanning before reaching $m2$. This subcase is thus not possible at all.
The correctness of the update algorithm now follows immediately. If the hardware, while scanning memory entries encounters a start-marker, it indicates the start of a prefix which is necessarily deeper than Y, and hence is not to be updated. This is exactly what the algorithm does. By updating only when DC equals 1, it ensures that a memory entry is updated only if it had a prefix shallower than Y covering it before the update. ❑

Appendix B

Choice of Codeword Lengths in a Depth-constrained Alphabetic Tree

Lemma 1 A depth-constrained alphabetic tree with maximum depth D satisfies the characteristic inequality of Lemma 3.1 (Chapter 3), when the codeword lengths l_k of the k^{th} letter occurring with probability $q_k\left(q_k \geq 2^{-D} \forall k \right)$ are given by: $l_k^* = \left(\begin{array}{cc} min(\lceil -\log q_k \rceil, D) & k = 1, n \\ min(\lceil -\log q_k \rceil + 1, D) & 2 \leq k \leq n-1 \end{array} \right.$.

Proof: We need to prove that $s_n \leq 1$ where $s_k = c(s_{k-1}, 2^{-l_k}) + 2^{-l_k}$, $s_0 = 0$, and c is defined by $c(a,b) = \lceil a/b \rceil b$. We first prove by induction that

$$s_i \leq \sum_{k=1}^{i} q_k \qquad \forall 1 \leq i \leq n-1$$

For the base case, $s_1 = 2^{-l_1} \leq q_1$ by the definition of l_1. For the induction step, assume the hypothesis is true for $i-1$. By definition, $s_i = c(s_{i-1}, 2^{-l_i}) + 2^{-l_i}$. Now there are two possible cases:

1. $\lceil -\log q_i \rceil + 1 \le D$, and therefore $2^{-(l_i - 1)} \le q_i$. Using the fact that $\lceil a/b \rceil < a/b + 1$, i.e., $c(a, b) < a + b$ for all nonzero real numbers a and b, we get the following using inductive hypothesis:

$$s_i \le s_{i-1} + 2^{-l_i} + 2^{-l_i} \le 2^{-(l_i - 1)} + \sum_{k=1}^{i-1} q_k \le q_i + \sum_{k=1}^{i-1} q_k = \sum_{k=1}^{i} q_k$$

2. $\lceil -\log q_i \rceil + 1 > D$. This implies that $l_i = D$ and hence $q_i \ge 2^{-l_i}$. Also, as s_j is an integral multiple of $2^{-D} \forall j$, $c(s_{i-1}, 2^{-l_i}) = s_{i-1} \le \sum_{k=1}^{i-1} q_k$ and thus:

$$s_i = 2^{-l_i} + c(s_{i-1}, 2^{-l_i}) \le q_i + \sum_{k=1}^{i-1} q_k = \sum_{k=1}^{i} q_k$$

Therefore, $s_{n-1} \le \sum_{i=1}^{n-1} q_i = 1 - q_n \le 1 - 2^{-l_n}$. Also:

$s_n = 2^{-l_n} + c(s_{n-1}, 2^{-l_n}) \le 2^{-l_n} + c(1 - 2^{-l_n}, 2^{-l_n}) = 2^{-l_n} + 1 - 2^{-l_n} = 1$. This completes the proof that these codeword lengths satisfy the characteristic inequality.❑

Bibliography

[1] H. Ahmadi and W.E. Denzel. “A survey of modern high-performance switching techniques,” *IEEE Journal of Selected Areas in Communications*, vol. 7, no. 7, pages 1091-103, September 1989.

[2] G. Apostolopoulos, D. Williams, S. Kamat, R. Guerin, A. Orda and T. Przygienda. “QoS routing mechanisms and OSPF extensions,” *RFC 2676*, http://www.ietf.org/rfc/rfc2676.txt, August 1999.

[3] F. Baker, editor. “Requirements for IP version 4 routers,” *RFC 1812*, http://www.ietf.org/rfc/rfc2676.txt, June 1995.

[4] M. de Berg, M. van Kreveld and M. Overmars. *Computational geometry: algorithms and applications*, Springer-Verlag, 2nd rev. ed. 2000.

[5] D.P. Bertsekas. *Nonlinear programming*, Athena Scientific, 1995.

[6] R. Braden, L. Zhang, S. Berson, S. Herzog and S. Jamin. “Resource reSer-Vation protocol (RSVP) — version 1 functional specification,” *RFC 2205*, http://www.ietf.org/rfc/rfc2205.txt, September 1997.

[7] M.M. Buddhikot, S. Suri and M. Waldvogel. “Space decomposition techniques for fast layer-4 switching,” *Proceedings of Conference on Protocols for High Speed Networks*, pages 25-41, August 1999.

[8] B. Chazelle. “Lower bounds for orthogonal range searching, i: the reporting case,” *Journal of the ACM*, vol. 37, no. 2, pages 200-12, April 1990.

[9] B. Chazelle. “Lower bounds for orthogonal range searching, ii: the arithmetic model,” *Journal of the ACM*, vol. 37, no. 3, pages 439-63, July 1990.

[10] G. Cheung and S. McCanne. “Optimal routing table design for IP address lookups under memory constraints,” *Proceedings of IEEE Infocom*, vol. 3, pages 1437-44, March 1999.

[11] T. Chiueh and P. Pradhan. “High performance IP routing table lookup using CPU caching,” *Proceedings of IEEE Infocom*, vol. 3, pages 1421-8, March 1999.

[12] KC Claffy. “Internet measurement: state of DeUnion,” presentation for Internet Measurement, Oct 99. Available at http://www.caida.org/outreach/presentations/Soa9911/mgp00027.html.

[13] T.H. Cormen, C.E. Leiserson and R.L. Rivest. *Introduction to algorithms*, MIT Press, 1990.

[14] T.M. Cover and J.A. Thomas. *Elements of information theory*, Wiley Series in Telecommunications, 1995.

[15] D. Decasper, Z. Dittia, G. Parulkar and B. Plattner. “Router plugins: a software architecture for next generation routers,” *Proceedings of ACM Sigcomm*, pages 229-40, October 1998.

[16] S. Deering and R. Hinden. “Internet Protocol, version 6 (IPv6) specification,” *RFC 1883*, http://www.ietf.org/rfc/rfc1883.txt, December 1995.

[17] M. Degermark, A. Brodnik, S. Carlsson and S. Pink. “Small forwarding tables for fast routing lookups,” *Proceedings of ACM Sigcomm*, pages 3-14, October 1997.

[18] A. Demers, S. Keshav and S. Shenker. “Analysis and simulation of a fair queueing algorithm,” *Internetworking: Research and Experience*, vol. 1, no. 1, pages 3-26, January 1990.

[19] W. Doeringer, G. Karjoth and M. Nassehi. “Routing on longest-matching prefixes,” *IEEE/ACM Transactions on Networking*, vol. 4, no. 1, pages 86-97, February 1996.

[20] F. DuPont. “Multihomed routing domain issues for IPv6 aggregatable scheme,” *Internet draft,* <draft-ietf-ngtrans-6bone-multi-01.txt>, June 1999.

[21] R.J. Edell, N. McKeown and P.P. Varaiya. “Billing users and pricing for TCP,” *IEEE Journal of Selected Areas in Communications, Special Issue on Advances in the Fundamentals of Networking*, vol. 13, no. 7, pages 1162-75, September 1995.

[22] D. Estrin and D. J. Mitzel. “An assessment of state and lookup overhead in routers,” *Proceedings of IEEE Infocom*, vol. 3, pages 2332-42, May 1992.

[23] A. Feldmann and S. Muthukrishnan. “Tradeoffs for packet classification,” *Proceedings of IEEE Infocom*, vol. 3, pages 1193-202, March 2000.

[24] D.C. Feldmeier. “Improving gateway performance with a routing-table cache,” *Proceedings of IEEE Infocom*, pages 298-307, March 1988.

[25] S. Floyd and V. Jacobson. “Random early detection gateways for congestion avoidance,” *IEEE/ACM Transactions on Networking*, vol. 1, no. 4, pages 397-413, August 1993.

[26] V. Fuller, T. Li, J. Yu and K. Varadhan. “Classless inter-domain routing (CIDR): an address assignment and aggregation strategy,” *RFC 1519*, http://www.ietf.org/rfc/rfc1519.txt, September 1993.

[27] M.R. Garey. “Optimal binary search trees with restricted maximal depth,” *SIAM Journal on Computing*, vol. 3, no. 2, pages 101-10, March 1974.

[28] A.M. Garsia and M.L. Wachs, “A new algorithm for minimum cost binary trees,” *SIAM Journal on Computing*, vol. 6, no. 4, pages 622-42, December 1977.

[29] E. Gerich. “Guidelines for management of IP address space,” *RFC 1466*, http://www.ietf.org/rfc/rfc1466.txt, May 1993.

[30] E.N. Gilbert. “Codes based on inaccurate source probabilities,” *IEEE Transactions on Information Theory*, vol. 17, no. 3, pages 304-14, May 1971.

[31] P. Gupta, S. Lin and N. McKeown. “Routing lookups in hardware at memory access speeds,” *Proceedings of IEEE Infocom*, vol. 3, pages 1240-7, April 1998.

[32] P. Gupta and N. McKeown, “Packet classification on multiple fields,” *Proceedings ACM Sigcomm*, pages 147-60, September 1999.

[33] C. Hedrick. “Routing information protocol,” *RFC 1058*, http://www.ietf.org/rfc/rfc1058.txt, June 1988.

[34] J. Hennessey and D. Patterson. *Computer architecture: a quantitative approach*, Morgan Kaufmann Publishers, 2nd edition, 1996.

[35] Y. Horibe. “An improved bound for weight-balanced tree,” *Information and Control*, vol. 34, no. 2, pages 148-51, June 1977.

[36] T.C. Hu. *Combinatorial Algorithms*, Addison-Wesley, 1982.

[37] T.C. Hu and A.C. Tucker. “Optimal computer search trees and variable length alphabetic codes,” *SIAM Journal on Applied Mathematics*, vol. 21, no. 4, pages 514-32, December 1971.

[38] N.F. Huang, S.M. Zhao, Jen-Yi Pan and Chi-An Su. “A Fast IP routing lookup scheme for gigabit switching routers,” *Proceedings of IEEE Infocom*, vol. 3, pages 1429-36, March 1999.

[39] D.A. Huffman. “A method for the construction of minimum redundancy codes,” *Proceedings Inst. Radio Engineers*, vol. 40, no. 10, pages 1098-101, September 1952.

[40] L. Hyafil and R.L. Rivest. “Constructing optimal binary decision trees is NP-complete,” *Information Processing Letters*, vol. 5, no. 1, pages 15-7, May 1976.

[41] A. Itai. “Optimal alphabetic trees,” *SIAM Journal on Computing*, vol. 5, no. 1, pages 9-18, March 1976.

[42] S. Jamin, P.B. Sanzig, S.J. Shenker and L. Zhang, “A measurement-based admission control algorithm for integrated service packet networks,” *IEEE/ ACM Transactions on Networking*, vol. 5, no. 1, pages 56-70, February 1997.

[43] T. Kijkanjanarat and H.J. Chao. “Fast IP routing lookups for high performance routers,” *Computer Communications*, vol. 22, no. 15-16, pages 1415-22, September 1999.

[44] S. Kirpatrick, M. Stahl and M. Recker. “Internet Numbers,” *RFC 1166*, http://www.ietf.org/rfc/rfc1166.txt, July 1990.

[45] D. Knox and S. Panchanathan. “Parallel searching techniques for routing table lookup,” *Proceedings of IEEE Infocom*, vol. 3, pages 1400-5, March 1993.

[46] D.E. Knuth. *The art of computer programming, vol. 3: sorting and searching*, Addison-Wesley, 3rd edition, 1998.

[47] C. Labovitz, G. Malan and F. Jahanian. “Internet routing instability,” *Proceedings of ACM Sigcomm*, pages 115-26, September 1997.

[48] T.V. Lakshman and D. Stiliadis. “High-speed policy-based packet forwarding using efficient multi-dimensional range matching”, *Proceedings of ACM Sigcomm*, pages 191-202, September 1998.

[49] B. Lampson, V. Srinivasan and G. Varghese. “IP lookups using multiway and multicolumn search,” *Proceedings of IEEE Infocom*, vol. 3, pages 1248-56, April 1998.

[50] L.L. Larmore and T.M. Przytycka. “A fast algorithm for optimum height-limited alphabetic binary trees,” *SIAM Journal on Computing*, vol. 23, no. 6, pages 1283-312, December 1994.

[51] G. Malkin. “RIP version 2. Carrying additional information,” *RFC 1723*, http://www.ietf.org/rfc/rfc1723.txt, November 1994.

[52] A.J. McAuley and P. Francis. “Fast routing table lookup using CAMs,” *Proceedings of IEEE Infocom*, vol. 3, pages 1382-91, April 1993.

[53] S. McCanne and V. Jacobson. “A BSD packet filter: a new architecture for user-level packet capture,” *Proceedings of Usenix Winter Conference*, pages 259-69, January 1993.

[54] E.J. McCluskey. *Logic design principles: with emphasis on testable semicustom circuits*, Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[55] N. McKeown. “Scheduling algorithms for input-queued cell switches,” Ph.D. thesis, University of California at Berkeley, 1995.

[56] N. McKeown, M. Izzard, A. Mekkittikul, B. Ellersick and M. Horowitz. “The Tiny Tera: a packet switch core,” *Proceedings* of *Hot Interconnects V*, Stanford University, pages 161-73, August 1996.

[57] A. Mekkittikul. “Scheduling non-uniform traffic in high speed packet switches and routers,” PhD thesis, Stanford University, 1998.

[58] A. Mekkittikul, N. McKeown and M. Izzard. “A small high-bandwidth ATM switch.” *Proceedings of the SPIE*, vol. 2917, pages 387-97, November 1996.

[59] R.L. Mildiu and E.S. Laber. “Warm-up algorithm: A lagrangean construction of length restricted huffman codes,” *Monografias em Ciencia da Computacao*, no. 15, January 1996.

[60] R.L. Mildiu and E.S. Laber. “Improved bounds on the inefficiency of length-restricted prefix codes,” *unpublished manuscript*.

[61] R.L. Mildiu, A.A. Pessoa and E.S. Laber. “Efficient implementation of the Warm-up algorithm for the construction of length-restricted prefix codes,” *Proceedings of the ALENEX*, Baltimore, Maryland and in vol. 1619, *Lecture Notes in Computer Science*, Springer-Verlag, January 1999.

[62] A. Moestedt and P. Sjodin. “IP address lookup in hardware for high-speed routing,” *Proceedings Hot Interconnects VI*, August 1998.

[63] S. Morgan and M. Delaney. “The Internet and the local telephone network: conflicts and opportunities,” *XVI International Switching Symposium,* pages 561-9, September 1997.

[64] D.R. Morrison. “PATRICIA — practical algorithm to retrieve information coded in alphanumeric,” *Journal of the ACM*, vol. 15, no. 14, pages 514-34, October 1968.

[65] J. Moy. “OSPF version 2,” *RFC 2328*, http://www.ietf.org/rfc/rfc2328.txt, April 1998.

[66] M. Naldi. “Size estimation and growth forecast of the Internet,” *Centro Vito Volterra preprints*, University of Rome, October 1997.

[67] P. Newman, G. Minshall, T. Lyon and L. Huston. “IP switching and gigabit routers,” *IEEE Communications Magazine*, vol. 35, no. 1, pages 64-9, January 1997.

[68] P. Newman, T. Lyon and G. Minshall. “Flow labelled IP: a connectionless approach to ATM”, *Proceedings of IEEE Infocom*, vol. 3, pages 1251-60, April 1996.

[69] S. Nilsson and G. Karlsson. “IP-address lookup using LC-tries,” *IEEE Journal of Selected Areas in Communications*, vol. 17, no. 6, pages 1083-92, June 1999.

[70] H. Obara. “Optimum architecture for input queueing ATM switches,” *IEEE Electronics Letters*, pages 555-7, March 1991.

[71] H. Obara and Y. Hamazumi. “Parallel contention resolution control for input queueing ATM switches,” *IEEE Electronics Letters*, vol. 28, no. 9, pages 838-9, April 1992.

[72] H. Obara, S. Okamoto and Y. Hamazumi. “Input and output queueing ATM switch architecture with spatial and temporal slot reservation control,” *IEEE Electronics Letters*, vol. 28, no. 1, pages 22-4, January 1992.

[73] M.H. Overmars and A.F. van der Stappen. “Range searching and point location among fat objects,” *Journal of Algorithms*, vol. 21, no. 3, pages 629-56, November 1996.

[74] A.K. Parekh and R.G. Gallager. “A generalized processor sharing approach to flow control in integrated services networks: the single node case,” *IEEE/ACM Transactions on Networking*, vol. 1, no. 3, pages 344-57, June 1993.

[75] A.K. Parekh and R.G. Gallager. “A generalized processor sharing approach to flow control in integrated services networks: the multiple node case,” *IEEE/ACM Transactions on Networking*, vol. 2, no. 2, pages 137-50, April 1994.

[76] C. Partridge, P.P. Carvey, E. Burgess, I. Castineyra, T. Clarke, L. Graham, M. Hathaway, P. Herman, A. King, S. Kohalmi, T. Ma, J. Mcallen, T. Mendez, W.C. Milliken, R. Pettyjohn, J. Rokosz, J. Seeger, M. Sollins, S. Storch, B. Tober, G.D. Troxel, D. Waitzman and S. Winterble. “A 50-Gb/s IP router,” *IEEE/ACM Transactions on Networking*, vol. 6, no. 3, pages 237-48, June 1998.

[77] C. Partridge. “Locality and route caches,” *NSF workshop on Internet Statistics Measurement and Analysis*, San Diego, February 1996. Also see http://moat.nlanr.net/ISMA/Positions/partridge.html

[78] M. Patyra and W. Maly. “Circuit design for a large area high-performance crossbar switch,” *Proceedings International Workshop on Defect and Fault Tolerance on VLSI Systems*, pages 32-45, November 1991.

[79] F. Preparata and M. I. Shamos. *Computational geometry: an introduction*, Springer-Verlag, 1985.

[80] Y. Rekhter and T. Li. “A border gateway protocol 4 (BGP-4),” *RFC 1771*, http://www.ietf.org/rfc/rfc1771.txt, March 1995.

[81] Y. Rekhter and T. Li. “An architecture for IP address allocation with CIDR.” *RFC 1518*, http://www.ietf.org/rfc/rfc1518.txt, September 1993.

[82] Y. Rekhter, B. Davie, D. Katz, E. Rosen and G. Swallow. “Cisco systems’ tag switching architecture overview,” *RFC 2105*, http://www.ietf.org/rfc/rfc2105.txt, February 1997.

[83] C. Rose. “Rapid optimal scheduling for time-multiplex switches using a cellular automaton,” *IEEE Transactions on Communications*, vol. 37, no. 5, pages 500-9, May 1989.

[84] H. Samet. *The design and analysis of spatial data structures*, Addison-Wesley, 1990.

[85] B. Schieber. “Computing a minimum weight k-link path in graphs with the concave Monge property,” *Journal of Algorithms*, vol. 29, no. 2, pages 204-22, November 1998.

[86] R. Sedgewick and P. Flajolet. *An introduction to the analysis of algorithms*, Addison-Wesley, 1996.

[87] F. Shafai, K.J. Schultz, G.F.R. Gibson, A.G. Bluschke and D.E. Somppi. “Fully parallel 30-Mhz, 2.5-Mb CAM,” *IEEE Journal of Solid-State Circuits*, vol. 33, no. 11, pages 1690-6, November 1998.

[88] D. Shah and P. Gupta. “Fast incremental updates on ternary-CAMs for routing lookups and packet classification,” *Proceedings of Hot Interconnects VIII*, August 2000. To also appear in *IEEE Micro January/February 2001*.

[89] M. Shreedhar and G. Varghese. “Efficient fair queuing using deficit round-robin,” *IEEE/ACM Transactions on Networking*, vol. 4, no. 3, pages 375-85, June 1996.

[90] K. Sklower. “A tree-based packet routing table for berkeley unix,” *Proceedings of the 1991 Winter Usenix Conference*, Dallas, pages 93-9, 1991.

[91] F. Solensky and F. Kastenholz. “A revision to IP address classifications,” *work in progress,* Internet draft, March 1992

[92] F. Solensky and F. Kastenholz. “Definition of class E IP addresses,” *Internet draft*, available at http://gnietf.vlsm.org/17.txt, August 1991.

[93] V. Srinivasan and G. Varghese. “Faster IP lookups using controlled prefix expansion,” *Sigmetrics*, 1998. Also in *ACM Transactions on Computer Systems*, vol. 17, no. 1, pages 1-40, February 1999.

[94] V. Srinivasan and G. Varghese. “A survey of recent IP lookup schemes,” *Proceedings of Conference on Protocols for High Speed Networks*, pages 9-23, August 1999.

[95] V. Srinivasan, S. Suri, G. Varghese and M. Waldvogel. “Fast and scalable layer four switching,” *Proceedings of ACM Sigcomm*, pages 203-14, September 1998.

[96] V. Srinivasan, S. Suri and G. Varghese. “Packet classification using tuple space search”, *Proceedings of ACM Sigcomm*, pages 135-46, September 1999.

[97] V. Srinivasan. “Fast and efficient Internet lookups,” PhD thesis, Washington University, 1999.

[98] W.R. Stevens and G.R. Wright. *TCP/IP Illustrated, vol. 2, the implementation*, Addison-Wesley, 1995.

[99] H. Suzuki, H. Nagano, T. Suzuki, T. Takeuchi, S. Iwasaki. “Output-buffer switch architecture for asynchronous transfer mode,” *IEEE International Conference on Communications*, vol. 1, pages 99-103, June 1989.

[100] Y. Tamir and G. Frazier. “High performance multi-queue buffers for VLSI communication switches,” *Proceedings of the 15th Annual Symposium on Computer Architecture*, pages 343-54, June 1988.

[101] Y. Tamir and H.C. Chi. “Symmetric crossbar arbiters for VLSI communication switches,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 1, pages 13-27, January 1993.

[102] R.E. Tarjan. “Data structures and network algorithms,” *Society for Industrial and Applied Mathematics*, Pennsylvania, November 1983.

[103] L. Tassiulas and A. Ephremides. “Stability properties of constrained queueing systems and scheduling policies for maximum throughput in mul-

tihop radio networks," *IEEE Transactions on Automatic Control*, vol. 37, no. 12, pages 1936-48, December 1992.

[104] F.A. Tobagi. "Fast packet switch architectures for broadband integrated services digital networks," *Proceedings of the IEEE*, vol. 78, no. 1, pages 133-67, January 1990.

[105] T.P. Troudet and S.M. Walters. "Neural network architecture for crossbar switch control," *IEEE Transactions on Circuits and Systems,* vol. 38, no. 1, pages 42-57, January 1991.

[106] P. Tsuchiya. "A search algorithm for table entries with non-contiguous wildcarding," *unpublished report, Bellcore*.

[107] H.H.Y. Tzeng and T. Przygienda. "On fast address-lookup algorithms," *IEEE Journal of Selected Areas in Communications*, vol. 17, no. 6, pages 1067-82, June 1999.

[108] M. Waldvogel, G. Varghese, J. Turner and B. Plattner. "Scalable high-speed IP routing lookups," *Proceedings of ACM Sigcomm*, pages 25-36, October 1997.

[109] R.L. Wessner. "Optimal alphabetic search trees with restricted maximal height," *Information Processing Letters*, vol. 4, no. 4, pages 90-4, January 1976.

[110] N. Weste and K. Eshraghian. *Principles of CMOS VLSI design: a systems perspective*, Addison-Wesley, Massachusetts, 1993.

[111] D.E. Willard. "Log-logarithmic worst-case range queries are possible in space $\Theta(N)$," *Information Processing Letters*, vol. 17, no. 2, pages 81-4, 1983.

[112] T. Y. C. Woo, "A modular approach to packet classification: algorithms and results," *Proceedings of IEEE Infocom*, vol. 3, pages 1203-22, March 2000.

[113] T. Yamagata, M. Mihara, T. Hamamoto, Y. Murai, T. Kobayashi, M. Yamada and H. Ozaki. "A 288-kb fully parallel content addressable mem-

ory using a stacked-capacitor cell structure," *IEEE Journal of Solid-state Circuits*, vol. 27, no. 12, December 1992.

[114] R. W. Yeung. "Alphabetic codes revisited," *IEEE Transactions on Information Theory*, vol. 37, no. 3, pages 564-72, May 1991.

[115] H.S. Yoon. "A large-scale ATM switch: analysis, simulation and implementation," *Proceedings ICATM-98*, pages 459-64, 1998

[116] H. Zhang. "Service disciplines for guaranteed performance service in packet-switching networks," *Proceedings of the IEEE*, vol. 83, no. 10, pages 1374-96, October 1995.

[117] L. Zhang. "VirtualClock: a new traffic control algorithm for packet-switched networks," *ACM Transactions on Computer Systems*, vol. 9, no. 2, pages 101-24, May 1991.

[118] All vBNS routes snapshot, at http://www.vbns.net/route/Allsnap.rt.html.

[119] Arrow Electronics Inc., at http://www.arrow.com.

[120] Cisco 12000 series GSR, at http://www.cisco.com/warp/public/cc/pd/rt/12000/index.shtml.

[121] Histogram of packet sizes, at http://www.nlanr.net/NA/Learn/plen.970625.hist.

[122] IBM ASIC Standard Cell/Gate Array Products, at http://www.chips.ibm.com/products/asics/products/edram/index.html.

[123] Information Processing Systems — Open Systems Interconnection, "Transport Service Definition," International Organization for Standardization, International Standard 8072, June 1986.

[124] "Internet routing table statistics," at http://www.merit.edu/ipma/routing_table.

[125] IPng working group in the IETF, at http://www.ietf.org/html.charters/ipngwg-charter.html.

[126] Juniper Networks Inc., Internet Processor II: Performance without compromise, at http://www.juniper.net/products/brochures/150006.html.

[127] Lara Networks, at http://www.laratech.com.

[128] Lucent NX64000 multi-terabit switch/router, at http://www.lucent.com/ins/products/nx64000.

[129] Memory-memory, at http://www.memorymemory.com.

[130] Mosaid, at http://www.mosaid.com/semiconductor/networking.htm.

[131] Netlogic microsystems, at http://www.netlogicmicro.com. CIDR products at http://www.netlogicmicro.com/products/cidr/cidr.html.

[132] Network Address Translators IETF working group, at http://www.ietf.org/html.charters/nat-charter.html.

[133] “Nortel Networks breaks own ‘land speed record’ using light — redefines speed of Internet and networking,” press release, Nortel Networks, http://www.nortelnetworks.com/corporate/news/newsreleases/1999d/10_12_9999633_80gigabit.html, October 12, 1999.

[134] NLANR Network Analysis Infrastructure, at http://moat.nlanr.net.

[135] Sibercore Technologies, at http://www.sibercore.com.

[136] Telstra Internet — AS 1221 BGP table data, at http://www.telstra.net/ops/bgptable.html.

[137] Toshiba America Electronic Components, at http://www.toshiba.com/taec.

[138] VTune(TM) Performance Analyzer Home Page, at http://developer.intel.com/vtune/analyzer/index.htm.

देहिनोऽस्मिन् यथा देहे कौमारं यौवनं जरा ।
तथा देहान्तरप्राप्तिर्धीरस्तत्र न मुह्यति ॥

As the embodied soul continuously passes,
in this body, from boyhood to youth to old age,
the soul similarly passes into another body at death.

Lord Krishna to Arjuna in Bhagvad-gita